SRI International

REX PROGRAMMER'S MANUAL

Technical Note 381R

July 1, 1988

By: Leslie Pack Kaelbling and Nathan J. Wilson

Artificial Intelligence Center
Computer and Information Sciences Division
and
Center for the Study of Language and Information
Stanford University

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED

This work was supported in part by a gift from the System Development Foundation, in part by FMC Corporation under Contract 147466 (SRI Project 7390), in part by General Motors Research Laboratories under Contract 50-13 (SRI Project 8662), and in part by DARPA and NASA under NASA Grant PR 5671 (SRI Project 4099).

SRI International®

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

Credits and Trademarks

Sun Workstation ® is a registered trademark of Sun Microsystems, Inc.

UNIX is a trademark of AT&T Bell Laboratories.

June, 1988

Contents

Introduction

This manual describes Rex, a programming language for specifying *machines* by declaratively describing their behavior. The Rex language consists of a set of LISP functions that define primitive Rex machines and provides methods for building complex machines out of simpler components.

A Rex machine is a synchronous abstract device that has inputs, local state, and outputs, all of which are *storage locations*. Storage locations may be thought of as wires that can be set to certain values and whose values can be read by Rex machines. The value of a storage location is determined by its *constraint*, some function of the values of a set (possibly empty) of storage locations.

A Rex machine operates by repeatedly computing a mapping from its inputs and current state into its outputs and next state. By hierarchically dividing a large state into small components and specifying their state transitions, we can "make the combinatorial explosion work for us"[3]. The size of the smallest component may vary from implementation to implementation; it could be a bit, an integer, or a small enumerated type. The state transitions are described by functions that map tuples of elements of the primitive data types into other tuples. The new value of any given component could, in principle, depend on all of the inputs and the entire current state of the machine, but, in practice, the dependencies are usually local.

Any machine that is described by Rex has the property of being *real-time*. By our definition, a machine is real-time if there is a guaranteed constant bound on the length of time between (1) the receipt of a particular input by the machine and (2) the generation of an output of the machine that could have depended upon that input. In the domain of mobile robots, for example, this means that there is a constant bound on the time between when the sensors tell the machine an obstacle is in front of it and when the machine can activate the brakes. Symbolic computation may still take place in such a machine, but it must be constrained to prevent a part of the program from doing arbitrary amounts of computation without allowing other parts to run. We discuss methods for creating such machines in a following section. By adhering to a strict discipline, it is possible to create real-time machines in nearly any language. In Rex it is impossible *not* to. Rex also simplifies the task of real-time programming by allowing dynamic symbolic specification of static run-time structures.

As well as guaranteeing real-time performance, Rex was designed to have a simple and clear structure to facilitate analysis and synthesis of programs with specified properties. It is easily axiomatizable, and work has been done on formalizing epistemic (or informational) properties of machines specified in Rex [4]. Although few programmers actually prove properties of their programs, having a simple semantics makes it easier for them to predict what a program will do.

High-level Rex machine specifications are compiled into low-level machine descriptions. The low-level descriptions are very much like hardware descriptions, specifying connections between components and wires. We run the machines on a standard computer by generating

code that sequentially simulates the circuit described by the low-level description. Since the low-level representation is a circuit description, Rex programs may be easily mapped to computers with differing degrees of parallelism (from a very fine-grained processor to two standard processors) or to custom hardware.

This paper is divided into three sections. The first, a tutorial exposition of Rex explains how to use the Rex compiler, introduces most of the Rex constructs, and gives examples of their use. The second section is a reference manual. It describes each construct in detail, explains the use and invocation of Rex, and discusses error messages that may occur during the course of compilation. The last section consists of examples of Rex programs, and a set of exercises and their solutions. A short appendix at the end of the document describes the errors that can be generated by the compiler.

To aid the reader, the following typographical conventions are used throughout the document. Examples and function definitions that are printed exactly the way they must be typed in are printed in `typewriter font`. Examples are `makem`, `int`, and `(+ 1 2)`. Within such text *italics* are used as unspecified arguments of functions. Examples are *denote-expr*, *type-expr*, and expr. These terms are also printed in *italics* in the normal text folowing function definitions. In normal text *italics* are also used to emphasize selected words.

1 Tutorial Introduction to Rex

This tutorial provides the user with basic information about the Rex language: how Rex programs are compiled and executed, the types of expression allowed in the language, and examples of the language. We describe the use of the compiler and the execution environment prior to describing the language itself so that the reader can experiment with the examples as they appear in the rest of the tutorial. The tutorial assumes that the reader is familar with COMMON LISP, since the current implementation of Rex is embedded in COMMON LISP.

1.1 Compiling Rex Programs

This section describes how a Rex program is compiled to create a machine-code program that simulates the machine described in the Rex program, and how to run this machine on a Sun 3 Workstation.

A Rex program is a special kind of COMMON LISP program. The Rex primitives, special syntax, and special functions are all contained in a COMMON LISP system named REX. After loading this system, you should compile or evaluate a Rex program with the COMMON LISP compiler or interpreter.

Once this has been done you evaluate the top-level function of the Rex program, using the Rex function `makem`. One of the arguments to `makem` is a module name. This name is used to name the resulting files and is the *name* argument given to the UNIX shell commands described below. One of the results of `makem` is a file with the suffix `.robj`. This file is referred to as the *robj-file*. The evaluation of `makem` is referred to as *Rex compile time*.

Once you have an robj-file it must be linked on a Sun Workstation with the shell command `rex_lnk` *name*. `rex_lnk` creates two new files: the *rbin-file*, with the suffix `.rbin` and the *rmap-file*, with the suffix `.rmap`. The rbin-file is a binary file that contains the code that simulates the described machine, and the rmap-file contains debugging information. The use of the rmap-file in debugging is described in the "Compiling and Running Rex Machines" subsection of the "Reference Manual".

To execute the machine contained in the rbin-file you use another shell command, `rex_run -p` *name* `-0`. The purpose of the flags to `rex_run` are described in the section mentioned above. The execution of `rex_run` is referred to as *Rex run time*.

Introductory Example

The following is a small Rex program that expects the user to input a single number each cycle and then outputs the square of the number:

```
(defun squarem (x)
  (timesm x x))
```

```
(defun squarem* ()
  (out (squarem (in int))))
```

This example demonstrates several naming conventions used throughout the examples in this manual. (These conventions make the code easier to interpret, but are in no way required by Rex.) First, the suffix `m` indicates that the function describes a machine, i.e., it returns some type of storage location, and probably at least some of its arguments are storage locations. So in the case of `squarem`, the function accepts a storage location containing an integer and returns a storage location that is constrained to contain the square of the value in the input storage location. A second convention used in this example is the suffix `*`, which usually indicates a top-level machine that can be Rex-compiled using the function `makem` and that tests the function having the name that occurs before the `*`. In all cases these top-level functions handle all of the runtime input and output controlled by the Rex functions `in` and `out`, respectively.

To run the Rex program above, first compile or evaluate the definitions with COMMON LISP. In general, Rex functions are of no use except as arguments to the Rex function `makem` and as components of other Rex functions. Once these functions have been added to your environment, `makem` can be used to create an actual Rex machine with the following form:

```
(makem (squarem*) :module "squarem" :obj "")
```

The `:module` keyword specifies a name for the output files. If no name is provided, the name `rex-mod` is used. The `:obj` specifies the directory the robj-file will be written in. If the keyword is not specified, then no robj-file is created. The directory "" uses the default directory. Once the robj-file has been created it must be linked on a Sun Workstation. If the `:obj` directory is not on a Sun, then the binary file must be copied over to a Sun.

The robj-file created with the previous `makem` is linked with the following shell command:

```
% rex_lnk squarem
```

Now the `squarem` machine is ready to run — execute the shell command:

```
% rex_run -p squarem -O
```

Once the machine is running, you are prompted for input. In this case the input is a single integer; in response to each input, the machine outputs the square of that integer.

In this example, only one machine can result from the top-level function `squarem*`. Often it is more useful to write functions that can result in different machines, depending on the values of the arguments given at Rex compile time. Hence, a Rex function actually specifies a *parameterized class* of Rex machines, one of which will be produced at Rex compile time. In effect, a Rex function specifies a family of components of a given class, for

example resistors, and when the Rex program actually needs a resistor, a particular resistor is produced and put into the machine.

To give an example of a function that defines a class of machines, we need to introduce the use of !. In Rex !*number* creates a storage location that always has the value *number*. The following function defines a class of machines that have a single input, and the output is constrained to be that input plus some constant that is determined at Rex compile time:

```
(defun add-constm (the-const the-input)
  (plusm !the-const the-input))

(defun add-constm* (num)
  (out (add-constm num (in int))))
```

This function may be Rex compiled with the form

```
(makem (add-constm* 3) :module "add-const" :obj "")
```

to make a machine that always adds the value 3 to its input. An `add-constm` machine created this way has a single input, `the-input`, to which it always adds 3. An identical top-level machine could be made from the following forms:

```
(defun add-values (the-input1 the-input2)
  (plusm the-input1 the-input2))

(defun add-constm* (num)
  (out (add-values !num (in int))))
```

The only real change has been to move the ! from the subfunction into `add-constm*`. However, the resulting submachine, `add-values`, has changed quite significantly; it now has two inputs, and the output is constrained to contain their sum. The constant storage location is created in `add-constm*`.

1.2 Expressions

Rex has four kinds of expression: value expressions, type expressions, denoting expressions, and constraining expressions.

1.2.1 Value Expressions

Value expressions designate compile-time values that can range over the full domain of LISP objects. Numeric value expressions are often used as initial values of storage expressions; other value expressions are used to control the structure of the machine being described.

1.2.2 Type Expressions

Type expressions define different data-types of storage locations. In the current implementation of Rex there are four primary types: *boolean* (`bool`), *integer* (`int`), *float* (`float`), and *string* (`str`). The constants for booleans are `0b` and `1b`; integers are 32-bits long and signed; floats are standard 32-bit IEEE floating point, with a sign bit, 8 bits for the exponent, and 23 bits for the mantissa; and strings are double-quoted strings of characters. Strings are only used in very special cases, and no run-time string manipulation functions exist for Rex.

In addition to the atomic types, it is possible to construct more complex type expressions from atomic ones. In the current implementation of Rex, complex types are simply LISP lists of other types. Two types are considered to be equal if they have the same list structure and if the corresponding atomic elements have the same primary type. Since complex types are simply LISP lists, standard LISP functions like `car`, `cdr`, `cons`, `list`, and `append` can be used on type expressions.

Several Rex forms return type expressions or declare new types. First, there is a special Rex tupling syntax, [*type-expr*$_1$ ... *type-expr*$_n$]. The square brackets simply perform the `list` function from LISP.

A second type-expression form is the function (`list-type` *value-expr type-expr*). This function returns a complex type that is a list of length *value-expr* of *type-expr*s.

Structured data types can be declared with the function

(`define-rex-struct` *atom* ((*fieldname*$_1$ *type-expr*$_1$)...(*fieldname*$_n$ *type-expr*$_n$)))

The parameter, *atom*, is bound to the type-expression [*type-expr*$_1$ ...*type-expr*$_n$]. In addition, a set of selector functions is created. The names of the functions are *atom-fieldname*$_1$ ...*atom-fieldname*$_n$. They all take a storage location of the newly defined type, *atom*, and return the component storage location of the type-expression corresponding to the named field. Finally, a constructor function named `make-`*atom* is created. This function returns a storage location of type *atom*, with the fields set to the denoting expressions given as arguments in the appropriate order.

The following are some examples of type expressions:

```
int
[[int int] float]
(list-type 5 [bool int])
[int (list-type 5 [bool int]) bool]
```

The following example of a complete machine specification includes complex types. This function calculates the distance between two points:

```
(define-rex-struct point ((x-coord int)
                          (y-coord int)))
```

```
(defun distancem (point1 point2)
  (sqrtm (plusm (squarem (minusm (point-x-coord point1) (point-x-coord point2)))
                (squarem (minusm (point-y-coord point1) (point-y-coord point2))))))

(defun distancem* ()
  (some* ((pt1 point) (pt2 point))
    (== [pt1 pt2] (in [point point]))
    (out (distancem pt1 pt2))))
```

1.2.3 Denoting Expressions

A *denoting expression* denotes a storage location, together with constraints on its behavior with respect to other storage locations. A storage location can be of any of the types mentioned in the previous subsection.

A storage location that is constrained to contain a constant value is represented by `!`*value-expr*, where the LISP value of *value-expr* is an integer, floating point number, or boolean ('0b '1b). Thus, `!(+ 2 3)` denotes a storage location whose value is always 5.

The second kind of simple denoting expression in Rex is actually a class of expressions that describe *primitive-function machines*. Primitive-function machines are the lowest-level building blocks of a Rex machine description and map directly to functional components in the low-level machine description. As an example, `(plusm a b)` denotes a storage location that is constrained to always contain the sum of the values of the storage locations denoted by `a` and `b`. Another commonly used primitive-function machine is `(ifm a b c)`, which denotes a storage location that is constrained to contain the value of `b` if the run-time value of `a` is equal to 1, otherwise the value of `c`. The regular `if` form from COMMON LISP allows conditional machine construction at compile time, but note that the `ifm` construct works entirely at run time. There are primitives for integer and floating point arithmetic functions, boolean functions, and relational functions. By convention, an identifier *namem* names a primitive-function machine related to the function *name*.

New function machines can be created with the standard COMMON LISP `defun` form. Such functions are referred to as *denoting functions*. When doing this keep in mind which arguments are value-expressions and which are denoting-expressions. Value-expression arguments always get compiled away during Rex compilation. Typically, value-expressions are part of a denoting-expression using the `!` operator or are used to control the layout of the machine (e.g. the number of times a sub-component appears). The denoting-expression arguments, on the other hand, indicate storage locations that will be used by the resulting machine during Rex runtime.

Finally, square brackets can be used to create denoting expressions that denote storage locations with complex types. For example, the denoting expression `[!'1b !2.3 !2]` denotes a complex storage location of type `[bool float int]` whose value is (1b 2.3 2).

For convenience, the Rex predicate `(var-p` *denote-expr*`)` returns `t` if the type of the designated storage location is `bool`, `int`, or `float`; and `nil` otherwise. An example of

where this predicate is useful is the following function, `sum-structm`, which sums all the elements of any type of structure. The type of the structure is specified at compile time as an argument to the top-level function.

```
(defun sum-structm (thing)
  (cond ((null thing) !0)
        ((var-p thing) thing)
        (t (plusm (sum-structm (car thing)) (sum-structm (cdr thing))))))

(defun sum-structm* (type)
  (out (sum-structm (in type))))
```

As with type expressions, denoting expressions can be manipulated with standard LISP selection functions, like `car`, `cdr`, `first`, `second`, `third...`, `nth`. All of these are evaluated at Rex compile time. For run-time list selection, Rex has the form

`(selectm` *denote-expr*$_1$ *denote-expr*$_2$`)`

The run-time value of a `selectm` machine is the value of the nth component of *denote-expr*$_2$, where n is the value of *denote-expr*$_1$. *Denote-expr*$_1$ in `selectm` is constrained to be a list of identical structures. This constraint is a necessary result of the fact that a Rex denoting expression must designate a storage location of a particular type.

1.2.4 Constraining Expressions

A *constraining expression* designates a collection of behavioral constraints on storage locations. The constraints specify the way in which storage locations relate to other storage locations. The basic constraining expression, referred to as *structure equation*, is of the form `(==` *denote-expr*$_1$ *denote-expr*$_2$`)`. This form constrains the storage locations designated by the denoting expressions to be *behaviorally equivalent*: at every point in time, each storage location in *denote-expr*$_1$ is constrained to contain the same value as the corresponding storage location in *denote-expr*$_2$. We can use this form to name the intermediate result of a computation, for example, `(== a (plusm b c))`. It is a programming error to attempt to constrain two storage locations to be behaviorally equivalent if they are already constrained to behave in a way that precludes this possibility. The current implementation of Rex imposes the slightly stronger requirement that at least one of each pair of corresponding atomic storage locations in the two denoting expressions in a `==` form must designate a storage location with no constraints. Thus, the form `(== (plusm a b) (plusm b a))` is not allowed, even though the constraint it expresses is trivially satisfied. The storage locations used in a structure equation can be of any type, so forms such as `(== a [!1 !2])` are legal.

When complex storage locations are referred to by atoms, the operation of structure equation is similar to unification. For example, the constraining expression

```
(== [a [b c]] [[d e f] g])
```

constrains a to be a triple whose components are named by d, e, and f, and constrains g to be a pair with components named by b and c. It also constrains b to be (first g), c to be (second g), etc. One aspect of structure equation is that confusing code like the following is legal:

```
(== [a a] [[b c] [!3 !4]])
```

This expression causes a to always have the value [!3 !4]; b to have the value of location denoted by (first a), i.e. !3, and c to have the value denoted by (second a), i.e., !4. Note that for each constraint, one of the storage locations still must be unconstrained. In addition, the length of the two arguments must be equal.

Behavioral equivalence in structure equation is often implemented in the resulting machine as actual equivalence of parts of the machine, so == can also be viewed as a way of giving another name to a particular storage location.

Writing functions that constrain their inputs is often convenient, some LISP functions, for example, can take a set of unconstrained inputs and a set of constrained inputs and do nothing other than constrain the unconstrained inputs as a function of the constrained inputs. The invocations of such *contraining functions* are then constraining expressions. Top-level functions discussed previously, are an important kind of constraining function. Such functions are unusual, since the storage locations they constrain are not passed as parameters but instead are declared with the functions in and out. (See the example in the next subsection for an example of a function that is also a constraining expression.) The result returned by such constraining expressions is arbitrary, since the storage locations being defined are passed as arguments to the function.

The basic constraining expression, the function ==, however, does have a defined result, namely, a storage location that is behaviorally identical to the storage locations being constrained. This feature is provided primarily for use in denoting functions so that the resulting storage location does not have to be explicitly listed as the last form in the function. This property of structure equation should be used sparingly and only when meaning is very obvious. (See the following discussion of some*.)

1.3 Creating and Naming Storage Locations

Naming intermediate values in a computation is often convenient and sometimes necessary. The let construct of LISP provides that facility for value expressions. The some* construct provides a similar facility for storage locations. The form

(some* ((*atom*$_1$ *type-expr*$_1$)...(*atom*$_n$ *type-expr*$_n$))
  constr-expr$_1$ ... *constr-expr*$_m$ *denote-expr*)

performs the conjunction of the constraining expressions and allows the *atom*$_i$'s to be used to name intermediate storage locations. The result of some* is the value of *denote-expr*. In the case where *constr-expr*$_m$ also denotes the desired result (see previous subsection) the final *denote-expr* is not necessary.

We can use some* to compute two values in terms of some common value, as in the following calculation of the roots of a quadratic:

```
(defun rootsm (a b c r1 r2)
  (some* ((radical float)
          (neg-b float)
          (divisor float))
    (== radical (sqrtm (minusm (squarem b) (timesm !4 (timesm a c)))))
    (== neg-b (unary-minusm b))
    (== divisor (timesm !2 a))
    (== r1 (divm (plusm neg-b radical) divisor))
    (== r2 (divm (minusm neg-b radical) divisor))))
```

In this example r1 and r2 are unconstrained storage locations that are passed as parameters to the function rootsm along with the already constrained storage locations: a, b, and c. Thus any calls to the function rootsm are constraining expressions, as in the following top-level function:

```
(defun roots* ()
  (some* ((a float)
          (b float)
          (c float)
          (r1 float)
          (r2 float))
    (== [a b c] (in [float float float]))
    (rootsm a b c r1 r2)
    (out [r1 r2])))
```

This program points out another interesting aspect of Rex. In most computer languages a fatal error is generated by some mathematical expressions, including taking the square root of a negative number. Rex, however, automatically checks for such error conditions and avoids generating fatal errors. Thus, the above program always produces some output for any input. If a particular input causes the output to be a function of the square root of a negative number, (e.g., the inputs [1.0 0.0 1.0]) then the content of the output is undefined, but the machine still returns two floating point numbers. The programmer can make the output always defined by testing the input to machines that may return an undefined result. For example, here is a modified rootsm function that returns 0 for both roots if the input would cause a negative square root to be calculated:

```
(defun rootsm (a b c r1 r2)
  (some* ((discriminant float)
   (radical float)
          (neg-b float)
          (divisor float))
    (== discriminant (sqrtm (minusm (squarem b) (timesm !4 (timesm a c)))))
    (== radical (ifm (<m discriminant 0) !0 (sqrtm discriminant)))
    (== neg-b (unary-minusm b))
```

```
(== divisor (timesm !2 a))
(== r1 (divm (plusm neg-b radical) divisor))
(== r2 (divm (minusm neg-b radical) divisor))))
```

1.4 Machines with State

All machines that we can specify using the constructs discussed so far are pure functional machines — that is, the outputs depend only on the most recent set of inputs, independent of previous inputs and outputs of the machine. Remember that a Rex machine acts by repeatedly calculating a mapping from the current inputs and state to a set of outputs. Each time period from the perception of a set of inputs to the generation of the next set of outputs is called a *tick*. Here we introduce a construct that allows us to constrain storage locations that preserve state over time. An instance of the form (`init-next` *value-expr denote-expr*) designates a storage location that is constrained to contain the value of *value-expr* initially and to contain at tick $t + 1$ the value of the location designated by *denote-expr* at tick t. If *denote-expr* designates a complex storage location comprised of atomic storage locations, the structure of *value-expr* must be the same as that of *denote-expr*. Thus, `(init-next '(1 2 (3 4)) [a b [c d]])` is valid if each of `a`, `b`, `c`, and `d` designates an integer atomic storage location. The storage location designated by the `init-next` form has the same structure as its arguments. Note that each time the machine is started with `rex_run`, the state is reset to the value of *value-expr*.

An `init-next` form must be used whenever the state of a storage location depends on the previous value of another storage location. To avoid race conditions, it is an error to have the value of any storage location be directly dependent on itself; it must depend on a previous value of that location. Thus `(== x (plusm x !1))` causes an error; to specify a machine that counts by 1, what should be written instead is

```
(== x (init-next 0 (plusm x !1))).
```

As a simple example of the use of the `init-next` form, we define the `everm` machine:

```
(defun everm (input)
  (some* ((ever? bool))
    (== ever? (init-next '0b (orm ever? input)))))

(defun everm* ()
  (out (everm (in bool))))
```

This machine has a single atomic storage location as a parameter. An instance of `everm` denotes a storage location that contains `1b` if the storage location given as `input` has ever contained `1b` in any *previous* tick, and `0b` otherwise. The `orm` expression denotes a storage location that contains `1b` if either of its two argument locations does, and `0b` otherwise. Thus, the `ever?` location contains `1b` if it contained `1b` last time, or if the input was `1b` last time. The machine described by this definition has its output fed back into itself; the structure is shown in Figure 1. Note that the output from this machine just after the first

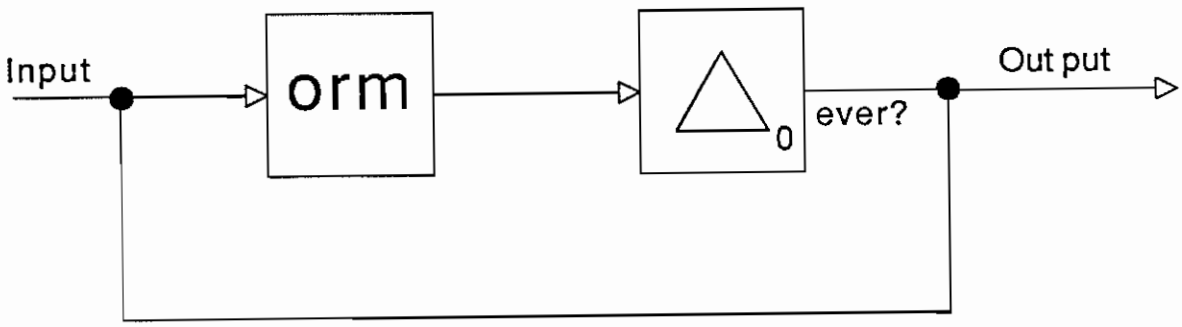

Figure 1: Schematic of the everm machine

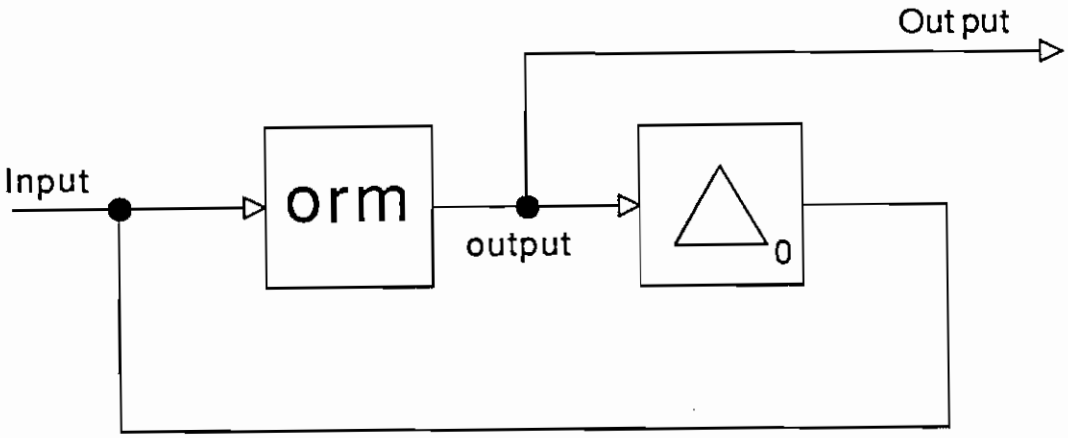

Figure 2: Schematic of the ever2m machine

1b is given is 0b, since the delay component is between the input and the output. To create a function that responds immediately to its input, you must name the output of the orm expression and output that storage location, instead of the one denoted by the init-next expression, as follows:

```
(defun ever2m (input)
  (some* ((output bool))
    (== output (orm input (init-next '0b output)))))

(defun ever2m* ()
  (out (ever2m (in bool))))
```

The structure for this machine is given in Figure 2. Note that the only change has been to swap the order of the init-next and the orm functions.

1.5 Recursive Examples

In this section we present more complex examples that illustrate the use of all constructs discussed in this tutorial.

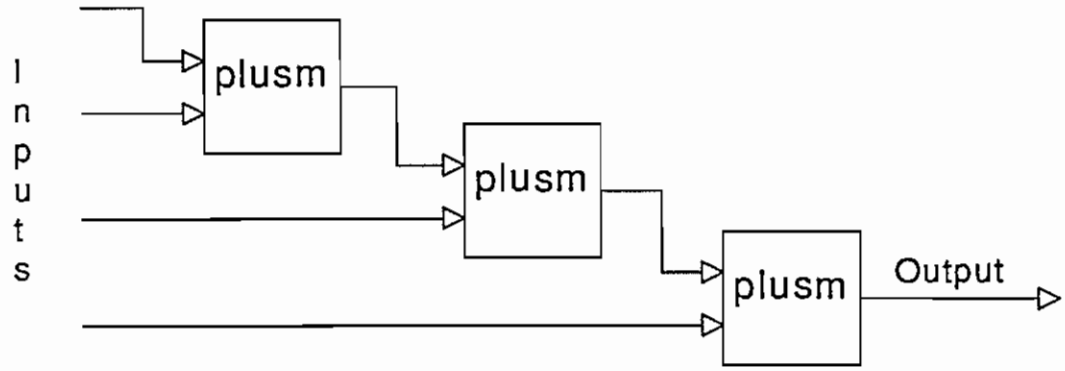

Figure 3: Schematic diagram of the machine specified by `(sum-list* 4)`

The first example uses a recursive definition to describe a machine that sums the contents of a list of atomic storage locations. The length of the list may vary at compile time, but is fixed in the run-time machinery.

```
(defun sum-listm (list)
  (if (= (length list) 1)
      (car list)
      (plusm (car list) (sum-listm (cdr list)))))

(defun sum-listm* (length)
  (out (sum-listm (in (list-type length int)))))
```

The `sum-listm` function defines a class of machines that takes any size of list of either integers or floats and returns the sum of the elements of the list.[1] This definition corresponds very closely to the LISP definition of a function that sums a list of numbers. Note, that all of the recursion is done at compile time. A schematic version of the machine created by the invocation `(sum-listm* 4)` is shown in Figure 3.

In the second example we present two parameterized denoting functions for selecting the nth element of a list. One of them does the selection at compile time; the other at run time. If the selection occurs at compile time, no run-time structure for selection is generated; if it occurs at run time, it generates a linear address-decoding network.

```
(defun compile-time-select* (index length type)
  (out (nth index (in (list-type length type)))))


(defun recur-select (current-index index list)
  (if (= 1 (length list))
      (car list)
      (ifm (>=m !current-index index)
```

[1] If we were implementing this in parallel hardware, it would be more efficient to use a combining scheme that has log(length) depth.

```
          (car list)
          (recur-select (+ current-index 1) index (cdr list)))))

(defun run-time-select (index list)
  (recur-select 0 index list))

(defun run-time-select* (length type)
  (some* ((the-index int)
          (the-list (list-type length type)))
    (== [the-index the-list] (in [int (list-type length type)]))
    (out (run-time-select the-index the-list))))
```

The function `compile-time-select*` has both a value-expression parameter and a denoting-expression parameter. The value-expression indicates the index of the element of the denoting-expression that is always returned by a given instance of `compile-time-select`. If the index is 0 or below, then the first element is returned. If the index is greater than the length of the list, then the last element is returned. Note that the abstract submachine created by `compile-time-select*` has a single input, namely `list`, while the function has two parameters.

The function `run-time-select`, on the other hand, has two denoting-expression parameters. The component created by this function has two inputs: `list` and `index` (which selects the element of `list` to return). A machine that selects from a set of four integers is shown in Figure 4.

1.6 Current Implementation and Future Directions

We have implemented a Rex compiler that runs in any standard implementation of COMMON LISP. It generates low-level structural descriptions that may be fed into a back-end that generates optimized MC68020/MC68881 code to simulate the structure in software. We have used this system to generate programs that control SRI's mobile robot.

Rex is very similar to hardware description languages [2] and so lends itself particularly well to implementation on various parallel architectures. On a fine-grained parallel machine such as the Connection Machine [1], each primitive function element and delay element could be mapped to a processor. The routing would have to be set up once before the machine began running and could remain static after that. In principle, Rex machine descriptions could also be run on multiple, medium-sized processors by partitioning the circuit into as many pieces as there are processors.

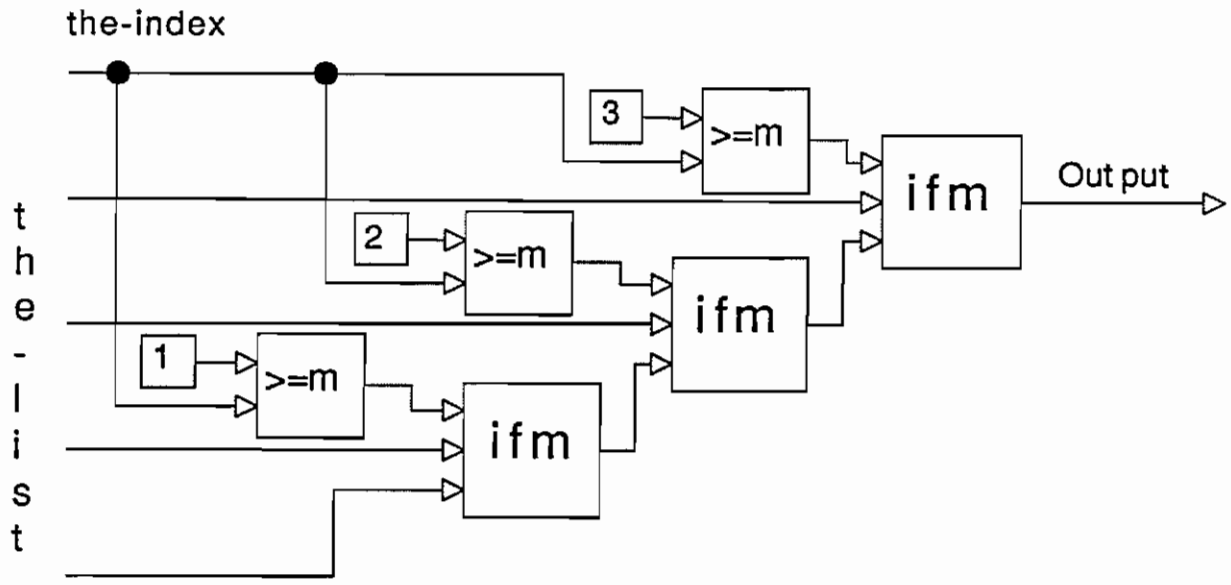

Figure 4: Schematic of the machine specified by `(run-time-selectm* 4 int)`

2 Reference Manual

2.1 Rex Forms

This section describes all of the standard Rex forms. Examples of many these forms are given in the preceding tutorial and in the problems and solutions that follow this section. All of these forms contain at least one denoting expression. In the syntax given they are all referred to as *denote-expr*. If there are further restrictions on the type of the denoting expression, it is given either in the paragraph at the beginning of each subsection or as part of the form description. *Boolean storage locations* refer to storage locations that can contain the values 0b or 1b. *Numeric storage locations* refer to storage locations having an integer or a float value.

2.1.1 Primitive-function machines

The family of primitive-function machine specifiers

$$(primfn\ denote\text{-}expr_1 \ldots denote\text{-}expr_n)$$

denotes locations constrained to always contain the result of applying the function associated with *primfn* to the values of the locations denoted by $denote\text{-}expr_1 \ldots denote\text{-}expr_n$. By convention, an identifier *namem* names a machine specifier intuitively related to the function *name*. The denoting expressions created from primitive-function machine specifiers all denote *atomic* storage locations; they also require atomic storage locations as arguments. Following is a discussion of the primitive-function machine specifiers currently available in Rex.

Arithmetic Operators

All of the arithmetic forms require numeric storage locations as arguments and refer to numeric storage locations. If the binary operators in this section are given one `int` and one `float`, the `int` is coerced to a `float` and the result is a `float`. If the arguments are of the same type, then the result is the same type as the arguments.

(`plusm` $denote\text{-}expr_1$ $denote\text{-}expr_2$) Denotes a storage location constrained to contain the sum of the values of $denote\text{-}expr_1$ and $denote\text{-}expr_2$.

(`minusm` $denote\text{-}expr_1$ $denote\text{-}expr_2$) Denotes a storage location constrained to contain the difference of the values of $denote\text{-}expr_1$ and $denote\text{-}expr_2$.

(`timesm` $denote\text{-}expr_1$ $denote\text{-}expr_2$) Denotes a storage location constrained to contain the product of the values of $denote\text{-}expr_1$ and $denote\text{-}expr_2$.

(divm *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain the rounded quotient of the values of *denote-expr*$_1$ and *denote-expr*$_2$. If *denote-expr*$_2$ has a value of 0 then the result is undefined, but no error is generated.

(modm *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain the value of *denote-expr*$_1$ modulo the value of *denote-expr*$_2$. Both *denote-expr*$_1$ and *denote-expr*$_2$ must be of type `int`. Modulo is defined in Rex to be the number, z, between 0 and *denote-expr*$_2$$-1$ such that the relation k(*denote-expr*$_2$)$+z =$*denote-expr*$_1$ holds for some integer k.

(sqrtm *denote-expr*) Denotes a storage location constrained to contain the square root of the value of *denote-expr*. If *denote-expr* is negative then the result is undefined but no error is generated.

(unary-minusm *denote-expr*) Denotes a storage location constrained to contain the negation of the value of *denote-expr*.

(absm *denote-expr*) Denotes a storage location that contains the absolute value of *denote-expr*.

Bit Operators

The following operators all require and denote integer storage locations.

(bitandm *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain the bit-wise *and* of *denote-expr*$_1$ and *denote-expr*$_2$.

(bitcompm *denote-expr*) Denotes a storage location constrained to contain the bit-wise complement of *denote-expr*.

(bitiorm *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain the bit-wise inclusive *or* of *denote-expr*$_1$ and *denote-expr*$_2$.

(bitxorm *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain the bit-wise exclusive *or* of *denote-expr*$_1$ and *denote-expr*$_2$.

(lshm *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain *denote-expr*$_1$ shifted left by *denote-expr*$_2$ bits. Bits shifted off the left end are lost, and the new bits on the right are set to 0.

(rshm *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain *denote-expr*$_1$ shifted right by *denote-expr*$_2$ bits with sign extension. Bits shifted off the right end are lost, and the new bits on the right are set to 0 if *denote-expr*$_1$ was positive and 1 if it was negative.

Trigonometric Operators

The trigonometric operators accept any numeric storage locations as input and denote storage locations of the same type. When floats or a mixture of floats and integers are used, angle measurements are assumed to be in radians. When only integers are used the trigonometric machines implement functions that use a scaled representation for real values. Integer angles are expressed in terms of RAUs (robot angular units) — the value of an angular argument can range from 0 to 4096, with π radians equal to 2048 RAUs. The outputs of `sinm`, `cosm`, and `tanm` range from positive to negative 4096, being the range $[-1, 1]$ scaled by 4096. The inputs to `atan2m` may be anything but are similarly scaled. The results of the inverse functions are angles in RAUs, so it always remains the case that $f(arc\text{-}f(x)) = x$.

(`sinm` *denote-expr*) Denotes a storage location constrained to contain the sine of the value of *denote-expr*.

(`cosm` *denote-expr*) Denotes a storage location constrained to contain the cosine of the value of *denote-expr*.

(`atan2m` *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain the arctangent of the quotient of the values of *denote-expr*$_2$ and *denote-expr*$_1$. The signs of the values of *denote-expr*$_1$ and *denote-expr*$_2$ are used to derive quadrant information.

Boolean Operators

The representation of boolean values in Rex is `1b` for true and `0b` for false. The use of any other values in functions that take boolean arguments is undefined. The following primitive-function machine specifiers require that the contents of the argument locations be boolean values. The contents of the denoted locations are boolean as well.

(`orm` *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain the disjunction of the values of *denote-expr*$_1$ and *denote-expr*$_2$.

(`andm` *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain the conjunction of the values of *denote-expr*$_1$ and *denote-expr*$_2$.

(`notm` *denote-expr*) Denotes a storage location constrained to contain the negation of the value of *denote-expr*.

Relational Operators

The following primitive-function machine specifiers express relational functions on the values of storage locations. All of these functions denote storage locations of type `bool`. The function `equalm` accepts any two expressions that denote storage locations with the same structure. Two complex storage locations are considered equal if all corresponding atomic

elements of the structure have the same value. The other relational operators accept only numeric or boolean storage locations. If one of the inputs is an `int` and the other is a `float`, then the `int` is coerced to a `float` and their values are compared. When `equalm` is used on complex storage locations, corresponding atomic elements are similarly coerced.

(`equalm` *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain 1b if the values of *denote-expr*$_1$ and *denote-expr*$_2$ are equal, 0b otherwise.

(`not-equalm` *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain 1b if the values of *denote-expr*$_1$ and *denote-expr*$_2$ are not equal, 0b otherwise.

(`>m` *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain 1b if the value of *denote-expr*$_1$ is greater than the value of *denote-expr*$_2$, 0b otherwise.

(`>=m` *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain 1b if the value of *denote-expr*$_1$ is greater than or equal to the value of *denote-expr*$_2$, 0b otherwise.

(`<m` *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain 1b if the value of *denote-expr*$_1$ is less than the value of *denote-expr*$_2$, 0b otherwise.

(`<=m` *denote-expr*$_1$ *denote-expr*$_2$) Denotes a storage location constrained to contain 1b if the value of *denote-expr*$_1$ is less than or equal to the value of *denote-expr*$_2$, 0b otherwise.

Conditional Operator

Run-time conditional expressions are implemented through the `ifm` machine constructor. The first argument location must contain a boolean value; the others can be any denoting expressions that have the same type. The result is the same type as the arguments.

(`ifm` *denote-expr*$_1$ *denote-expr*$_2$ *denote-expr*$_3$) Denotes a storage location constrained to contain the value of *denote-expr*$_2$ if the value of *denote-expr*$_1$ is 1b, otherwise the value of *denote-expr*$_3$.

Conversion Operators

(`float-to-intm` *denote-expr*) Denotes a storage location of type `float` constrained to contain the integer closest to the value of *denote-expr*. If the fractional portion is exactly .5, then the closest even integer is returned.

(`int-to-floatm` *denote-expr*) Denotes a storage location of type `float` constrained to contain the value of the given atomic integer storage location, *denote-expr*.

Complex Storage Locations and Type Constructors

[*denote-expr*$_1$ ... *denote-expr*$_n$] Denotes the complex storage location that is the list of the storage locations denoted by *denote-expr*$_1$ ... *denote-expr*$_n$.

[*type-expr*$_1$ ... *type-expr*$_n$] Denotes the complex type expression that is the list of types *type-expr*$_1$ ... *type-expr*$_n$.

(`list-type` *value-expr type-expr*) Denotes a complex type expression that is a *value-expr* element list of *type-expr*.

(`define-rex-struct` *atom* ((*fieldname*$_1$ *type-expr*$_1$)...*fieldname*$_n$ *type-expr*$_n$)) Binds *atom* to [*type-expr*$_1$ ... *type-expr*$_n$]. In addition, selector functions are created with the names *atom-fieldname*$_1$ through *atom-fieldname*$_n$ which take a storage location that has the type *atom* and return the appropriate constituent storage location. Finally, a constructor function, `make-`*atom* is created which expects one argument for each *fieldname* in the same order and of the appropriate type.

(`var-p` *denote-expr*) Returns `t` if *denote-expr* is of type `bool`, `int`, or `float`.

(`var-type` *denote-expr*) Returns the type of *denote-expr*.

Other Rex Forms

The remaining Rex forms are presented briefly here. For further clarification, see the tutorial section of this manual.

(`==` *denote-expr*$_1$ *denote-expr*$_2$) Constrains the storage locations denoted by *denote-expr*$_1$ and *denote-expr*$_2$ to be behaviorally equivalent.

(`some*` ((*atom*$_1$ *type-expr*$_1$)...(*atom*$_k$ *type-expr*$_k$)) *constr-expr*$_1$ ... *constr-expr*$_n$ *denote-expr*) Generates k new storage locations, with names *atom*$_1$ ... *atom*$_k$ of the types specified by *type-expr*$_1$ ... *type-expr*$_k$, respectively, and imposes the conjunction of the constraints of *constr-expr*$_1$ ... *constr-expr*$_n$ on them. The *atom*$_1$ ... *atom*$_k$ denote storage locations and may be used as storage expressions within the scope of the `some*` form. The result of the entire form is the final *denote-expr*. If the form is not being used as a denoting expression or the final constraining expression is also a structure equation, then the final denoting expression may be omitted.

(`init-next` *value-expr denote-expr*) Denotes a storage location that is constrained to contain *value-expr* initially and to contain at tick $t + 1$ the value at tick t of the location denoted by *denote-expr*. Currently, strings may not be used in any part of an `init-next` expression.

2.1.2 Utilities

The forms described in the preceding sections are sufficient for writing any Rex machine constructor. We have included a small set of utilities as well, to simplify programming. Since each of these utilities is implemented in Rex, we include the code in the section "Programming Examples and Exercises" as an example of Rex programming.

To simplify the use of numeric constants, we use the notation !*value-expr* as an abbreviation for the constructor of a machine that has a constant output equal to the value of *value-expr*. It is equivalent to the form

`(some* ((x` *type*`)) (== x (init-next` *value-expr* `x))).`

For convenience in manipulating numeric values, we include the following machine constructors:

`(minm` *denote-expr*$_1$ *denote-expr*$_2$`)` Denotes a storage location that contains the minimum of *denote-expr*$_1$ and *denote-expr*$_2$.

`(maxm` *denote-expr*$_1$ *denote-expr*$_2$`)` Denotes a storage location that contains the maximum of *denote-expr*$_1$ and *denote-expr*$_2$.

`(signm` *denote-expr*`)` Denotes a storage location that contains the value 1 if *denote-expr* is greater than zero, 0 if it is equal to zero, and -1 if it is less than 0.

To select one storage location out of a set of storage location, there is the following function:

`(selectm` *denote-expr*$_1$ *denote-expr*$_2$`)` Denotes the *denote-expr*$_1$th element of *denote-expr*$_2$. Note that *denote-expr*$_2$ must be a list of denoting expressions that all denote storage locations of the same type.

The next two Rex form are complex conditionals similar to COMMON LISP's `case` and `cond` statements. The primary differences are that a default expression must always exist, and each consequent is only a single denoting expression.

`(casem` *denote-expr*$_s$ (*case-expr*$_1$ *denote-expr*$_1$)...(*case-expr*$_n$ *denote-expr*$_n$) *denote-expr*$_d$`)` The case expressions in this form are either a single denoting expression or a list of one or more denoting expressions. This form operates by comparing the value of the selector denoting expression, *denote-expr*$_s$, with the denoting expressions in each of the case expressions. The first case expression that has a denoting expression that is `equalm` (numberically equivalent) to the selector denoting expression is chosen, and the value of its consequent denoting expression is used as the value of the storage location denoted by the `casem` expression. If none of the denoting expressions in the case expressions are `equalm` to the selector expression, then *denote-expr*$_d$ is used as the value of the storage location denoted by the `casem` expression.

(condm (*denote-expr*$_{t1}$ *denote-expr*$_{c1}$)...(*denote-expr*$_{tn}$ *denote-expr*$_{cn}$) *denote-expr*$_d$)
All of the *denote-expr*$_t$'s must denote storage locations of type bool. In addition, all the *denote-expr*$_c$'s and *denote-expr*$_d$ must be of the same type. The condm form examines the value of each of the *denote-expr*$_t$'s in turn. The first one whose value is 1b is selected and the value of corresponding *denote-expr*$_c$ is used as the value of the storage location denoted by the entire expression. If no *denote-expr*$_t$ has a value of 1b then the value of *denote-expr*$_d$ is used as the value of the denoted storage location.

2.2 Modules

To facilitate modular programming, a program can use an already compiled program as a component. Such submachines are called *modules*. The input and output of modules are defined with the in function and the out function. Each of these functions should appear only once in the program for the module. A module is included in a second Rex program with the defmodule form. The module is then called using the name of the module as though it were a function. Note that each time the module is used a new instance of its machine is created.

(in *type-expr*) Defines the structure of the input of a module to be *type-expr*.

(out *denote-expr*) Defines the output of a module to be *denote-expr*.

(defmodule *module-name*) Adds the machine named *module-name* to the current environment as a constraining function that can be called by the program. The name of the constraining function is the symbol *module-name*. Such functions expect two arguments. The first is the input whose type is declared in the in form within the module definition. The second is the output whose type is declared in the out form. *Module-name* can be either a string or an atom.

The program below uses a submodule that adds a list of four floats to create a class of machines that take a list of four-tuples of floats and returns a list of the sum of each of the four-tuples. The submodule is defined and Rex-compiled with the following forms:

```
(defun sum-listm (list)
  (if (null list)
      ()
      (plusm (car list) (sum-listm (cdr list)))))

(defun sum-listm* (length type)
  (out (sum-listm (in (list-type length type)))))

(makem (sum-list* 4 float) :module "plus-4-float" :obj "" :opt t)
```

The Rex compilation creates the files `plus-4-float.mod` and `plus-4-float.robj` in the current directory. The file with the `.mod` suffix is the *module-declaration*, and contains a description of the inputs required and outputs provided by a `sum-list` machine that works on lists of four `float` storage locations. Note that the above functions define a class of machine that sum the elements of any list of numbers. A machine that sums a list of four floats is selected by the arguments given within the function `makem`.

This machine can now be used to create a four-tuple adder:

```
(defmodule plus-4-float)

(defun sum-four-tuples (list)
  (some* ((output float))
    (cond ((equal (length list) 1)
           (plus-4-float (car list) output)
           output)
          (t (plus-4-float (car list) output)
             (cons output (sum-four-tuples (cdr list)))))))

(defun sum-four-tuples* (number)
  (out (sum-four-tuples (in (list-type number (list-type 4 float))))))
```

The submodule defined previously is used as the constraining function `plus-4-float`. The types of its two arguments are determined by the types of the `in` and `out` forms when the module was compiled. Note that the same module can be used more than once in a program as demonstrated by the recursive call to `sum-four-tuples`. Each time the machine from a module is used, it represents a different copy of that machine with separate state. The function `sum-four-tuples` can now be Rex-compiled to create a machine that accepts a particular length of list of four-tuples. The following form creates one that accepts a list of three four-tuples and, hence, returns a list of three floats:

```
(makem (sum-four-tuples* 3) :module "sum-3-four-tuples" :obj "" :opt t)
```

2.3 Compiling and Running Rex Machines

Currently only one working system implements the described version of Rex. The Rex compiler is written in COMMON LISP and has been used successfully under four different implementations of COMMON LISP. The circuit simulation programs can only be run on Sun Workstations that support the MC68020 and MC68881 floating point coprocessor instruction set.

2.3.1 The Compiler

Rex compilation is done with the special COMMON LISP function `makem`. The syntax for `makem` is

(`makem` *constr-expr* `:module` *module-name* { `:list` *dest-directory* } { `:obj` *dest-directory* } { `:main` *bool* } { `:name` *bool* } { `:no-verify` *bool* } { `:opt` *bool* } { `:picky` *bool* } { `:silent` *bool* })

The argument *constr-expr* is a top level constraining expression. The keywords surrounded with {}'s are optional. The keywords have the following effects:

:module The argument, *module-name*, is a string or atom that is used as the prefix when making the interface and object file. The default is `rex-mod`.

:list The argument, *dest-directory*, is a string that is used as the path to the directory where the user wants a file containing the MC68020 assembly code listing for the machine to be written. If this keyword is omitted, then no assembly listing is generated.

:obj The argument, *dest-directory*, is a string that is used as the path to the directory where the user wants the object code to be written. If this keyword is omitted, then just the interface file and no object code file is created.

:name If the argument is non-`nil`, then each node in the intermediate LISP representation of the Rex machine is given a descriptive name in its variable identifier (`var-id`) field. This option can be used for debugging purposes, but requires some knowledge of the intermediate LISP representation of Rex machines. The default is `nil`.

:no-verify If the argument is non-`nil`, then the compiler does not check the module interface of the current program or of any submodules. Since the module interface is not updated, the circuit depth and parallelism calculations of programs that use this module will be wrong. This option allows the compiler to run faster, but is somewhat risky since incompatibilities between modules can cause the linker to fail or incorrect programs to be generated. In general, this option should only be used if the user is certain the input or output structure of the new module will not change. The default is `nil`.

:opt If the argument is non-`nil`, then the compiler generates optimized code. This process increases compile time somewhat, but in general greatly improves the speed of the machine simulation. One case for which you should turn this optimizer off is for functions that include many module calls and little actual computation. The default is `nil`.

:picky If the argument is non-`nil`, then whenever the compiler finds that it needs to coerce an `int` to a `float`, it executes a `cerror` instruction. This allows the user to make all such coercions explicit. The default is `nil`.

:silent If the argument is non-`nil` then the compiler only prints out the names of any files that it writes and mentions if the module-declaration was changed.

Otherwise, the compiler prints out various useful statistics on the program it is compiling and asks the user if it should update the module-declaration when appropriate. The statistics include the number of times each primitive Rex machine is used; the depth of the circuit (i.e., the greatest number of calculations between an input and an output dependent on that input); and an estimation of the degree of parallelism of the described circuit. The default for this keyword is `nil`.

To create the machine code for the `sum-four-tuples*` Rex machine, evaluate the form:

```
(makem (sum-four-tuples* 3) :module "three-four-tuples"
                            :obj "a-sun:/usr/foo/rex/"
                            :main t
       :opt t)
```

This will create a file named `three-four-tuples.robj` in the directory `/usr/foo/rex` on the Sun, `a-sun`.

2.3.2 The Linker

Once the object code file has been generated, it must be linked. Linking can be done on a Sun with the UNIX shell command `rex_lnk`. The syntax for this command is

`rex_lnk` *module-name*

If the argument *module-name* has a suffix (i.e. is of the form *.*), then `rex_lnk` looks for a file with that name. If no file is found or the argument has no suffix, then `rex_lnk` looks for a file with the name *module-name*`.robj`. If this module has any submodules then the robj-files for these modules must be present in the same directory as the module being linked. If an appropriate file is found along with all the needed `.robj` files for any modules that are used, `rex_lnk` generates two new files: the executable binary file *module-prefix*`.rbin` and the debugging-information file *module-prefix*`.rmap`.

For our example, once the two files, `plus-4-float.robj` and `three-four-tuples.robj`, are in the same directory, they can be linked with the UNIX command

```
rex_lnk three-four-tuples
```

Yielding the files `three-four-tuples.rbin` and `three-four-tuples.rmap`.

2.3.3 The Executor

Once `rex_lnk` has been run, the resulting machine, *module-prefix*`.rbin`, can be run in an interactive mode using the UNIX shell command `rex_run`. The syntax for `rex_run` is

`rex_run -p` *name* {`-d0tr`} { `-o` *name* }

The flags surrounded by {} are optional. The flags have the following interpretations:

- `-p` *name* Indicates that the program in the rbin-file *name* should be run. This is the only flag that is required.
- `-d` Invokes the Rex run-time debugger described in the next section.
- `-O` Causes the storage locations given to the `out` function in the top-level function for the entire program to be printed out every tick. Since information on the type of these outputs is not always available, the values are printed in both `int` and `float` versions. `bools` are printed as `ints` with the value 1 or 0, and `strs` are printed as `ints` that represent the address of the string in memory.
- `-t` Causes the length in milliseconds of each tick to be printed after each tick is completed.
- `-r` Causes `rex_run` to feed the data in a log-file to the program as input. Currently the only way to generate a log-file is with `rex_ex` using the `-l` flag. (See the description of `rex_ex` below.)
- `-o` *name* Indicates the name of the log-file to be used for the `-r` flag. If this flag is used, then the `-r` flag must be used. If the `-r` flag is specified but the `-o` flag is not used, then the program looks for the file `/home/dr4/flakey/in_record.log`. If this file does not exist, `rex_run` exits with an appropriate error.

The program `rex_ex` is a specialized version of `rex_run` that is useful only for running programs that control Flakey, the SRI mobile robot. It can be used only with programs whose input type is `in_buf` and whose output type is `out_buf`. These two types are defined in the LISP file `robot_header.rex`. The syntax for `rex_ex` is

`rex_ex -p` *name* {`-dtlrnPBSTF`} {`-o` *name*} {`-s` *lang voc*}

The flags surrounded by {} are optional. The flags have the following interpretations:

- `-p` *name* Indicates that the program in the rbin-file *name* should be run. This is the only flag that is required. Note the constraints on the program's input type and output type discussed above.
- `-d` Invokes the Rex run-time debugger described in the next section.
- `-t` Causes the length in milliseconds of each tick to be printed after each tick is completed.
- `-l` Causes `rex_ex` to create a compact record of the inputs the robot gets during an execution of `rex_ex`. These files are called log-files. Currently this is the only way to create log-files.
- `-r` Causes `rex_ex` to feed the data in a log-file to the program as input. This flag should not be used with the `-l` flag.

-o *name* Indicates the name of the log-file to be used for the -r or -l flags. If this flag is used with the -l, flag a log-file is created called *name*.log. If this flag is used with the -r flag, then rex_ex looks for the file *name*.log in the current directory and uses it to run the program. If the -r or the -l flag is specified but the -o flag is not used, then the program looks for or creates the file /home/dr4/flakey/in_record.log. If this file does not exist, rex_run exits with an appropriate error.

-n Causes rex_ex to set up a socket and start an ethernet server to control network communication between Flakey and another machine on the network.

-s *lang voc* Causes the speech recognizer to load the speech grammar *lang*.ldf and the user vocabulary file *voc*.voc. These files should be on a disk of the PC controlling the speech recognizer.

-P -B -S -T -F These flags are all related to controlling the display for programs that use Flakey's vision system.

2.3.4 The Rex Runtime Debugger

The Rex run-time debugger allows the programmer to control the execution of a Rex program and to monitor the values of different storage locations during run time. If the -d flag is used in either rex_run or rex_ex, then just before the program is executed the first time, the program goes into the debugger. This is signified by the prompt

```
Bugm>
```

From this prompt you can perform any of a set of commands described below. The commands fit into three basic categories. One set of commands controls the execution of the program. With these commands you can continue the program until it terminates in the usual way, or step from tick to tick. The second set of commands allows you to print and trace the values of predesignated storage locations. The final set of commands allows you to control the way in which these values are printed. The storage locations that can be printed or traced are specified by the programmer explicitly in the Rex program, using the special Rex function name. The syntax of name is the following:

(name *name denote-expr*) Assigns the name *name* to *denote-expr* for use by the debugger. *Name* can be either a string or an atom. The value is *denote-expr*. If the user tries to name two storage locations identically, name automatically adds as many x's to the end of name as needed to make the name unique. Note that this name function is totally separate from the :name keyword used in makem.

For example, if we replace the top-level function sum-four-tuples* with

```
(defun sum-four-tuples* (number)
  (out (name 'sum-out (sum-four-tuples (in (list-type number (list-type 4 float)))))))
```

and recompile, then the storage location resulting from this call to `sum-four-tuples` can be examined during run time. In addition to the explicitly named storage locations, a special storage location named `tick` always contains the number of the most recently executed tick.

The debugger has the following commands:

Execution Control Commands:

s *number* Steps *number* Rex ticks and returns to the debugger. If *number* is omitted, then a single Rex tick is executed. If any storage locations are being traced, they are printed after each tick.

go Executes the machine until it is interrupted or it runs out of input data. If any storage locations are being traced they are printed after each tick.

Tracing and Printing Commands:

trace Should be followed by a list of the names of storage locations named with the Rex function name. These names are added to the *trace-table.* When the machine is executed, after each tick the values of all storage locations whose names appear in the trace-table are printed out. The storage locations are printed out in the order they appear on the trace-table. In addition to being printed on the screen, the trace output is sent to a file named `trace.log`. Even though name tries to avoid name conflict by adding x's to the end if two storage locations are given the same name, two storage locations can still have the same name if they are in different submodules. This is especially common if a module with named storage locations is used more than once in a program and this results in more than one instance of that module. If there is a name conflict of this type, the different storage locations can be referred to by the name followed by a space and a number between 1 and the number of times the name appears. The number depends on the order in which the storage locations appear in the rmap-file. If there is a name conflict and no number is specified, the first (number 1) storage location is traced.

trace-all Causes all the storage locations you have named and the additional location, `tick`, to be traced.

untrace Should be followed by a list of the names of storage locations already in the trace-table. For each name, the first occurrence of that name is removed from the trace-table.

clear Clears the trace-table.

list Lists the trace-table.

trace-qualifier Expects the name of a single boolean storage location. The values of the storage locations named in the trace-table are only printed when the location contains 1b.

print Should be followed by a list of the names of storage locations named with Rex function name. The current values of these storage locations are immediately printed. If this is done before the program has ever been run, all the storage locations will contain 0.

defined *reg-exp* Lists all the names that can be added to the trace-table that match the regular expression *reg-exp*. If *reg-exp* is omitted, then all the names are printed. If *reg-exp* is of the form **reg-exp*, then all names ending with this second *reg-exp* are listed. The command operates similarly for the form *reg-exp**.

Formating Commands:

enable-fat-numb causes numbers to be printed with greater precision.

disable-fat-numb causes numbers to be printed compactly.

enable-header causes a line containing the names in the trace-table to be printed when the machine is started. The names on the screen have the same spacing as the numbers that are being traced. Initially, headers are enabled.

disable-header causes no header line to be printed. This is useful if the `trace.log` file is going to be analyzed by a program that expects only numbers for input.

In addition to the above commands the Rex run-time debugger has a history mechanism. History in the debugger works similarly to history in the UNIX C-shell. The forms that work are `!!`, `!#`, `!-#`, and `!`*string*. History modification is not permitted. The history length is always 19 and history is preserved over runs in the file `.bugm_history`. The current history can be printed with the debugger command h.

3 Programming Examples and Exercises

3.1 Rex Examples

This section contains the Rex code for the machine specifiers discussed previously in the subsection on Rex utilities.

The following are machines that compute minimum, maximum, and absolute value:

```
(defun minm (x y)
  (ifm (<m x y)
       x
       y))

(defun maxm (x y)
  (ifm (>m x y)
       x
       y))

(defun absm (x)
  (ifm (<m x !0)
       (unary-minusm x)
       x))
```

The following is a selector for structured storage locations. Note that all of the recursion in these definitions happens at compile time.

```
(defun select-recurm (current-index index list)
  (if (= 1 (length list))
      (car list)
      (ifm (>=m !current-index index)
           (car list)
           (select-recurm (1+ current-index) index (cdr list)))))

(defun selectm (index list)
  (select-recurm 0 index list))
```

This machine constructor creates a family of running-sum machines with differing initial values:

```
(defun running-sum (init input)
    (some* ((sum int))
      (== sum (init-next init (plusm sum input)))
      sum))

(defun running-sum* (init)
  (out (running-sum init (in int))))
```

Given a list of length `length` and an item, output a 1 if `item` is equal to one of the elements of the list, and 0 otherwise:

```
(defun memberm (item list)
  (if (null list)
      !'0b
      (orm (equalm item (car list))
           (memberm item (cdr list)))))

(defun memberm* (length type)
  (some* ((the-item type)
          (the-list (list-type length type)))
    (== [the-item the-list] (in [type (list-type length type)]))
    (out (memberm the-item the-list))))
```

This function maps the `plusm` machine over two lists of length `length`, generating a list whose components contain the sums of the corresponding components in the input lists:

```
(defun plus-list (a b)
  (if (null a)
      []
      (cons (plusm (car a) (car b))
            (plus-list (cdr a) (cdr b)))))

(defun plus-list* (length)
  (some* ((list1 (list-type length int))
          (list2 (list-type length int)))
    (== [list1 list2] (in [(list-type length int) (list-type length int)]))
    (out (plus-list list1 list2))))
```

All of the examples given so far are written in a functional style. The following program is an example of a more relational style of programming. It implements a push-down stack.

```
(defvar *pop* )
(defvar *push* )
(defvar *noop* )
(defvar *undefined* -99999)

(defun stack-element (operation below element above)
  (== element (init-next *undefined*
                         (ifm (equalm operation !*pop*)
                              below
                              (ifm (equalm operation !*push*)
                                   above
                                   element)))))

(defun stack (operation push-value the-stack)
  (if (= (length the-stack) 1)
```

```
      (stack-element operation !*undefined* (car the-stack) push-value)
      (some* ()
        (stack-element operation (second the-stack)
                       (car the-stack) push-value)
        (stack operation (car the-stack) (cdr the-stack))))
  (ifm (equalm operation !*pop*)
       (car the-stack)
       !0))

(defun stack* (size)
  (some*
    ((the-operation int)
     (the-push-val int)
     (the-stack (list-type size int)))
    (== [the-operation the-push-val] (in [int int]))
    (out (stack the-operation the-push-val the-stack))))
```

This stack is structured as a one-dimensional cellular automaton. The contents of the storage location `operation` control what is to happen to the stack at each cycle. It can contain one of the following values (the encoding is unimportant): `*push*`, `*pop*`, and `*noop*`. If the value of `operation` is `*push*`, the value of any given stack element is the previous value of the element above it and the top element takes on the value of the storage location `push-value`; if the value of `operation` is `*pop*`, the value of any stack element is the previous value of the element below it and the top value is returned by the function `stack`; if the value is `*noop*`, each stack element retains its previous value. The relation `stack-element` has four denoting-expression parameters: `operation` contains the stack operation to be performed, `above` is the cell above, `element` is the stack element whose value we are computing, and `below` is the stack element below. The definition encodes the rule given above for the behavior of stack elements.

The stack is created recursively. The size of the stack is determined by the length of the list passed to the function `stack`. If there is only one element in the stack, then the `below` element is set to a special bottom element that always contains the constant `*undefined*` since there is no real element below it. Thus, as values are popped off the stack, the stack locations are filled in with that value. This allows us to check to see if the stack is full — if it is, the last element is different from `*undefined*`. If more than one element is in the stack, we constrain the top of the stack to be a stack element with `above` bound to the `push-value`, and `below` to be the second element of the stack. Then we constrain the rest of `stack` to behave as a (`length` − 1)-sized stack with the push-value set to the old top of the stack.

3.2 Sample Problems

Following is a list of problems to be solved by writing Rex machine constructors. These problems illustrate the variety of programming techniques available in Rex. The solutions follow in the next subsection.

Write Rex constructors for the following machines:

1. The **sometime** machine. It takes a single boolean (1b or 0b) input, and outputs 1b if the input has ever been 1b, 0b otherwise.

2. The **always** machine. It takes a single boolean input and outputs 1b if the input has always been 1b, 0b otherwise.

3. The **running-average** machine. It takes one integer input. The output should be the average of all the inputs so far.

4. The **plus-struct** machine. It takes two structured storage arguments. It should output a storage structure with the same structure as the inputs, but with components containing the sums of the corresponding components in the input structures. A call to this machine constructor might look like

```
(plus-structm [a1 [a2 a3] [a4 a5]] [b1 [b2 b3] [b4 b5]])
```

5. The **life** machine. Write a Rex program that simulates Conway's Game of Life. The simulation takes place on a rectangular array of cells, each of which may contain an organism. Except for borders, each cell has eight cells immediately adjacent to it, and so each organism may have up to eight neighbors. The survival and reproduction of an organism from generation to generation depends on the number of neighbors it has and is determined by four simple rules:

   (a) If an organism has no neighbors or only one neighbor, it dies of loneliness.

   (b) If an organism has two or three neighbors, it survives to the next generation.

   (c) If an organism has four or more neighbors, it dies of overcrowding.

   (d) An organism is born in any empty cell that has exactly three neighbors.

   All changes occur simultaneously; the fate of an organism depends on the current generation irrespective of what may happen to its neighbors in the next generation.

   The Rex machine constructor should have one value parameter, n (the Life game takes place on an n x n grid), and no storage arguments. It should output the rectangular grid containing the Life simulation. For convenience in processing the borders, you may make the grid $n+2$ by $n+2$, with an outer border initialized to always contain 0's.

3.3 Solutions

Following are possible solutions to the problems posed in the previous section.

1. This specifies a machine that returns 1b if the input has *ever* been 1b, and 0b otherwise:

```
(defun sometime (input)
  (some* ((result bool)
          (output bool))
    (== output (orm input result))
    (== result (init-next '0b output))
    output))

(defun sometime* ()
  (out (sometime (in bool))))
```

2. This specifies a machine that returns 1b if the input has *always* been 1b, and 0b otherwise:

```
(defun always (input)
  (some* ((result bool)
          (output bool))
    (== output (andm input result))
    (== result (init-next '1b output))
    output))

(defun always* ()
  (out (always (in bool))))
```

3. The following returns the running average of the machines inputs. We keep a counter of time since the machine began and a running sum. The returned result is the quotient of the sum and the count.

```
(defun running-average (in-type input)
  (some* ((count in-type)
          (count-out in-type)
          (sum in-type)
          (sum-out in-type))
    (== count-out (plusm count !1.0))
    (== sum-out (plusm sum input))
    (== count (init-next 0.0 count-out))
    (== sum (init-next 0.0 sum-out))
    (divm sum-out count-out)))

(defun running-average* (in-type)
  (out (running-average in-type (in in-type))))
```

4. Here we sum two storage locations of equivalent structure, returning a storage location whose elements are the sums of the corresponding elements in the input structures. This is done by general structure recursion on the structure of the inputs.

```
(defun plus-struct (a b)
  (if (null a)
      []
```

```
      (if (var-p a)
          (plusm a b)
          (cons (plus-struct (car a) (car b))
                (plus-struct (cdr a) (cdr b))))))

(defun plus-struct* (type)
  (some* ((a type)
          (b type)
          (output type))
    (== [a b] (in [type type]))
    (== output (plus-struct a b))
    (name 'result output)
    (out output)))
```

5. This specifies a cellular automaton that simulates Conway's Game of Life (see problems for the description) on an n by n grid, with empty borders. The compile time value parameter `init-grid` must be an n-element list of n-element lists of either `0b` or `1b`. That parameter is the initial value of the Life array.

   The function `grid-type` specifies a `size` by `size` grid of booleans. This method of duplicating the type of `int-grid` ensures that it is a square grid of booleans.

```
(defun grid-type (size)
  (list-type size (list-type size bool)))
```

   This is the top-level function that allocates and initializes the boolean grid of storage locations that are the cells of the game:

```
(defun life* (init-grid)
  (some* ((a-grid (grid-type (length init-grid))))
    (== a-grid (init-next init-grid (hook-up-grid 0 a-grid)))
    (out a-grid)))
```

   The denoting function, `hook-up-grid`, recursively constrains each row in the grid. The value parameter, `row`, indicates the row being constrained. The denoting-expression parameter, `grid`, is used for indicating the size of the grid and is passed on to other denoting functions that denote and constrain the rows of the grid denoted by this function. The first and last rows are denoted by the function `hook-up-zero-row`, which constrains these rows to always contain `0b`. The other rows are computed with the function `hook-up-row`:

```
(defun hook-up-grid (row grid)
  (cond ((= row 0)
         (cons (hook-up-zero-row (length grid)) (hook-up-grid (1+ row) grid)))
        ((= (+ row 1) (length grid))
         [(hook-up-zero-row (length grid))])
        (t (cons (hook-up-row row 0 grid) (hook-up-grid (1+ row) grid)))))
```

Below we generate one row of the Life grid as a function of the whole grid (really only the row above and below, but it's easier to pass the whole thing around). This is done recursively by the column. It takes two value parameters: which row this is, and what column we're working on. If this is the first or last column, it is constrained to 0b; otherwise, it is the result of the denoting function `cellm`. The first function, `element`, denotes the cell of `grid` indexed by `row` and `col`. This function is used by the second function, `hook-up-row`, to create the list of neighbors of each cell passed to the function `cellm`.

```
(defun element (row col grid)
  (nth col (nth row grid)))

(defun hook-up-row (row col grid)
  (cond ((= col 0)
         (cons !'0b (hook-up-row row (1+ col) grid)))
        ((= (+ col 1) (length grid))
         [!'0b])
        (t (cons (cellm (element row col grid)
                        [(element (1- row) (1- col) grid)
                         (element (1- row) col grid)
                         (element (1- row) (1+ col) grid)
                         (element row (1- col) grid)
                         (element row (1+ col) grid)
                         (element (1+ row) (1- col) grid)
                         (element (1+ row) col grid)
                         (element (1+ row) (1+ col) grid)])
                 (hook-up-row row (1+ col) grid)))))
```

The following function, `bool-count`, counts the number of 1b's in a list of boolean storage locations. This function is used by `cellm` to calculate the value of the cell using the value of the corresponding cell denoted by the parameter `cell` and those of the neighboring cells denoted by the parameter `neighbors`.

```
(defun bool-count (l)
  (cond ((null l)
         !0)
        (t (ifm (car l)
                (plusm !1 (bool-count (cdr l)))
                (bool-count (cdr l))))))

(defun cellm (cell neighbors)
  (some* ((neighbor-count int))
    (== neighbor-count (bool-count neighbors))
    (ifm (orm (<=m neighbor-count !1)
              (>=m neighbor-count !4))
         !'0b
         (ifm (equalm neighbor-count !3)
              !'1b
              cell))))
```

This function denotes a storage structure that is a list of length `length`, each of whose elements has the value 0b:

```
(defun hook-up-zero-row (length)
  (if (zerop length)
      []
      (cons !'0b (hook-up-zero-row (- length 1)))))
```

Rex compilation would be done with a form like the following:

```
(makem (life* '((0b 0b 0b 0b 0b 0b)
                (0b 0b 0b 0b 0b 0b)
                (0b 0b 1b 0b 0b 0b)
                (0b 0b 1b 0b 0b 0b)
                (0b 0b 1b 0b 0b 0b)
                (0b 0b 0b 0b 0b 0b))))
```

The pattern here should oscillate between a vertical and a horizontal row of three 1b's.

Acknowledgments

The Rex language was iteratively designed and implemented by Stanley Rosenschein and Leslie Kaelbling. Nathan Wilson implemented the optimizing code generator, and Stanley Reifel provided the UNIX linking, execution, and debugging tools. David Chapman provided useful insights and much help on previous implementations of Rex. Stan Rosenschein, Stan Reifel, Sandy Wells, and David Kaelbling made helpful comments on earlier drafts of this paper.

References

[1] Hillis, W. Daniel, 1985: *The Connection Machine* (The MIT Press, Cambridge, Massachusetts).

[2] Johnson, Steven D., 1983: *Synthesis of Digital Designs from Recursion Equations* (The MIT Press, Cambridge, Massachusetts).

[3] Rosenschein, Stanley J., 1986: personal communication.

[4] Rosenschein, Stanley J. and Leslie Pack Kaelbling, 1986: "The Synthesis of Digital Machines with Provable Epistemic Properties," *Proceedings of the Conference on Theoretical Aspects of Reasoning About Knowledge*, pp. 83-98.

Appendix: Errors Encountered During Compiling

This appendix describes the error messages that can be generated by the Rex compiler. The majority error messages that can be generated by the Rex compiler are simple syntax errors in the use of a primitive Rex machine or a special Rex syntax. These messages include the name of the function with the error and a brief description of the problem. There is also a set of fatal compiler errors. These errors are all signalled with the string `Compiler Error!!!!`. Other error messages that can be generated are described below:

Address *number* is too large. This error indicates that the number of vars has exceeded 8,192. This causes an error because the 68xxx family of chips cannot easily address locations more than 32k off of an address register. When this error occurs, find some way to break the program down into modules.

Value *value* is not of type *type*. This error indicates that an attempt was made to use *value* incorrectly as a *type*. Look in your code for constants equal to *value*. If none are found, the compiler may be attempting to create a new constant from other constants and encountering an error. Look up the stack to find the other constants.

***Type* is not a valid data type.** *Type* is undeclared or incorrectly declared.

Attempt to equate nonconformable structures. The storage locations in a structure equation do not have the same structure.

Attempt to equate vars of different types. A pair of corresponding atomic storage locations in a structure equation have different types.

Attempt to equate two already defined vars. A pair of corresponding atomic storage locations in a structure equation are both constrained elsewhere in the code.

The module, *mod*, has not been properly declared. Try recompiling the module *mod* and evaluating the form `(defmodule` *mod*`)`. If the problem persists with the same module, remove the file *mod*.mod and evaluate the form `(clrhash nrex::*module-hashtable*)`, then recompile the module and evaluate the form `(defmodule` *mod*`)`. If the problem still persists, it may be necessary to reload the entire LISP world.

***Input/output* type does match module declaration for *mod*.** A module has been called with the wrong type of arguments. There is an error in either (1) the use of the module or (2) the `in` or `out` forms of the module.

Propagation or module var depends on itself with no delay. An illegal circularity has been found in the circuit. You probably forgot an `init-next`.

***Thing* is not a structure of vars.** The input, *thing*, to a Rex machine is not a storage location. Most likely there is a missing '!'.

The node, *var*, has no definition. Look up the stack to see who depends on it. Some output, named variable, or descendent of such is not defined. Looking

up the stack tells you what kind of thing is undefined and hopefully gives clues as to where in the code the problem is. One common cause of this error is when the computation of *var* has been removed from the code, but *var* is still named.

The output, *var*, is undefined. One of the outputs, as declared in the out form, is not constrained.