MIT Reversible Computing Project

Memo #M8

The R Programming Language and Compiler

WORKING DRAFT MEMO

Revision: 1.3

Michael P. Frank
http://www.ai.mit.edu/~mpf

MIT AI Lab
545 Technology Sq.
Cambridge, MA 02139

Started July 28, 1997.
Revision Date: 1997/07/30 19:08:57 GMT
Typeset July 30, 1997.

A current version is available online at
http://www.ai.mit.edu/~mpf/rc/memos/M08/M08_rdoc.html

1 Introduction

R is a programming language for reversible machines. The language is currently very incomplete and is developing rapidly. This memo documents the current state of the language, to convey a feel for the language as it stands, and solicit feedback regarding how the language should develop.

The R language compiler translates R source into Pendulum assembly code. In this document we will also describe the workings of the R compiler. Currently the compiler works by applying code transformations similar to macro expansions, to reduce high-level constructs into successively lower level constructs until the expansion bottoms out with Pendulum assembly language instructions.

Because of the many levels of constructs involved in this gradual transformation process, the distinction between the constructs intended for end-use in R source programs and the intermediate constructs used internally by the compiler is currently rather fuzzy. This document will attempt to separate user-level from compiler-level constructs, but the status of constructs may change as the language evolves, and currently there is nothing to prevent an R source program from using constructs at all the different levels. However, the lower-level constructs are perhaps somewhat more likely to change as the language and compiler evolve, so their use in application programs is discouraged.

2 What type of language is R?

R is like C in that it is (currently) a procedural language, not a strict functional language, with data types and primitive operations centered around the two's complement fixed-precision integers and the corresponding arithmetic/logical operations that are supported directly by the machine hardware. The language supports simple C-like arrays, `for` loops, `if` statements, and recursive subroutines with arguments.

Reversibility of execution of R programs is guaranteed by the reversibility of the assumed version of the Pendulum instruction set, so long the program does not use the `EMIT` assembly-language instruction (which explicitly permits information to be removed irretrievably from the processor). However, if the user wishes his programs to run not only reversibly but correctly, he is responsible for ensuring that certain conditions are met by his code. Currently, these conditions are not checked automatically. If the conditions are not met, then the program will silently proceed anyway, with nonsensical (but still reversible) behavior. However, this is not as fatal as it sounds, because the reversibility of execution allows the errant program to be debugged, after the misbehavior is discovered, by running it in reverse from the error to see what caused it.

3 Overview of R Syntax

R programs are currently represented using nested, parenthesized lists of symbols and numbers, as in Lisp. Similarly to Lisp, the first element of a list may be a symbol that identifes the kind of construct that the list is representing, for example, a function definition, an `if` statement, a `let` construct for variable binding. However, in R, currently some constructs may also be denoted using *infix* notation, in which the identifying symbol is the *second* element of the list instead of the first. Many of these infix lists have a C-like syntax and behavior, for example, the `(a += b)` statement which adds b into a. Infix notation is also often used in subexpressions of a statement which are intended to evaluate to a value, for example, `(a + b)` in the statement `(print (a + b))`.

4 User-level Constructs

This section describes constructs that are intended for use in end-user R applications.

4.1 Program Structure

The executable portion of a program normally consists of a single `defmain` statement, and any number of `defsub` statements. These statements may appear in any order.

`defmain`

Define program's main routine.

Syntax: (defmain *progname*
statement$_1$
statement$_2$ ...)

Elements:

progname — A symbol naming the program. The name should be a sequence of letters and digits starting with a letter. It should be distinct from the names of all subroutines and static data items in the program.

statement$_1$, *statement*$_2$, ... — Statements to be executed in sequence as the main routine of the program.

Description:

The `defmain` statement is used to define the main routine of a program. It is intended to appear as a top-level form, but may actually appear anywhere a statement may appear. (If executed as a statement, it does nothing.) Currently there is no "command line" or other argument list available to the program; it must either be self-contained along with its data or explicitly read data from an input stream. `Defmain` generates information in the output file that tells the run-time environment where to begin executing. If there are zero or more than one `defmain` statements in a given program, then the result of attempting to run the program is undefined.

`Defmain` currently also has the side effect of causing the entire standard library to be included in the output program. Right now there is only one subroutine in the standard library (named `smf`), so this is not too burdensome.

`defsub` — Define subroutine.

Syntax: (`defsub` *subname* (arg_1 arg_2 ...)
statement$_1$
statement$_2$ ...)

Elements:

subname — A symbol naming the subroutine. The name should be a sequence of letters and digits starting with a letter. It should be distinct from the names of the main routine, all other subroutines, and all static data items in the program.

arg_1, arg_2, ... — Formal argument names, with the same alphanumeric format. These names are not required to be distinct from any other names in the program. However a single subroutine cannot have two arguments with the same name. Currently, the compiler does not support subroutines taking more than 29 arguments on the Pendulum architecture.

statement$_1$, *statement*$_2$, ... — Statements to be executed in sequence as the body of the subroutine.

Description:

`Defsub` statements are used to define subroutines within a program. They are intended to appear only as top-level forms, but may actually appear anywhere that a statement may appear. (If executed as a statement, a `defsub` construct does nothing.) If there are two `defsub` statements with the same *subname* in a given program, then the result of attempting to execute that program is undefined.

The formal arguments may be accessed as read-write variables within the body of the subroutine. On entry to the subroutine, the values of these variables are bound to the values of the actual arguments that were passed in via the `call` statement in the caller. The `call` statement must pass exactly the number of arguments required by the subroutine or else the behavior of the subroutine is undefined. On exit, the values of the argument variables become the new values of the actual arguments (see the description of `call`).

Any subroutine may also be called in reverse; see `rcall`.

4.2 Control Structure

Within the program's main routine and subroutines, the flow of execution is controlled using `call`, `rcall`, `if`, and `for` statements.

`call`, `rcall`

Call or reverse-call subroutine.

Syntax: `(call` *subname* arg_1 arg_2 ...`)` *or*
`(rcall` *subname* arg_1 arg_2 ...`)`

Elements:

subname — The name of the subroutine to call. If zero or more than one subroutines with that name exist in the program, the result of the call is undefined.

arg_1, arg_2, ... — Actual arguments to the subroutine. These may be variables, constants, or expressions, with restrictions described below. The number of arguments must match the number of formal arguments listed in the subroutine's `defsub` statement.

Description:

A `call` or `rcall` statement is used to call a subroutine either forwards or in reverse, with arguments. If a particular actual argument is a variable or a memory reference, then the subroutine may actually change the value of its corresponding formal argument, and the caller will see the new value after the call is completed. If the argument is a constant or an expression, then it is an error for the subroutine to return with the corresponding formal argument having a value that is different from the value that the constant or expression evaluates to after the return. (Nonsensical behavior will result.)

`Rcall` differs from `call` only in that with `rcall`, the subroutine body is executed in the reverse direction from the direction in which the `rcall` is executed.

`if`

Conditional execution.

Syntax:
```
(if condition then
    statement1
    statement2 ...)
```

Elements:

condition — An expression representing a condition; considered "true" if its value is non-zero.

$statement_1$, $statement_2$, ... — Statements to execute if the condition is true.

Description:

An `if` statement conditionally executes the body *statements* if the *condition* expression evaluates to a non-zero value. If the value of the *condition* expression ever has a different value at the end of the body from the value it had at the beginning, then program behavior after that point will in general be nonsensical.

The top-level operation in the *condition* expression may be a normal expression operation, or one of the relational operators `=`, `<`, `>`, `<=`, `>=`, `!=` which have the expected C-like meanings of signed integer comparison. These relational operators are not currently supported for use in expressions in contexts other than the top-level expressions in `if` *conditions*.

Actually the compiler does not yet moment support all the different relations with all of the possible types of arguments even in `if` *conditions*. The `if` implementation in the compiler needs some major rewriting.

In the future, `if` statements will also be allowed to appear in forms containing `else` clauses, using the syntax

(if *condition*
    $if\text{-}statement_1$ $if\text{-}statement_2$ ...
 else
    $else\text{-}statement_1$ $else\text{-}statement_2$ ...),

but currently this form of `if` is not yet implemented by the compiler.

`for`

For loop; definite iteration.

Syntax: (for *var* = *start* to *end*
    $statement_1$
    $statement_2$ ...)

Elements:

var — A variable name.

start — Start value expression.

end — End value expression.

$statement_1$, $statement_2$, ... — Statements to execute on each iteration.

Description:

A `for` statement performs definite iteration. *Var* must not exist as a variable at the point where the `for` construct appears, but it may exist as a name of a static data element, in which case this meaning will be shadowed during the `for`.

Before the loop, the *start* and *end* expressions are evaluated, and *var* is bound to the value of *start*. The scope of *var* is the body of the `for`. On each iteration, the body *statement*s are executed. After each iteration, if *var* is equal to the value of *end* which was computed earlier, the loop terminates; otherwise, *var* is incremented as a mod-2^{32} integer and the loop continues. After the loop, the *start* and *end* expressions are evaluated again in reverse to uncompute their stored values.

It is an error for either the *start* or *end* expressions to evaluate to different values after the loop than they do before the loop. If they do, program behavior afterwards will be nonsensical. The same goes for any of their subexpressions.

Note that although `for` is intended for definite iteration, in which the number of iterations is always exactly the difference between the initially-computed *start* and *end* values, actually there is nothing to prevent the value of *var* from being modified within the body, so that the number of iterations can actually be determined dynamically as the iteration proceeds. One can thus construct "while"-like indefinite iteration functionality using `for` as a primitive. However, this is inconvenient, so the language will eventually explicitly include a `while`-like construct, though it does not do so currently.

4.3 Variables

New local variables may be created and bound to values anywhere a statement may appear, using the `let` statement.

`let`

New variable binding.

Syntax: `(let (`*var* `<-` *val*`)`
`statement`$_1$
`statement`$_2$ `...)`

Elements:

var — A new variable name. This name must not exist as a local variable name at the point where the `let` statement occurs. However, it may exist as the name of a static data item, in which case that meaning will be shadowed within the body of the `let`.

val — An expression to whose value *var* will be bound.

statement$_1$, *statement*$_2$, ... — Statements to execute in the scope where *var* is available as a variable.

Description:

`Let` creates a new local variable *var* and binds it to a value. The body of the `let` may change the value of *var*, but the value of *var* at the end of the body *must* match the value that the *val* expression has at the time the body ends. Otherwise program behavior will be unpredictable thereafter. The *val* expression is actually evaluated twice, once forwards before the body, to generate the value to bind to *var*, and once backwards after the body, to uncompute this value.

Actually the current implementation of `let` does require the value of *val* and all its subexpressions to remain the same at both the start and end of the body. Future implementations may relax this restriction.

Other forms of the `let` construct currently exist, but are not currently documented as user-level constructs.

4.4 Data Modification

Currently, R programs modify variables and memory locations using a variety of vaguely C-like data modification constructs: `++`, `-`, `<=<`, `>=>`, `+=`, `-=`, `^=`, `<->`, and others not currently documented as user-level operations.

In general, it is an error for a data modification statement to modify a variable or memory location whose value is used in any subexpressions of the statement. If this happens, program behavior thereafter will be nonsensical.

`++` — Integer increment statement.

Syntax: `(`*place* `++)`

Elements:

place — A variable or an expression denoting a memory reference.

Description:

The mod-2^{32} integer word stored in *place*, which may be a variable or a memory reference, is incremented by 1.

`-` (minus sign) — Unary negate statement.

Syntax: `(- `*place*`)`

Elements:

place — A variable or an expression denoting a memory reference.

Description:

The integer word stored in *place* is negated in two's complement fashion.

`<=<`, `>=>`

Rotate left/right.

Syntax: (*place* `<=<` *amount*)
(*place* `>=>` *amount*)

Elements:

place — A variable or an expression denoting a memory reference.

amount — An expression for the amount to rotate by.

Description:

`<=<` rotates the bits stored in the given *place* to the left by the given *amount*. Rotating left by 1 means the bit stored in most significant bit-location moves to the least significant bit-location, and all the other bits shift over to the next, more significant position. Rotating by some other amount produces the same result as rotating by 1 *amount* times. `>=>` is the same but rotates to the right (exactly undoing `<=<`).

`+=`, `-=`

Add/subtract statement.

Syntax: (*place* `+=` *value*)
(*place* `-=` *value*)

Elements:

place — A variable, or an expression denoting a memory reference.

value — An expression for the value to add/subtract.

Description:

`+=` adds *value* into *place*, as an integer. `-=` subtracts *value* from *place*.

`^=`

Exclusive OR.

Syntax: (*place* `^=` *value*)

Elements:

place — A variable, or an expression denoting a memory reference.

value — An expression for the value to XOR.

Description:

`^=` bitwise exclusive-OR's *value* into *place*.

`<->` — Swap.

Syntax: `(`$place_1$ `<->` $place_2$`)`

Elements:

place$_1$, *place*$_2$ — Each is a variable or an expression denoting a memory reference.

Description:

`<->` swaps the contents of the two *place*s.

4.5 Expressions

Variables and constants count as expression, as do the more complex parenthesized expressions described here. Expressions may be nested arbitrarily deeply. Parentheses for all the subexpressions must all be explicitly present. Currently, all expression operations are of the infix style, where the symbol for the operator appears as the second member of the list; however new kinds of expressions may exist later.

Currently available expression operations include `+`, `-`, `&`, `<<`, `>>`, `*`, `*/`, `_`, and others for internal use by the compiler.

There are also relational operators `=`, `<`, `>`, `<=`, `>=`, `!=` which may currently only be used at top-level expressions in `if` *condition*s. They are not yet documented individually yet, but they have the expected behavior of signed integer comparison. Conceptually they return 1 if the relation holds, and 0 otherwise.

Inside expressions, only expression constructs may be used. Expression constructs may never be used in place of statements.

Expressions are generally evaluated twice each time they are used, once in the forward direction to generate the result, and once in the reverse direction to uncompute it.

There is currently no way within the language to define a new type of expression operator, but this may change later.

`+`, `-` — Sum/difference expression.

Syntax: `(`val_1 `+` val_2`)`
`(`val_1 `-` val_2`)`

Elements:

val$_1$, *val*$_2$ — Expressions for values to add.

Description:

Evaluates to the sum or difference of the values of the two sub-expressions taken as mod-2^{32} integers.

&

Bitwise logical AND expression.

Syntax: `(`val_1 `&` val_2`)`

Elements:

val_1, val_2 — Expressions for values to AND.

Description:

Evaluates to the bitwise logical AND of the word values of the two sub-expressions.

<<, >>

Logical left/right shift expression.

Syntax: `(`*val* `<<` *amt*`)`
`(`*val* `>>` *amt*`)`

Elements:

val — Expression for the value to be shifted.

amt — Expression for the amount to shift by.

Description:

This evaluates to the value of *val* logically shifted left or right as a 32-bit word, by *amt* bit-positions.

---

*

Pointer dereference expression.

Syntax: `(*` *address*`)`

Elements:

address — Expression that evaluates to a memory address.

Description:

This evaluates to a copy of the contents of the memory location at the given *address*. However, this expression may also be used as a *place* which may be modified by any of the data-modification statements above, in which case the actual contents of the location, not a copy, is modified.

It is an error for the contents of an address to be referred to by a subexpression of a statement that modifies that same address; if this is done, behavior thenceforth will be unpredictable.

`*/` — Fractional product expression.

Syntax: (*integer* `*/` *fraction*)

Elements:

integer — An expression whose value is taken as a signed integer.

fraction — An expression whose value is taken as fraction between -1 and 1.

Description:

This rather odd operator returns the signed 32-bit integer product of the two values, taking one as a signed 32-bit integer and the other as a signed 32-bit fixed-precision fraction between 0 and 1. Another way of saying this is that it is the product of two integers, divided by 2^{32}. Or, it is the upper word of the 64-bit product of the two integers, rather than the lower word.

This operation is useful for doing fixed-precision fractional arithmetic. It is used by the single existing significant test program `sch.r`.

Since the Pendulum architecture naturally does not support this rather unusual operation directly, the compiler transforms it into a call to the standard library routine `SMF` (Signed Multiplication by Fraction). `SMF` is itself written in R, but for efficiency it uses some optimized constructs not yet intended for general use.

`_` (underscore) — Array dereference expression.

Syntax: (*array* `_` *index*)

Elements:

array — Expression for the address of element 0 of an array in memory.

index — Expression for the index of the array element to access.

Description:

This type of expression evaluates to a copy of the contents of the element numbered *index* in the sequential array of memory locations whose element number 0 is pointed to by *array*. However, this expression may also be used as a *place* in any of the data-modification statements, in which case it is the real array element that will be modified, not just a copy of it.

It is an error for an array element or other memory location to be examined by an subexpression of a statement that ends up modifying that location.

4.6 Static Data

Two constructs, `defword` and `defarray`, allow single words and regions of memory to be named and initialized to definite values when the program is loaded.

`defword` — Define a global variable.

Syntax: `(defword` *name value*`)`

Elements:

name — An alphanumeric symbol naming this variable. Must be distinct from the names of routines and other static data items.

value — A 32-bit constant integer giving the initial value of the variable.

Description:

`Defword` is intended for use as a top-level form but actually it may appear anywhere a statement may appear. When executed as a statement it does nothing.

`Defword` defines the name *name* to globally refer to a particular unique memory location, whose initial value when the program is loaded is *value*. This meaning of *name* can be shadowed within subroutines that have *name* as a formal argument, or within the body of a `let` statement that binds that name. The *name* can be used as a *place* in data-modification statements.

Actually the *name* will only be recognized to refer to the memory location at statements in the program that occur textually *after* the `defword` declaration.

`defarray` — Define a global array.

Syntax: `(defarray` *name*
value$_0$ *value*$_1$ `... )`

Elements:

name — Unique alphanumeric name for the array.

value$_0$, *value*$_1$, ... — Integer constants giving the initial values of all the array elements.

Description:

`Defarray` is intended for use as a top-level form, but actually it may appear anywhere a statement may appear. When executed as a statement it does nothing.

`Defarray` sets aside a contiguous region of memory, containing a number of words equal to the number of *value* arguments, and defines the name *name* to globally (that is, after the `defarray`) refer to the address of the first word in the region. The words are initialized to the given *value*s when the program is first loaded. *Name* can be used as an *array* in array-dereference operations. It is a compile-time error to attempt to change the value of *name*. However, *name* can be shadowed by subroutine arguments and local variable declarations.

4.7 Input/Output

Currently there are no input constructs in R. However, there are two user-level output constructs, `printword` and `println`. These are rather ad-hoc. The set of I/O constructs is a part of R that is particularly likely to change in later versions of the language.

`printword`

Output a representation of a word of data.

Syntax: `(printword` *val*`)`

Elements:

val — An expression for the word to print.

Description:

`Printword` sends to the output stream a representation of the value of the *val* expression, as a 32-bit integer. Currently the representation consists of outputing the value `0` followed by the given value, to distinguish the output from that produced by `println`.

The *val* expression is evaluated twice, once to compute the value and again to uncompute it.

`println`

Output a representation of a line-break delimiter.

Syntax: `(println)`

Elements:

None.

Description:

`Println` sends to the output stream a representation of a line-break delimiter. Currently this consists of outputing the single word `1`.

5 Example Program `sch.r`

As an example of R programming style and of many of the user-level constructs described above, here is the first significant test program, `sch.r`, in its entirety. The character "`;`" indicates a comment that runs to the end of the line.

This program simulates the quantum-mechanical behavior of an electron oscillating at near the speed of light in a 1-dimensional parabolic potential well about 1 Ångstrom (10^{-10} m) wide. It takes about 1 minute to complete each 5×10^{-22} second long simulation step under the PendVM Pendulum virtual machine emulation program, running on a SPARC 20.

An interesting feature of this program is that although it is perfectly reversible, its outer loop can run for indefinitely long periods, without either slowing down or filling up the memory with garbage data.

The current version of the compiler successfully compiles this program to correct (though not optimally efficient) Pendulum code. When run, the compiled program produces exactly the correct output.

```
;;;-----------------------------------------------------------
;;;
;;;  Schroedinger Wave Equation simulation program.
;;;  The first major test of R (the reversible language)!
;;;
;;;-----------------------------------------------------------

;;; Currently all data must come before the code that uses
;;; it, so that the compiler will recognize these
;;; identifiers as names of static data items rather than as
;;; dynamic variables.

;; epsilon = hbar*dt/m*dx^2.  DX=7.8125e-13m, DT=5e-22s
(defword epsilon 203667001) ; 0.0948398 radians.
;; Parabolic potential well with 128 points.
(defarray alphas
  458243442 456664951 455111319 453582544 452078627
  450599569 449145369 447716027 446311542 444931917
  443577149 442247239 440942188 439661994 438406659
  437176182 435970563 434789802 433633899 432502854
```

```
  431396668 430315339 429258869 428227257 427220503
  426238607 425281569 424349389 423442068 422559605
  421701999 420869252 420061363 419278332 418520159
  417786845 417078388 416394790 415736049 415102167
  414493143 413908977 413349669 412815220 412305628
  411820895 411361019 410926002 410515843 410130542
  409770099 409434515 409123788 408837920 408576909
  408340757 408129463 407943027 407781450 407644730
  407532868 407445865 407383720 407346432 407334003
  407346432 407383720 407445865 407532868 407644730
  407781450 407943027 408129463 408340757 408576909
  408837920 409123788 409434515 409770099 410130542
  410515843 410926002 411361019 411820895 412305628
  412815220 413349669 413908977 414493143 415102167
  415736049 416394790 417078388 417786845 418520159
  419278332 420061363 420869252 421701999 422559605
  423442068 424349389 425281569 426238607 427220503
  428227257 429258869 430315339 431396668 432502854
  433633899 434789802 435970563 437176182 438406659
  439661994 440942188 442247239 443577149 444931917
  446311542 447716027 449145369 450599569 452078627
  453582544 455111319 456664951)

;; This is the shape of the initial wavefunction; amplitude
;; doesn't matter.
;; Real part.
(defarray psiR 2072809 3044772 4418237 6333469 8968770
  12546502 17338479 23669980 31921503 42527251 55969298
  72766411 93456735 118573819 148615999 184009768 225068513
  271948808 324607187 382760978 445857149 513053161
  583213481 654924586 726530060 796185813 861933650
  921789572 973841548 1016350163 1047844835 1067208183
  1073741824 1067208183 1047844835 1016350163 973841548
  921789572 861933650 796185813 726530060 654924586
  583213481 513053161 445857149 382760978 324607187
  271948808 225068513 184009768 148615999 118573819 93456735
  72766411 55969298 42527251 31921503 23669980 17338479
  12546502 8968770 6333469 4418237 3044772 2072809 1393998
  926112 607804 394060 252382 159681 99804 61622 37586 22647
  13480 7926 4604 2642 1497 838 463 253 136 73 38 20 10 5 2
  1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0)
;;Imaginary part.
(defarray psiI 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)

;; This subroutine changes a point in the real wave DEST
;; according to the curvature in the corresponding
;; neighborhood in the real wave SRC, and the potential at
;; the given point.
(defsub pfunc (dest src i alphas epsilon)
  ((dest _ i) += ((alphas _ i) */ (src _ i)))
  ((dest _ i) -= (epsilon */ (src _ ((i + 1) & 127))))
  ((dest _ i) -= (epsilon */ (src _ ((i - 1) & 127)))))

;; Take turns updating the two components of the wave in a
;; way such that they will chase each other around in
;; (higher-dimensional) circles.
(defsub schstep (psiR psiI alphas epsilon)
  ;; psiR += f(psiI)
  (for i = 0 to 127
     ;; psiR[i] += pfunc(psiI,i)
     (call pfunc psiR psiI i alphas epsilon))
  ;; psiI -= f(psiR)
  (for i = 0 to 127
     ;; psiI[i] -= pfunc(psiR,i)
     (rcall pfunc psiI psiR i alphas epsilon)))

;; Print the current wave to the output stream.
(defsub printwave (wave)
  (for i = 0 to 127
       (printword (wave _ i)))
  (println))

;; Main program, goes by the name of SCHROED.
(defmain schroed
  (for i = 1 to 1000 ;Time for electron to fall to well bottom.
     (call schstep psiR psiI alphas epsilon)
     ;; Print both wave components.
     (call printwave psiR)
     (call printwave psiI)))
```

6 R Compiler User's Manual

Currently the R compiler does not have a very convenient user interface. But this section describes the interface, such as it is.

The R compiler resides in the files `~mpf/lang/compiler/*.lisp`. To use it, start a Common Lisp such as Lucid (*e.g.*, by typing `M-x lucid` in Emacs), and type `(load "loader")`. This will load up the system.

Write your R program in a file such as `program.r`. Then type at Lisp, `(rcompile-file "program.r")`. First, the source will be printed with comments stripped out, and then, after a delay, the entire Pendulum assembly code for the compiled program will be printed. The example program `sch.r` takes about 15 seconds to compile on a Sparc 20.

Alternatively, one can type `(rcompile-file "program.r" :debug t)`, and the full program will be printed out after every tiny little step in the compilation process. The output from this is voluminous, but by doing an incremental search (`^S/^R`) in Emacs for the characters "`==>`" in this output, one can easily scroll through it to see what is going on at each compilation step. However, it is not recommended to try this with large programs.

When there are compilation problems, it is helpful to try compiling individual subroutines and statements in isolation, using the calls `(rc` *source*`)` and `(rcd` *source*`)`, where *source* is a Lisp expression that evaluates to the list representing the fragment of source code to be compiled. `Rc` prints out the compiled assembly code, and `rcd` prints out the complete state of the program after each compilation step. For example, `(rc '(a += b))` outputs the single Pendulum assembly language instruction "`ADD $2 $3`."

7 Compiler Internals

Once written, this section will describe the compilation infrastructure, document the low-level R constructs that are not yet recommended for prime time, and describe how to extend the compiler to handle new constructs.

8 Conclusions

R is a pretty cool little language, but it has a long way to go. It would be nice to have support for floating-point arithmetic, strings, structures with named fields, dynamic memory allocation, various built-in abstract data types, type checking and other error checking, exception handling, *etc., etc.* Not to mention object-oriented programming. It would be nice to have the option to use high-level irreversible operations, and have the compiler deal intelligently with the garbage data.

But anyway, here is revision 0.0, ready for feedback.