Configurable REC[*]

Gerardo Cisneros[†]
Dirección de Cómputo para la Investigación,
Dirección General de Servicios de Cómputo Académico,
Universidad Nacional Autónoma de México, Apartado postal 20-059,
01000 México, D.F., México

Abstract

This paper describes `CREC`, a library of C functions for compiling and executing `REC` programs whose operators and predicates are provided by the caller. Given a C program with a menu-based interactive interface, `REC` can be added as a programming language to the user interface by assigning menu choices to `REC` operators and predicates through entries in a table, and linking in the appropriate functions in `CREC`. Being a very concise language, `REC` programs are easy to write in an interactive context. The same conciseness allows the compiler to require only a small amount of space. Experience whereby the library benefitted some mathematics and physics programs is described.

1 Introduction

When a reasonably complex program is designed with a set of elementary operations to be chosen interactively, it is impossible for the designer to foresee all the variations that users will want to have (or wish they had) at one time or another, even though a complete and versatile collection of elementary operations has been provided. The user may likewise find the primitives quite satisfactory, but tire of entering them repeatedly.

That is the reason that many programs in this category, such as text editors, provide macro facilities. For interactive use, instructions that the user must supply need to be short, concise, easy to remember, and possessed of a simple and readily understandable (if not familiar) syntax. But this emphasis on conciseness at the keyboard should not preclude the introduction of macro libraries, containing the longer sequences that can be prepared at leisure, and with greater care.

Rather than work up a macro package for each new program, `REC`[1, 2] is proposed here as a model that can be readily adapted to each new program, and a library that serves this purpose is presented. `REC` is a concise programming language that in its barest terms defines only a control structure that is both completely general and yet very simple; its design was intended to capture the essence of programming much like group theory captures the essence of solving simple equations with a single operation, which gets called multiplication.

Even programs with a graphical interface (like those found under NextStep, Microsoft Windows or X windows) might benefit by this approach. For example, if there were a sequence of mouse-keypresses the user wanted to perform repeatedly, it could be easier for him to type in a `REC` program that invokes in turn the functions corresponding to the keypresses and iterates the sequence until a condition is satisfied or a certain number of iterations have occurred. Although the library was originally built and used on a PC-type computer running MS-DOS, it is sufficiently portable that incorporating it into programs with graphical interfaces should pose no great problems.

In summary, the idea of `CREC` is to make programmability available at the application's user interface, where access to shell scripts or other tools provided by operating systems would not be relevant or appropriate to operation of the application.

The syntax of `REC` may be described in a couple of rules. Beyond this description, a specific `REC` implementation depends in assigning concrete operations to the symbols in the predicate set; typically these symbols are the printing ASCII characters. There have been `REC` versions for arithmetic operations,[3] graphic display[4] and symbol manipulation,[2] among others; some have been implemented as interpreters, others as compilers. Because of the compactness of `REC`'s syntax, the compiler proper is quite small; in the symbol manipulation version for microcomputers it only takes about 1 KB.

[*]Based on the paper "Biblioteca de REC para construcción de interfaces interactivas flexibles" presented at the Seventh International Conference *Computers in Institutions of Education and Research*, Universidad Nacional Autónoma de México/Unisys, México, October 1991.

[†]Manuscript prepared while on sabattical leave from Departamento de Aplicación de Microcomputadoras, Instituto de Ciencias, Universidad Autónoma de Puebla, Apdo. Postal 461, 72000 Puebla, Pue., México, where the work described herein was performed. The author is now with Cray Research de México, S.A. de C.V., Camino a Sta. Teresa 480-A #302, Bosques del Pedregal, 14020 México, D.F., México.

The main advantage of REC is its conciseness: the library described here allows C programmers to add to their programs' user interfaces a small compiler with which the user may program frequently-used sequences of operations (in the spirit of keyboard macros), run predefined REC programs as provided by the programmer, or modify and run those predefined REC programs if simple editing capability is provided by the programmer. A set of functions for editing a line under MS-DOS, developed for the applications mentioned below, is available; under Unix one could use the `curses` library or invoke an editor as a child process. The single-character lexeme feature is justified by the convenience it provides when programming interactively; a compact compiler is an added benefit. The loss in debugability and readability that could be attributed to the use of single-character lexemes is not an issue; as a language to be used interactively no program is likely to be longer than one or two lines, and, if the primitive operations are chosen carefully, many useful programs would be a fraction of that.

This paper describes a threaded[5] interpreter for REC in C, in which the compiler is a function whose main arguments are

- the source program,
- an array of pointers where the compiled program is to be placed, and
- a predicate table that contains, for each printing ASCII character, a pointer to the function that compiles it, a pointer to the function that executes the associated operation, and a pointer to a string (typically a short comment regarding the character's function as a REC predicate, which may be displayed during program editing).

The rest of the library includes the necessary functions for compiling simple predicates and control symbols (the minimum required by REC's control structure), functions to compile and execute some predicates with a more complex structure than a single character (e.g., counters, numeric constants, symbolic constants), and a function to control the execution of a compiled REC program. Thus, a given implementation only needs to provide the predicate table and the functions implementing the operations associated with each version-specific predicate.

The remainder of the paper is organized as follows. First, REC syntax and semantics are reviewed. The following two sections give a rundown of the CREC library contents, with an emphasis on what the components are used for. Another section describes the steps required to incorporate the components into an existing application. In conclusion, some applications into which CREC has been incorporated are described.

2 REC Syntax and Semantics

The syntax of REC in Extended BNF is the following

$$\mathit{Prog} ::= \mathit{Expr} \mid \text{“–”}\{\mathit{Prog}\ \mathbf{char}\}\mathit{Prog}\text{“″”}$$

$$\mathit{Expr} ::= \text{“(”}\{\{\mathit{Prog} \mid \mathbf{pred}\}(\text{“:”} \mid \text{“;”})\}\{\mathit{Prog} \mid \mathbf{pred}\}\text{“)”}$$

where *Prog* and *Expr* represent a REC program and expression, respectively, **char** represents any printing ASCII character except the right brace (“}”), and **pred** represents any predicate (usually one or two characters, but in some cases a more complicated lexeme, for instance, a quoted string).

A brace-enclosed REC program is a list of subroutines (each of them a program followed by its name) and a main program. The predicate @α is reserved for calling the subroutine whose name is α. Subroutine calls are made through a table and names are bound dynamically: definitions given within a brace-enclosed list are activated when the corresponding main program starts executing and previous definitions for the same names are stored for retrieval when the main program terminates.

A REC expression is a parenthesis-enclosed list of strings of predicates or programs separated by colons and semicolons; there may be a final string terminated by the right parenthesis. A REC expression is executed left to right, with flow of control being altered in four cases:

1. A colon causes a jump to the beginning of the expression.
2. A semicolon causes execution to continue to the right of the expression (the expression terminates **true**).
3. The execution of a predicate, besides its assigned operation, results in a truth value. If it is **true**, flow of control is not altered; if it is **false**, execution of the string in which it appears terminates, and continues to the right of the string's terminator (a colon, semicolon or right parenthesis). Certain predicates are always **true**, so they are distinguished by being called “operators”.
4. Coming to a right parenthesis causes the expression to become **false** thereby causing a jump to the next segment in the enclosing expression, as with a **false** predicate. If there is no enclosing expression, but the terminating expression is a subroutine, or the main program of a subroutine, the corresponding predicate that called it becomes **false**; if the terminating expression is the main program of a REC program embedded in an expression, the embedded program is treated as a **false** predicate in the containing expression.

With these rules it is easy to see how to construct Boolean combinations of predicates or any structured programming construct. Boolean combinations in general do not commute due to the possible non-reversibility of operations performed by the predicates; lazy evaluation is implicit in the rules for flow of control. There are no labels in REC, thus disallowing arbitrary jumps. Table I lists some common constructs.

Construct[a]	REC *Expression*
π_1 **or** π_2 **or** $\cdots$ **or** π_n	(π_1 ; π_2 ; $\cdots$; π_n ;)
π_1 **and** π_2 **and** $\cdots$ **and** π_n	(π_1 π_2 $\cdots$ π_n ;)
not π	(π)
true[b]	(;)
false	()
if π **then** ω_1 **else** ω_2	(π ω_1 ; ω_2 ;)
while π **do** ω	(π ω :;)
do ω **until** π	(ω π ;:)

[a] π, π_1, π_2 and π_n represent arbitrary predicates (including programs and expressions); ω, ω_1 and ω_2 represent operators, i.e., permanently **true** predicates.
[b] The null string ϵ is also **true** within a REC expression.

Table 1: REC expressions for common programming constructs

Outside of the formal definition given above, square brackets are reserved for enclosing comments (which may be nested), and the space and comma are reserved as noops, for use in improving the readability of REC programs. Other whitespace characters (tabs, carriage returns and line feeds) are also skipped over.

3 Interpreter Library

In what follows, a thorough familiarity with C is assumed. The CREC interpreter is a library of C functions to compile and control the execution of REC programs. Functions carrying out the operations associated to predicates and operators in a given version are provided by the caller by means of a table; this is what makes CREC configurable.

The library is available in both MS-DOS and UNIX versions. There are two versions of each: a minimal one that does not use `malloc` nor the `stdio` functions[6] and in which the REC program to be compiled may only be given directly in a character string argument; and another that does use `malloc` and `stdio`, and allows having the compiler read the program from a string, the standard input or an arbitrary file. The version using `malloc` and `stdio` is described in this section; differences distinguishing the minimal version are given in the following section.

The name of the library (under UNIX) is `librec.a`. The main module, `rec.o`, contains the basic functions for compiling and executing; other modules define certain specialized predicates and operators programmers may want to incorporate into their specific versions of REC.

Programs are compiled into an array of `Inst`, which is defined as a pointer to a function that returns `int`s. Operators are compiled by simply placing the executing function's address on the table; predicates use an extra cell containing the address to jump to. Compound operators and predicates (e.g. @α) use another cell to contain or point at the argument. The idea for using an array of pointers to functions (which early assembler versions used, before the term "threaded code"[5] was coined), rather than an array of integers as in the early Fortran interpreter,[3] was derived for the present version from Chapter 8 of Kernighan and Pike's excellent book on UNIX.[7]

The compiler uses an internal stack to account for parenthesis and brace nesting. Three cells in the stack are taken up for each level of parenthesis nesting, and one cell per brace pair plus two cells per subroutine defined within the pair for brace nesting. This same stack is used during REC program execution to save subroutine definitions when brace-enclosed programs are invoked. (Each invocation takes one cell per subroutine defined within the pair of braces.) The stack size is 1024 in the package as released, but source code is included so that the size may be increased if deeper recursion is required.

Two header files, `rec.h` and `rectbl.h` are included. `rec.h` contains general declarations and definitions for types `Inst` and `Symbol`. The latter is used by the optional number and string operator subroutines. `rectbl.h` contains the predicate table, which must be modified for each specific predicate set and be included by only one of the programmer's modules, usually the one containing the call to the compiling function, `rec_c`.

Global data structures or variables that are acted upon by the programmer-provided predicates should be initialized, if needed, before calling `rec_x`, the function that executes a compiled REC program. Other possibilities are to include an initializing operator, or to use predicates with arguments such that a certain value of the argument causes the required initialization.

To give an idea of the small size of CREC, Table II gives the sizes of the modules on a NeXTstation. The entire library, `librec.a`, takes only 23378 bytes. Sizes are slightly larger for both a DECstation 5000/200 and a Sun SPARCstation 2. Using Turbo-C to compile the modules, the MS-DOS version of the library is 12800 bytes long.

3.1 Module rec.o

Of the functions in the list that follows, the first two are CREC's main interface to the programmer; others are

Module	*Text Size*	*Data Size*
`rec.o`	3376	4828
`sym.o`	292	0
`tty.o`	264	16
`cnum.o`	892	0
`cquo.o`	836	2068
`ctr.o`	260	0
`mem.o`	68	0

Table 2: Sizes of modules in `librec.a`

invoked indirectly through the predicate table, although they could be called by predicate compilation or execution functions written by the programmer. Global variables included in the list are those required by some of the predicates.

3.1.1 Function `int rec_c(int stype, char *source, Inst *prog, int plen, struct fptbl *table)`

This is the compiling function. `stype` determines where the source program comes from, as follows:

`stype` $= 0$,	`source` is not used; the program is read from standard input;
`stype` $= 1$,	`source` is the name of a file containing the program;
`stype` > 1,	`source` is a pointer to the program itself, `stype` is the string's length.

`prog` is a pointer to an array of `Inst` (allocated by the caller) where the compiled code is to be placed, `plen` is this array's length, and `table` is the predicate table defining for each printing ASCII symbol (from space to tilde inclusive) the function that compiles it, the function that executes it, and a pointer to a string describing it. Structure `fptbl` is declared in `rec.h` as follows:

```
struct fptbl {
  int   (*cfun)(); /* compiling function */
  int   (*xfun)(); /* executing function */
  char  *r_cmnt;   /* descriptive comment */
};
```

and a typical entry in the table is a line such as

```
r_code,   ropux,  "X - store rule table on disk  ",
```

where `r_code` is the `CREC` function to compile a simple operator, `ropux` is the programmer-defined function that carries out the function assigned to the corresponding character, and the string (whose address is the third element of the `fptbl` structure) is intended as an aid for display during program editing to remind the user of the character's assigned function as the cursor moves over it; this helps overcome one disadvantage of using single-character lexemes. The programmer may also prepare help screens for display using the strings available in the table; by placing these strings in the table alongside the definitions, a string is more likely to be updated when the corresponding character's definition changes than if it were located in a separate help section.

The file `rectbl.h` included in the distribution provides a skeleton table that can be modified to suit the needs of a given implementation.

`rec_c` returns 1 (**true**) if there were no errors and -1 if termination was due to an error. This value should be used to control the execution of `rec_x`, since execution of an incompletely compiled program is bound to cause problems.

Upon succesful compilation of a `REC` program, the global pointer `r_pc` points to the next available cell in `prog`. This way one may define a freestanding subroutine α that is not lost with the next compilation by assigning `r_vst['`α`'] = prog`, copying `r_pc` to a local variable (say, `pp`) and giving further calls to `rec_c` second and third arguments `pp` and `plen - (pp - prog)`, respectively, or using a different program array altogether. If the subroutine contains number or string operators (referencing `r_cnum` from `cnum.o` or `r_dquote` from `cquo.o`), care should be taken to save also the linked list referenced by the global pointer `r_symlist`, because `rec_c` frees symbol space before each new compilation. To save, copy `r_symlist` to a local pointer to `Symbol` (say `sp`) and null `r_symlist`; to free the space, execute `r_frsym(sp)`. This feature lets programmers build their own internal libraries of predefined `REC` functions.

3.1.2 Function `int rec_x(Inst *pc)`

The execution controller function; `pc` points to the place where execution should begin; it should be the same as the pointer previously passed as `prog` in a succesful call to `rec_c`. The value returned is the final truth value of the `REC` program (0 for **false**, 1 for **true**) if there were no errors, or -1 if there was an error.

Before starting the execution of the `REC` program, `rec_x`, sets up to trap the interrupting character (typically control-C), so that interruption of a running `REC` program does not kill the entire application, but returns to the caller of `rec_x` with an error indication. Since this feature depends on the operating system, the module containing this function (`rec.c`) is compiled with the flag -DMSDOS for the MS-DOS library; without it for UNIX.

3.1.3 Function `int r_code(Inst func)`

Compiler for simple operators, i.e., operators fully specified by a single character. `func`, taken from the predicate table, is a pointer to the function that performs the operation at runtime. The function is called `r_code` rather than `r_oper` because other compiling functions also call it to insert code into the program array.

3.1.4 Function `void r_compile()`

Compiles a program, recursively. It is provided so that, if desired, it can be invoked at runtime by an operator that compiles a program. Such an operator would have to set up the appropriate input stream in the global variable `FILE *r_fin`. If `r_fin` is `NULL`, the program is "read" from a string specified by the globals `char *r_pbufp` (a pointer to the string) and `int r_pbufl` (the string's length). Of course, a predicate would also have to be set up to execute a program compiled this way, or provision would have to be made to insert in the subroutine table, `r_vst`, the address of the array where the code is stored so that it is accessible to `@`α.

3.1.5 Function `void r_errterm(char *msg)`

Error termination. Writes the string `msg` to the standard error file and executes `longjmp`, so that `rec_c` or `rec_x` returns -1 to the calling program, no matter how deeply nested in calls the error occurred.

3.1.6 Function `int r_initvst()`

Initializes `r_vst` to the state prevailing at start of execution: elements 0 through 32 and 127 (corresponding to ASCII control characters) are set to zero; elements 33 through 126 ('`!`' through '`~`') are set to point to an `Inst` cell containing the address of an error subroutine, `r_usub`.

3.1.7 Function `int r_oper1(Inst func)`

Compiles an operator taking an ASCII argument (i.e., one in the form $\chi\alpha$, where χ is the operator and α the argument). Argument `func` is taken by the compiler from the predicate table entry for χ; `r_oper1` reads the character following χ in the source program and places it after the value of `func` in the compiled code. Given an `int` variable `c`, the programmer's code should use an expression such as `c = (int) *r_pc++;` in the executing function to retrieve the character and leave `r_pc` pointing at the next instruction.

3.1.8 Function `int r_oper2(Inst func)`

Compiles an operator with two ASCII arguments, i.e., an operator in the form $\chi\alpha\beta$. `func` is the execution address for χ. `r_oper2` reads two more characters from the source and places them packed into a cell following `func` in the compiled program. The following would retrieve them into variables `one` and `two`:

```
int one, two;
one = ((int) *r_pc) >> 8;
two = ((int) *r_pc++) & 0xFF;
```

and would leave `r_pc` ready for the return.

3.1.9 Variable `Inst *r_pc`

The program counter. Access to it is required by the execution routines of predicates and operators with compiled arguments.

3.1.10 Function `int r_pred(Inst func)`

Compiles a simple predicate. `func` points to the runtime address. Function `func` must return 0 for a **false** exit or 1 (or any other nonzero value, so as to conform to C practice) for a **true** exit.

3.1.11 Function `int r_pred1(Inst func)`

Compiles a predicate with one ASCII argument, such as `@`α. `func` is the execution address. `r_pred1` reads one more character from the source and places it following `func` in the compiled program. If `c` is an `int`, `c = *r_pc++;` retrieves the ASCII argument and leaves `r_pc` ready for the return. `func` must return 0 for a **false** exit or 1 (or any other nonzero value) for a **true** exit.

3.1.12 Function `int r_pred2(Inst func)`

Compiles a predicate with two ASCII arguments. `func` is the execution address. `r_pred2` reads two more characters from the source and places them packed into a cell following `func` in the compiled program. The following would retrieve them into variables `one` and `two`:

```
int one, two;
one = ((int) *r_pc) >> 8;
two = ((int) *r_pc++) & 0xFF;
```

leaving `r_pc` ready for the return. `func` must return 0 for a **false** exit or 1 (or any other nonzero value) for a **true** exit.

3.1.13 Array `Inst *r_vst[128]`

Addresses of subroutines are dynamically loaded into and unloaded from this array. Array elements 0 through 32, 125 and 127 do not correspond to valid subroutine names, so they are available for other pointer storage; a suitable operator could be provided for this purpose (such as the `$` operator of REC/MARKOV).

Access to the source program for the compilation of special operators is provided by `rec.o` through two variables: `FILE *r_fin`, a pointer to the source stream, and `int r_lastchar`, the last character read, and two functions: `int r_nxtchar(FILE *inp)`, which returns the next character from the source stream, and `int r_ungetch(int c, FILE *inp)`, which returns the last character read to the source stream.

Table III summarizes other symbols defined in `rec.o`.

Name	*Purpose*
`r_call`	Executes predicate `@`α.
`r_colon`	Compiles colons.
`r_comment`	"Compiles" comments by skipping over them.
`r_lbrace`	Compiles brace-enclosed programs.
`r_lpar`	Compiles left parentheses.
`r_noopc`	Compiles null operators (it generates no code).
`r_quit`	Ends execution through `r_errterm`.
`r_rpar`	Compiles right parentheses.
`r_semicol`	Compiles semicolons.
`r_ubrace`	Ends compilation due to unbalanced right brace.
`r_ubrack`	Ends compilation due to unbalanced right bracket.
`r_usub`	Default function for undefined subroutines.
`r_xbrace`	Executes brace-enclosed programs.
`r_xpred`	Executes predicates.

Table 3: Additional symbols in `rec.o`

3.2 Module sym.o

`Sym.o` contains symbol table management routines. Entries in the symbol table are generated by the quoted string and number operator compilers `r_dquote` and `r_cnum` contained in `cquo.o` and `cnum.o`, respectively. There are two global functions defined in `sym.o`. Function `Symbol *r_install(char *s, int len)` enters string `s` of length `len` into the symbol table (if it is not already there) and returns a pointer to the entry. Function `Symbol *r_lookup(char *s, int len)` looks up string `s` of length `len` in the symbol table and returns a pointer to the entry if found, the `NULL` pointer otherwise.

3.3 Module tty.o

This module, containing terminal control routines, must be compiled with one of the flags -DMSDOS (for a compiler running under MS-DOS), -DSYSV (for a compiler running under UNIX System V) or -DBSD (for a compiler running under BSD UNIX). The module contains three functions. Function `void r_ttget()` gets the current terminal settings, stores them in a static structure. Function `void r_ttreset()` resets terminal to the settings stored by `r_ttget`. Function `void r_ttset()` sets the terminal to "raw" mode; this allows an operator such as `REC`/MARKOV's `R` to have character by character control over the keyboard.

3.4 Module cnum.o

Optional module for compiling numbers. Three types are recognized: `WORD` (no decimal point, no exponent, no leading zero, except for a single zero), `LONG` (no decimal point and no exponent, must have at least two digits the first of which is 0) and `REAL` (decimal point and/or exponent must be present).

Inclusion of this module requires that the entry for the compiling function name in the predicate table, for all digits and the period, be `r_cnum`. The compilation entry for the minus sign must be `r_cnum` if the minus sign is to be used only in connection with numbers, or `r_cmin` if a plain minus sign (not followed by a period or digit) is to be compiled as a simple operator.

3.4.1 Function `void r_cmin(Inst func)`

Provided as an entry point for the compilation of "`-`" as an operator. `func` is the execution entry point for a plain "`-`", but if the "`-`" is immediately followed by a digit or period, `r_cnum` is invoked with argument `r_ld` (which is expected to be the programmer-provided entry point for retrieving a number from the symbol table).

3.4.2 Function `void r_cnum(Inst func)`

Compiles a number, which is installed in the symbol table. The pointer to the symbol table entry follows the pointer `func` in the compiled program. The programmer must provide the function that retrieves the number, and this function's name must be `r_ld` if `r_cmin` is used to compile numbers preceded by a minus sign.

3.5 Module cquo.o

Optional module for compiling quoted strings. Both singly quoted characters and doubly quoted strings are considered. Backslash-escaped characters as in C are allowed; they are processed by an internal (static) function `r_bslash`.

3.5.1 Function `void r_dquote(Inst func)`

The enclosed string is stored in the symbol table. If the string exceeds 2048 bytes (the size of the buffer), compilation ends with an error message. `func` is the programmer-provided execution function for the double quote character. The execution function should retrieve the pointer to the symbol table node as `(Symbol *)(*r_pc++)`.

3.5.2 Function `void r_squote(Inst func)`

The enclosed character is coded into the program array to be retrieved by the execution function as `(int)(*r_pc++)`.

3.6 Modules ctr.o and mem.o

`Ctr.o` provides the entry points necessary for compiling and executing counter predicates, `r_ctrc` and `r_ctrx`, respectively. The customary syntax is `!`n`!`, with n a fixed integer; other likely candidates for assignment of this predicate could be `$`n`$` or `#`n`#`. A counter `!`n`!` is a predicate with $n+1$ states. The first n times the predicate is executed it returns **true**; the next time it returns **false**. The $n+1$ state cycle repeats and cannot be reset except by placing the counter in a subroutine of its own and setting up another subroutine that exhausts the cycle: `{(!`n`!;)a (@a:;) b (···)}`.

Module `mem.o` provides a single entry point, `(char *) r_malloc(int size)`, which calls the standard library function `malloc` and calls `r_errterm` if `malloc` returns a null pointer. It is used by the functions in `sym.o`.

4 Minimal version

The minimal version was developed to comply with a request for a version that made minimal demands on the standard C library, specifically one that would not invoke `malloc` or file-handling functions in `stdio`. Thus, modules `sym.o`, `cnum.o` and `cquo.o` are not present. A source file cannot be given to the compiling function `rec_c`; it takes one less argument. Modules are provided that contain functions to compile and execute operators for program literals (strings, characters, integer constants, floating point constants), which refer to the literals by means of pointers to the source string or which save numeric constants in the code array.

4.1 Module rec.o

Functions that are different in `rec.o` are `rec_c`, `r_errterm`, `r_nxtchar`, `r_ungetch`. The declarations for these functions are as follows:

```
int rec_c(char *source, Inst *prog, int plen,
                         struct fptbl *table)
int r_errterm(int msgno)
int r_nxtchar()
int r_ungetch()
```

In `rec_c`, `source` must be a pointer to a string containing the `REC` program to be compiled. Since `stdio` is not invoked, the argument to `r_errterm` is an integer, whose value is returned by execution of `longjmp` to `rec_c` or `rec_x`, which in turn return the negative of the value to the caller, providing it with an error indicator from whose value the type of error may be discerned.

Again, since no files are involved, `r_nxtchar` and `r_ungetch` need no arguments.

The globals `r_fin` and `r_symlist` of the `malloc/stdio` version are not declared in the minimal version.

Two global variables are declared for string handling: `char *r_pbufp` and `int r_pbufl`. At the beginning of compilation, they hold the address and length, respectively, of the input buffer from which the source program is read, and are updated by `r_nxtchar` and `r_ungetch` as compilation progresses. During execution, these two variables are used by the functions in module `str.o` to hold the address and length of strings quoted within the program.

4.2 Module int.o

Provides the entry points necessary for compiling and executing an (`long`) integer parameter operator. If used, the names `r_cintp` and `r_lintp` must appear as the compilation and execution entry points for a character other than the decimal digits in the predicate table. The parameter is loaded at execution time into the global variable `long r_intpar`.

4.2.1 Function `int r_cintp()`

Compiles an integer parameter. The syntax is $\delta n \delta$, where n is decimal and δ is the operator chosen as the delimiter; `#` is suggested as the character to be used both as the operator and the delimiter.

4.2.2 Function `int r_lintp()`

Loads the integer parameter into the global variable `r_intpar`. This may then be used by subsequent operators or predicates.

4.3 Module dbl.o

Provides the entry points necessary for compiling and executing a `double` parameter operator. If used, the names `r_cdblp` and `r_ldblp` must appear as the compilation and execution entry points for the character chosen as the operator (which should not be a digit or the period) in the predicate table. The parameter is loaded at execution time into the global variable `double r_dblpar`.

4.3.1 Function `int r_cdblp()`

Compiles a double parameter. The syntax is $\delta n \delta$, where n is a decimal number (with optional decimal point and optional power-of-ten factor denoted by `e`k or `E`k, where k is the exponent) and δ is the operator chosen as the delimiter. `$` is the suggested character to use for this operator.

4.3.2 Function `int r_ldblp()`

Loads the double parameter into the global variable `r_dblpar`. This may then be used by subsequent operators or predicates.

4.4 Module fnum.o

This module contains functions with the same names as those in `cnum.o` discussed in the previous section; however, in this version of the library all numbers are treated as double precision numbers. No incompatibility arises from using both `fnum.o` and `dbl.o`; `fnum` allows numbers without delimiters to be compiled as operators in a `REC` program.

Proper use of this module requires that the compilation entry in the predicate table for all digits and the period be `r_cnum`. The compilation entry for the minus sign must be `r_cnum` if the minus sign is to be used only in connection with numbers, or `r_cmin` if a plain minus sign (not followed by a period or digit) is to be compiled as a simple operator.

The programmer is responsible for providing a function called `r_dnum` that must be listed as the execution name for the digits and the period and that calls `r_fnum()` once to fetch the number out of the program array. If the compilation address for the minus sign is listed as `r_cmin`, the user must also provide a function (whose name is not fixed by module `fnum.o`) to be listed as the execution address for a plain minus sign.

4.4.1 Function `int r_cnum()`

Compiles a floating point number. The syntax requires that at least one digit or the decimal point be present; the number may be preceded by a minus sign and followed by a power-of-ten factor whose form is an `e` or `E`, an optional sign (`+` or `-`) and zero or more digits. No embedded blanks are allowed. The string is converted by the standard function `sscanf`, defaulting to 0 if the string is not converted by `sscanf`, as in the case of a single decimal point. The resulting value is stored in the program array.

4.4.2 Function `int r_cmin(Inst func)`

Compiles a minus sign. This function looks ahead in the input stream; if a digit or period is found, the minus sign is assumed to be a part of a number, so compilation continues at `r_cnum`, with execution address of the operator given as `r_dnum`. Otherwise, it compiles as a simple operator with execution address given by argument `func` (that is, from what is listed in the predicate table).

4.4.3 Function `double r_fnum()`

Returns the double value stored in the program array; this function must be called once by the user-provided function `r_dnum` that is to be the execution address for all digits and the period.

4.5 Module chr.o

Provides the entry points necessary for compiling and executing a character parameter operator. If used, the names `r_cchrp` and `r_lchrp` must appear as the compilation and execution function entries of the predicate table element for the character chosen as operator (and delimiter).

4.5.1 Function `int r_cchrp()`

Compiles a character parameter. The syntax is $\delta\alpha\delta$, where α is a character and δ (typically the apostrophe) is the character chosen as operator and delimiter. The C conventions regarding the backslash as an escape character are implemented. The quoted character α is stored in the program array.

4.5.2 Function `int r_lchrp()`

Loads the quoted character into the global variable `r_chrpar`. This may then be used by subsequent operators or predicates.

4.6 Module str.o

Provides the entry points necessary for compiling and executing a string parameter operator. If used, the names `r_cstrp` and `r_lstrp` must appear as the compilation and execution entry names for the chosen operator (and delimiter) in the predicate table. Note that if a `REC` program is to be included as a constant string in a C program, and the `REC` program contains quoted strings, the outer quotes must be escaped by a backslash, and any inner characters needing escaping must be doubly escaped, e.g., `"(\"Isn\\\'t this ridiculous?\";)"`. In this example the string parameter compiler `r_cstrp` would compile the string `"Isn\'t this ridiculous?"`

4.6.1 Function `int r_cstrp()`

Compiles a string parameter. The syntax is $\delta\omega\delta$, where ω is the string and δ is the character chosen as operator and delimiter, usually the double quote character.

The C conventions regarding the backslash as an escape character are implemented, but they are *not* interpreted during compilation, because the string is not moved from its location in the `REC` program source string.

4.6.2 Function `int r_lstrp()`

Loads the string parameters, address and length, into the global variables `r_pbufp` and `r_pbufl`, respectively. The length is that of the uninterpreted string without the delimiters, the address is that of the character following the left delimiter. These variables are declared in module `rec.o`, and may be used by subsequent operators or predicates, which should use `r_bslash` to interpret properly characters escaped by a backslash.

4.7 Module bsl.o

Contains a function, `r_bslash`, that interprets characters escaped with \ according to C conventions. This function is called by `r_cchrp` in `chr.o` and is provided as a separate module for use with the string-handling functions in `str.o`. Function `int r_bslash(int c)` returns `c` if not a backslash; if `c` is a backslash, it returns a value depending on the character following it:

`\a`	alarm (BEL)
`\b`	backspace (BS)
`\f`	form feed (FF)
`\n`	newline (LF)
`\r`	carriage return (CR)
`\t`	tab (HT)
`\v`	vertical tab (VT)
`\x`n	hexadecimal number n (at most 3 digits)
`\`ω	octal number ω (at most 3 digits)
`\`α	character α, when α is none of the above.

5 Case study

Consider adding some operators to CAMEX,[8] an exerciser program for the Automatrix, Inc., CAM-PC cellular automata board. The program contains provisions to allow the user to move data between each of the board's bitplanes and a disk file. Instead of setting up a popup menu that asks what bitplanes are to be saved or loaded and file names for each bitplane, two operators `Y`α and `y`α are added into CAMEX's set of `REC` predicates. `Y`α will store the plane indicated by α (0, 1 or 2) in the file whose name is given by the last quoted string operator (c.f. `str.o` above) and `y`α will load the file given by the last quoted string into the bitplane specified by α (1, 2 or 4—a caprice of the CAM-PC, not `REC`).

What has to be done to the program is

- Include in the header file containing the predicate table the declarations for the functions that execute operators `Y` and `y`:

```
int ropuy(), roply();
```

- Change the corresponding lines in the table:

```
struct fptbl dtbl[] = {
    r_noop,     FALSE,
      "Space - no op                  ",
    r_ctrc,     r_ctrx,
      "! - Counter !n!, n decimal    ",
    r_cstrp,    r_lstrp,
      "\" - String parameter          ",
    /* entries for # through W */
    r_oper1,    ropuy,
      "Y - store plane in disk file  ",
    /* entries for Z through w */
    r_oper1,    roply,
      "y - load bitplane n from disk ",
    /* entries for z through ~ */
};
```

Since the syntax for the new operators is `Y`α is and `y`α, their compilation entry point is listed as `r_oper1`.

- Add the functions that carry out the associated operations:

```
ropuy() {pltofi(r_pbufp,nibbl(*r_pc++));}
roply() {fitopl(r_pbufp,nibbl(*r_pc++));}
```

Function `nibbl` converts a hexadecimal ASCII digit to the corresponding integer value; `pltofi` and `fitopl` carry out the actual transfers between disk file and bitplane. They assume that `r_pbufp` has been set up previously by execution of the quoted string operator; `*r_pc++` picks up the character α seen during compilation and advances the program counter.

- Provide an array for `REC` programs; add sample programs to it:

```
char recrule[NY][CLEN]= {
 /* ... */
 "([store]\"WW1.PAT \"Y0\"WW2.PAT"
        " \"Y1\"WW4.PAT \"Y2\" \";;)",
 "([load] \"WW1.PAT \"y1\"WW2.PAT"
        " \"y2\"WW4.PAT \"y4\" \";;)",
 /* ... */
};
```

NY is the number of program strings; CLEN is the maximum size of each string. Editing, compiling and executing either of the two REC programs shown replaces going through the three dialogues that would be required to set up the file names associated with the bitplanes. The bracketed comments inside each sample program remind the user what the program does, lest the operations associated with Yα and yα be forgotten or confused.

- In the keyboard event loop, provide for ways to choose a REC program, edit it, and compile and execute it:

```
case Kf2 : /* f2 = rec program list */
    rij=lim(1,recdem(Kf2),NY)-1;
    camri(rij);
    for (i=0; i<CLEN; i++)
        cstr[i]=recrule[rij][i];
    break;
case Kf3: /* f3 = edit rec program */
    edrec(cstr); txtmode();
    break;
case Kf4: /* f4 = execute rec program */
    if (rec_c(cstr,rprg,PLEN,dtbl)==1)
        rec_x(rprg);
    break;
```

  In this code fragment lim(i,j,k) returns i if j<i, k if j>k and j otherwise; recdem displays the menu of built-in REC programs and returns the index of the selection, camri displays the name of the selected REC program, edrec allows editing of the program, and txtmode resets the screen after editing. Kf2, Kf3 and Kf4 are symbolic constants whose values are those produced by the PC function keys F2, F3 and F4, respectively; cstr is an array of characters to hold the program during editing and compiling, rprg is the program array and PLEN is its size.

If predicates are to be included, one must make sure their executing functions return integer values appropriate to the corresponding truth values.

6 Experience

CREC has been used to provide programmability to the menu interfaces of the following programs:

- CAMEX,[8] described in the previous section
- LCAU41,[9] a set of programs for the study of (4,1) linear cellular automata: their evolution, statistics, etc.
- TWOC,[10] a program to study the trajectories of a charged particle in the field of two centers with independent magnetic and electric charges.

Besides the added programmability, the menus now include sets of sample REC programs that may be executed as they are, or edited and then executed.

The inclusion in CAMEX of the Y and y operators and the two sample programs described in the previous section provided several benefits:

- File names may be changed quickly, including putting in path names. This can't be done with Forth, as included with the CAM-PC board.
- A different set of planes may be loaded, or some may be dropped. Being editable, the sample programs can be taken just as a suggestion.
- If it were worth the trouble, yα could become a predicate (returning **false** for a nonexistent file), allowing the user to specify alternates to be loaded in order of priority.
- The programmer can also reuse the new operators in other REC programs, which are much shorter to write than new C source.

From LCAU41, the following sample REC program runs through the built-in set of demonstration rules of (4,1) automata (4 states, first neighbors), ending when the set is exhausted or a key is pressed:

```
(D F (y g N Z h: s;);)
```

where

D selects an internal "demo" category switch (operators for other categories include T for "totalistic rules" or I for "invertible rules"),

F selects the first rule in a category,

y generates a random line to start the automaton (other operators can generate other lines, such as a single point in the middle),

g graphs a screenful of the automaton's evolution, displaying the rule on the first row,

N moves to the next choice in the current category,

Z is a predicate that returns **true** if the current rule is *not* the last selection in its category,

h is a predicate that clears the screen, restores the main display and returns **false** if a key has been pressed (allowing the user to decide when to interrupt an ongoing cycle), and

- `s` clears the screen and restores the main display (used in this example for the case in which the iteration ends by `Z` becoming **false**).

As a final example, the following program from `TWOC` follows a trajectory with given initial conditions that are saved in case a reflected plot is desired; displayed trajectory parameters are updated every ten steps, and the program ends when a key is pressed:

```
(i m p S ((!10! r p:;) Pa k:;) ;)
```

In this program,

- `i` initializes a Runge-Kutta run,
- `m` evaluates the constants of the motion for the current position and velocity,
- `p` plots the position on the screen (with a green point),
- `S` saves the space vector,
- `!10!` is a counter predicate,
- `r` performs one Runge-Kutta step,
- `P`α prints selected data on the screen ($\alpha =$ `a` causes all parameters to be displayed), and
- `k` is predicate returning **false** if a key has been pressed.

`TWOC` contains many variables and parameters, and provides several kinds of graphs. Being able to choose which of them is wanted at a given moment, and in what sequence, is the flexibility offered by `REC`. If the program prints coordinate values all the time, the display slows down and the viewer is distracted; if the coordinates are never displayed, the user does not know what is going on. Thus, in the example, the counter determines how frequently the coordinates are displayed. Another counter could be used to decide how long to run a trajectory before changing the initial conditions. Once programmability has been added to the user interface, the user's possibilities become endless.

6.1 Availability

All versions of `CREC` are available from the author, who can be reached on BITNET as CISNEROS@UNAMVM1 or on the Internet as `cisneros@unamvm1.dgsca.unam.mx`. The software distribution includes a small sample compiler that is essentially a reverse Polish notation calculator; copies of `LCAU41`, `TWOC` and `CAMEX` are also available.

7 Acknowledgments

Suggestions by H. V. McIntosh were invaluable for the development of the library described herein and many of his comments were incorporated into the revised manuscript. The author was supported in part by the Consejo Nacional de Ciencia y Tecnología (grant #P228CCOX904651) the Secretaría de Educación Pública (grant #C90-01-0473) and the Sistema Nacional de Investigadores.

References

[1] H. V. McIntosh, "A CONVERT compiler of REC for the PDP-8," *Acta Mex. Cienc. Tecnol.* **2**, 1 (Jan-Apr 1968), 33–43.

[2] H. V. McIntosh and G. Cisneros, "The programming languages REC and Convert," *SIGPLAN Notices* **25**, 7 (July 1990) 81–94.

[3] G. Cisneros, "A FORTRAN coded Regular Expression Compiler for the IBM 1130 Computing System," *Acta Mex. Cienc. Tecnol.* **4**, 1 (Ene-Abr), 30–86 (1970)

[4] R. Carlos García-Jurado M., "Un REC visual para la PDP-15 en comunicación con la PDP-10," B.Sc. Thesis (Spanish), Instituto Politécnico Nacional, México (1971)

[5] J. R. Bell, "Threaded code," *Commun. ACM* **16**, 370–372 (1973)

[6] B. W. Kernighan and D. M. Ritchie, "The C Programming Language," 2nd. edition, (Englewood Cliffs, NJ: Prentice-Hall, 1988), Ch. 7.

[7] B. W. Kernighan and R. Pike, "The UNIX Programming Environment," (Englewood Cliffs, NJ: Prentice-Hall, 1984), Ch. 8.

[8] H. V. McIntosh, "The CAM/PC exerciser CAMEX," Instituto de Ciencias, Universidad Autónoma de Puebla (1991) [An abridged version appeared in "CAM News: A newsletter for users of CAM-PC/CAM-6," Robert B. Andreen, ed., Mount Saint Mary College, Newburgh, N.Y., Sept. 1991]

[9] H. V. McIntosh, "LCAU," Instituto de Ciencias, Universidad Autónoma de Puebla (1990), cited in D. Hiebeler, "Appendix I: A brief review of cellular automata packages," *Physica D* **45**, 463–476 (1990)

[10] H. V. McIntosh, "TWOC," Instituto de Ciencias, Universidad Autónoma de Puebla (1991)