Generic REC compiler in C

Gerardo Cisneros
Depto. de Aplicación de Microcomputadoras
ICUAP
3.4.91, Rev. 3.10.91

16.10.91

This directory contains a library with subroutines for compiling and executing generic REC programs. The main module rec.c contains the basic compilation and execution subroutines; the other modules define specialized operators and predicates that the user may optionally include in his customized version of REC. This version uses malloc to allocate storage for program constants (numbers and strings).

The arguments given to the compiler subroutine are:

- an integer indicating where the source is to be obtained from
- a string with the REC program (type 'char *')
- a pointer to the object code array, which MUST be an array of type 'Inst', a type defined in rec.h.
- the length of the object code array (type 'int')
- a pointer to the table with the definitions for predicates and operators (a triple for each ASCII character between space and tilde consisting of two pointers to functions specifying type of compilation and place of execution and a pointer to a string, which may be initialized to a short comment describing the character's function). This table, hereafter called the compilation/execution table, should be modified from the sample one included in rectbl.h.

The compiler uses an internal stack to keep track of parenthesis and brace nesting; it is declared to have 1024 cells. Each parenthesis level requires three cells, each brace level requires two cells per subroutine plus one extra cell.

The execution subroutine only requires the pointer to the object code array. A program incorporating REC should '#include' rec.h (which contains basic declarations) in every module; the main module (or the one calling subroutine rec_c) should '#include' rectbl.h instead of rec.h, suitably

modified for the set of desired predicates and operators.

The same internal stack used by the compiler is used to keep track of brace execution; each entry into a brace-enclosed program takes one cell per subroutine. This has to be taken into consideration when writing complicated recursive programs.

Any data structures or pointers acted upon by the predicates and operators which need to be initialized should be initialized prior to calling rec_x to execute the REC program.

The entry points defined in rec.c are the following (the first two being the functions called directly by the user, and the others referenced via the compilation/execution table and possibly by compilation and execution routines provided by the user):

rec_c int rec_c(int stype, char *source, Inst *prog, int slen,
                 struct fptbl *table)

The compiling functions; 'stype' determines the program source as follows:

stype = 0,	'source' not used, program is read from the standard input;
stype = 1,	'source' is the name of a file containing the program;
stype > 1,	'source' points to the program itself, 'slen' is the length of the program.

'prog' is the pointer to an array (provided by the user) in which the generated object code will be placed, 'plen' is the length of the object array and 'table' is the table containing compiling/executing pointer-to-function pairs for the printing ASCII characters (rectbl.h provides a skeleton table which can be modified to suit the particular compiler's needs).

The value returned is 1 (TRUE) if there were no errors, or a -1 if termination ocurred due to an error. The return value should be used to control execution of rec_x.

Upon successful compilation, the global pointer r_pc points at the next available cell in the array pointed at by 'prog'. Thus a standalone subroutine x may be defined by setting r_vst['x'] = prog, copying r_pc to a local variable (say 'pp') and performing further compilation with 2nd and 3rd arguments

'pp' and 'plen - (pp - prog)'.

rec_x int rec_x(Inst *pc)

The executing 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 0 (FALSE) or 1 (TRUE) if there were no errors, or -1 if there was an error.

r_call int r_call()

Performs execution of predicate @x. The ASCII code for x is used as an index into r_vst, and subroutine x is thus called.

r_code int r_code(Inst func)

Compiler for simple operators, i.e., operators fully specified by a single character. 'func' is a pointer to the function which will perform the operation at runtime.

r_colon int r_colon()

Compiler for colons. The name must appear at the proper position in the compilation/execution table (see rectbl.h in this directory).

r_comment int r_comment()

Compiler for comments. Recursively skips over comments built from nested square brackets. The name must appear as the compiling function for '[' in the compilation/ execution table.

r_compile 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.

r_errterm 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.

r_initvst 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 at a pointer containing the address of an error subroutine, r_usub.

r_fin FILE *r_fin

Source input stream; NULL if source is a string.

r_lastchar int r_lastchar

Last character read at a given point during compilation.

r_lbrace void r_lbrace(Inst func)

Compiles a program enclosed in curly braces. The argument 'func' will be the address of r_xbrace, which does the housekeeping functions of entering and exiting a brace-enclosed program.

r_lpar void r_lpar()

Compiles a left parenthesis.

r_noopc void r_noopc()

Compiles a void operator (no code is generated).

r_nxtchar int r_nxtchar(FILE *inp)

Gets the next character from the source input stream.

r_oper1 int r_oper1(Inst func)

Compiles an operator with one ASCII argument. 'func' is the execution address. r_oper1 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.

r_oper2 int r_oper2(Inst func)

Compiles an operator with two ASCII arguments. 'func' is the execution address. 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;
```

leaving 'r_pc' ready for the return.

r_pc Inst *r_pc

The program counter, required by the execution routines of predicates and operators with compiled arguments.

r_pred 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) for a TRUE exit.

r_pred1 int r_pred1(Inst func)

Compiles a predicate with one ASCII argument. '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.

r_pred2 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.

r_quit int r_quit()

Terminates execution through r_errterm. It is usually the execution address for operator underscore (_).

r_rpar int r_rpar()

Compiles a right parenthesis, which always balances a left parenthesis; an unbalanced right parenthesis will go unnoticed, or if the corresponding code in rec.c is enabled, provoke a warning about "Nonblank characters before left delimiter", unless it occurs in a position where it can be taken as the name of a subroutine.

r_semicol int r_semicol()

Compiles a semicolon.

r_symlist Symbol *r_symlist

Head of the symbol table (a linked list managed by routines in sym.c)

r_ubrace int r_ubrace()

Denounces an unbalanced right brace found within a REC expression.

r_ubrack int r_ubrack()

Denounces an unbalanced right bracket found within a REC expression.

r_ungetch int ungetch(int c, FILE *inp)

Returns lookahead character c to the input stream inp.

r_usub int usub()

Function to which all subroutine entries (elements corresponding to printing ASCII characters) in the subroutine table 'vst' are initialized. Its execution will

result in termination with an error message.

r_vst Inst *r_vst[128]

Addresses of subroutines are dynamically loaded and unloaded from this array. Array elements 0-32, 126 (since the right brace may not be used as a subroutine name) and 127 are available for other pointer storage.

r_xbrace int r_xbrace()

Executes a brace-enclosed program.

r_xpred int r_xpred()

Executes a predicate (advances r_pc or not depending on whether the actual function performing the predicate's operation returns TRUE or FALSE).

sym.c Symbol table and memory management routines. Entries in the symbol table are usually generated by quoted string and number operators.

r_install Symbol *r_install(char *s, int len)

Enters string 's' of length 'len' into the symbol table. Returns a pointer to the entry.

r_lookup Symbol *r_lookup(char *s, int len)

Looks up string 's' of length 'len' in the symbol table, returns pointer to entry if found, NULL otherwise.

tty.c Terminal control routines. This module must be compiled with either 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).

r_ttget void r_ttget()

Gets the current terminal settings, stores them in a static structure.

r_ttreset void r_ttreset()

Resets terminal to settings stored by r_ttget.

r_ttset 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.

cnum.c 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).

r_cmin void r_cmin(Inst func)

Provided as an entry point for the compilation of '-' as an operator. 'func' will be the normal execution point for '-', 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 user- provided entry point for retrieving a number from the symbol table).

r_cnum void r_cnum(Inst func)

Compiles a number, which is installed in the symbol table. The pointer to the symbol table will follow the pointer 'func' in the compiled program.

cquo.c Optional module for compiling quoted strings. Both singly and doubly quoted strings are considered, with alternate nesting; backslash-escaped characters as in C are allowed; they are processed by an internal (static) function r_bslash, listed below.

r_dquote void r_quotes(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 user-provided execution function for the double quote character.

r_squote 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++).

r_bslash static int r_bslash(int c)

This function is called by the previous two to allow a C-like mechanism for escaping quotes and non-printable characters. The escape sequences recognized are:

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

ctr.c provides the entry points necessary for compiling and executing counter predicates; these names (r_ctrc and r_ctrx) should appear at the appropriate point in the compilation/execution table (see rectbl.h for an example).

r_ctrc int r_ctrc()

Compiles a counter predicate. The syntax is !n!, where n is a string of decimal digits. The exclamation point is only customary, and any non-digit symbol may be used as delimiter (#n# and n are other likely choices).

r_ctrx int r_ctrx()

Executes a counter predicate. For a given counter !n!, this will be TRUE the first n times and FALSE the (n+1)st; the (n+1)-state cycle repeats indefinitely.

mem.c provides a single entry point, r_malloc(int size), which calls the standard library function malloc and calls r_errterm if malloc returns a null pointer.

A makefile is provided to rebuild the library, librec.a. simply type make.

Subdirectory Sample contains a sample REC compiler. rec-a.c may be used as the basis for a new compiler; the structure and size of the table

defining operators and predicates must be preserved. In particular, the entries for the parentheses, the square brackets, the curly braces, the colon, the semicolon and the atsign MUST be left as they are. It is recommended to keep also the definitions for the underscore (_). Subdirectory Sample also contains a makefile to build an executable file rec-a. rec.h is linked to the directory above.

[Rev. 7.5.91, 26.8.91, 3.10.91, 16.10.91(Unix)]