REC and Convert as aids in teaching Automata Theory*

Gerardo Cisneros† and Harold V. McIntosh
Departamento de Aplicación de Microcomputadoras,
Instituto de Ciencias, Universidad Autónoma de Puebla,
Apdo. Postal 461, 72000 Puebla, Pue., México.
E-mail: CISNEROS@UNAMVM1, MCINTOSH@UNAMVM1

Abstract

The programming languages REC and CONVERT, designed in México two and a half decades ago, have matured in recent years as symbol manipulation languages implemented on a variety of microcomputers. In this paper we present an integrated program package for the study and simulation of several models of automatic computation: regular expressions, push-down automata, Markov algorithms, Turing machines, Post production systems and the LISP programming language. The package was developed for computers based on the Intel 8086 microprocessor family running MS-DOS. It includes a screen editor (written in REC), compilers (written in CONVERT) for the various automaton models, and a command interpreter (also written in CONVERT) which allows iteration of the edit-compile-test cycle for any of the automaton models. The package may be used in Mathematical Logic courses or any others where automata theory is studied.

Introduction

Automata theory is a subject whose inclusion is obligatory in any program of studies in computing with any claim to completeness. Through automata theory one arrives at the concept of an effective computation and at the limits of the effectively computable. A system allowing experimentation with examples of the various automata models that have arisen, affording concrete experience with their operation, may help strengthen one's grasp of the abstract concepts of the theory.

Since automata are mostly symbol-manipulating machines, automata simulation becomes particularly simple if languages designed specifically for symbol manipulation are used. REC and CONVERT, a pair of languages designed by McIntosh almost 25 years ago and implemented in sundry versions (especially REC) by several collaborators [Gu65, Gu66, Se67, McI68, Ci70, Ga71, Va71], have had a recent resurgence as string manipulation languages on microcomputers. [McI85, Ci86a, Ci85, McI89].

In this paper we describe a package of "compilers" written in CONVERT for various automata types, controlled by an interactive command interpreter. The package runs under REC86, the version of REC for the 8086 (and compatible) microprocessors with the MS-DOS operating system. In the following sections we describe the command interpreter, the compiler for parsers of strings generated by regular expressions, the compiler for push-down automata simulators, the compiler for Markov algorithms, the compiler for Post production schemes, the compiler for Turing machine simulators, the LISP compiler, the regular equation system solver and the screen editor. Several of the programs are based on compilers originally written in REC under CP/M[McI84].

The command interpreter

The command interpreter, AUTOMATA.CNV, is the central program in the package. Its main program just saves the PATH variable from the MS-DOS environment, adding a directory AUTLIB, creates buffers for saving the current working file name and directory, and invokes the command selection subroutine. The command line for execution of AUTOMATA may have either of the following forms:

```
REC86 AUTOMATA
REC86 AUTOMATA A ...
```

where A ... represents the names of one or more files (which should include extensions). In the first case, the command selection subroutine displays a list of available commands; in the second, it invokes the screen editor to create or edit each of the files listed in turn.

Once the initial command is executed, the command selection subroutine reads commands from the keyboard, and interprets them according to the following list:

*The original Spanish version, "REC y Convert en la enseñanza de la Teoría de Autómatas", was presented at the Fifth International Conference *Computers in Institutions of Education and Research*, Universidad Nacional Autónoma de México/Unisys, México, November 1989, and appeared in the proceedings of the conference: "Memorias de la Quinta y Sexta Conferencia Internacional *Las Computadoras en las Instituciones de Educación y de Investigación*", 38–40 (1991)

†Member of the Sistema Nacional de Investigadores, México.

`c [file ...]`	compile one or more files
`d [file]`	display list of existing files
`e [file ...]`	edit one or more files
`h [subj]`	help on given subject
`k [file]`	delete file(s)
`l [file ...]`	list file contents
`p [dir]`	set prefix to use
`s`	show current prefix
`t`	terminate the run
`x [file]`	execute the indicated (REC) file
`?`	display this list

Square brackets indicate that the argument or arguments are optional; the ellipsis indicate more than one argument may be included.

Commands `e`, `c` and `x` are the most important; with these the automata are edited, compiled and executed. In particular, if a compilation fails, the editor is invoked and the cursor is positioned near the erroneous character or construct in the source file; on exit from the editor, the compilation is retried automatically. This cycle repeats until compilation produces no more errors or the editor is given the *quit* command. The compiler invoked depends on the file extension of the file to be compiled; `AUTOMATA` recognizes the following extensions:

`RXP`	regular expressions
`PDA`	push-down automata
`MKV`	Markov algorithms
`PST`	Post production schemes
`TNG`	Turing machines
`LSP`	LISP programs
`RSY`	regular equation systems.

If no argument is given on any of the `e`, `c` or `x` commands, the explicit name most recently used in an `e` or `c` command is assumed. In case the most recent `e` or `c` command with explicit arguments had more than one, the working file name will be the last one. This includes those arguments appearing on the MS-DOS command line used to have `REC` execute `AUTOMATA`, since in this case the main program simulates execution of an `e` command.

Command `p` allows the user to specify a subdirectory (with or without a disk identifier) to be attached to the beginning of file names not starting with a backslash (\) or disk identifier; command `s` shows the current prefix. If the current prefix is the null string, the directory current at the start of execution is used.

Commands `k`, `d` and `l` erase files, display directory contents and list files, respectively; their arguments may contain the MS-DOS wildcard characters, asterisk (`*`) and question mark (`?`). They are provided for user convenience. Command `k` does not erase indiscriminately, it asks for confirmation individually for each file whose name matches the given pattern.

Command `h` provides help: without an argument, it displays a list of subjects for which help is available (the available compilers); with an argument, it displays a short description of the compiler corresponding to the given subject. Command `?` displays the list of commands recognized by the interpreter itself.

Finally, the run is terminated by command `t`.

Blanks and comments in source files

The compilers executed by `AUTOMATA` allow inclusion of blanks, tabs and comments in source files. Comments are enclosed in square brackets (`[]`) and may be nested, just as in `REC`.

If there are one or more comments enclosed in double square brackets (`[[...]]`), the string enclosed by the last of these (appearing before the first character belonging to the expression, program or description of the automata proper) will be included in the generated parser or simulator as a sign-on message to be displayed when its execution begins. This string should provide the user with a brief description of the parser or automata and the input it requires. (`RSYCOM`, the regular equation system solver, copies bracket-enclosed comments including the brackets, since it does not generate a `REC` program, but a regular expression which is written on a file with extension `.RXP`.)

Regular Expressions

Regular expressions were invented by Kleene[Kl56] as a language with which to describe the strings recognized by the neural nets of McCulloch and Pitts[McC43]. A series of fundamental theorems in the theory of finite automata establishes the equivalence of regular expressions, (deterministic) finite automata and transition systems (nondeterministic finite automata)[Ho79].

`AUTOMATA` executes the `RXPCOM` compiler to transform a regular expression contained in a file whose extension is `.RXP` into a `REC` program which accepts strings read from the keyboard if they belong to the language denoted by the regular expression or rejects them otherwise.

Regular expressions accepted by this compiler may include any printing Ascii characters [between the exclamation point (`!`) and the tilde (`~`)], all of them taken as alphabet letters except for the following characters, which have the indicated meaning:

`#`	the empty set
`$`	the null string
`()`	grouping symbols
`|`	alternation operator
`*`	closure operator

Spaces and tabs may be used freely, and bracket-enclosed comments may appear at the beginning of the

file. The first non-blank, non-left-bracket character will mark the beginning of the regular expression for which a parser is to be generated; the expression is taken to include all non-blank characters up to the end of the file.

For example, the following lines

```
[ZAT3RD.RXP]
[[Binary strings w/3rd-from-last digit = 0]]

      (0 | 1)* 0 (0 | 1) (0 | 1)
```

may be placed in a file `ZAT3RD.RXP` for which `RXPCOM` will generate a file `ZAT3RD.REC`, which when executed will read strings and determine whether or not they are binary numbers whose third-from-last digit is a 0.

Push-down Automata

Push-down automata are the simplest automata able to recognize languages generated by context-free grammars. They are machines lying between finite automata and Turing machines. As opposed to finite automata, a non-deterministic push-down automaton (PDA) is not in general equivalent to a deterministic PDA; the class of languages accepted by deterministic PDAs is more restricted than that of languages accepted by non-deterministic PDAs. The book by Hopcroft and Ullman [Ho79] contains an extensive discussion on automata and languages, with numerous references to the original papers.

Given a file (whose extension is `PDA`) containing the description of a push-down automaton, `AUTOMATA` calls for the execution of `PDACOM`, which generates a simulator for the automaton.

A push-down automaton may be described by a set of n-tuples, with n odd and greater than or equal to 5. Each n-tuple is written as

$$(q,s,S,q_1,T_1,\ldots,q_k,T_k)$$

where q is a state, s is the null string or a symbol in the input alphabet, S is a symbol in the stack alphabet, the q_i are states to which transitions are possible on reading s from the input with S on the top of the stack, and the T_i are strings (formed by stack alphabet symbols, and possibly null) replacing S if a transition to the corresponding state q_i occurs.

The T_i strings are pushed on the stack from left to right, so that the rightmost character becomes the stack top.

State names may be any combination of digits and letters; stack and input symbols are any Ascii characters between the space (sp) and the tilde (~), except the round parentheses, the comma, the apostrophe and the double quote. The q state in the very first tuple is taken as the start state; the stack symbol S in the same first tuple is taken to be the initial stack contents.

States whose name starts with an `F` are deemed to be final states; if there are none, the automaton is assumed to accept strings by empty stack. For a given state q and stack symbol S, the tuple (if any) for which s is the null string should appear before the tuples in which s is nonnull.

The following is an acceptable description for a push-down automaton accepting even-length palindromes in binary[Ho79].

```
[MIRROR.PDA]
[[
This automaton accepts even-length
palindromes over the alphabet {0,1}.
]]

(q1,,R,q2,)             (q2,,R,q2,)
(q1,0,R,q1,RB)          (q1,1,R,q1,RG)
(q1,0,B,q1,BB,q2,)      (q1,0,G,q1,GB)
(q1,1,B,q1,BG)          (q1,1,G,q1,GG,q2,)
(q2,0,B,q2,)            (q2,1,G,q2,)
```

In accordance with the foregoing description, the start state is `q1`, and the initial stack symbol is `R`. The automaton is nondeterministic, both because there are (q,s,S) triples for which more than one transition is posible, and because there are q and S for which there are transitions from (q,s,S) with null and nonnull s.

The programs generated by `PDACOM` show at each step of their execution in a single line the contents of the stack, the top of the stack, the unread portion of the input string, and the state of the automaton. The input string may seem to grow when nondeterministic automata have to backtrack to consider some alternate transition.

Markov Algorithms

Markov algorithms, originally called by their author "normal algorithms", are the result of the attempt by the Russian mathematician A. A. Markov to make a rigorous algorithmic process out of the production schemes of E. L. Post. In a Post system, one is required to exhibit a sequence of transformation steps, or to prove that no such sequence can be found. This is a process which may depend upon a considerable amount of ingenuity, or involve non-constructive reasoning, which is unsatisfactory from the algorithmic point of view.

A Markov algorithm consists of a sequence of pairs—antecedents and consequents—which constitute transformation rules to be applied to a string. They are to be tried in the strict priority of the order in which they are listed, and the current rule is applied if the string under transformation contains the antecedent as a substring; that being the case, the substring found (the first instance in a left to right search) is replaced by the consequent. Furthermore, each rule is marked according to whether

the process is to be terminated fortwith or to be repeated again from the beginning on the transformed string. If no rule applies, the process also stops. Thus, a quite definite sequence of events is described, with which to effect a calculation.

Markov algorithms can be modelled in REC with especial ease just because they were taken as the model for REC's workspace and its associated operators and predicates[Ci86b].

Markov's own book[Ma54], which spells out his scheme in minute detail, is still the best source to consult for a description of these algorithms. Paralleling the development of the theory of Turing machines, Markov's treatment concludes in the enunciation of a Universal Algorithm.

Translation of a Markov algorithm into a file (with extension `MKV`) usable by `AUTOMATA` is straightforward. The rules may have either of two forms:

$(\alpha:\beta)$	if the rule is repetitive,
$(\alpha;\beta)$	if the rule is terminal.

Both the antecedent (α) and the consequent (β) may be strings of zero or more ASCII characters between the space (SP) and the tilde (~), but not including round parentheses, colon, semicolon, apostrophe or double quotes.

The following example shows the contents of a file describing a Markov algorithm to sum two binary numbers. The rules are ordered left to right and top to bottom, although it is more usual to list them one to a line.

```
[BSUM.MKV]
[[A Markov algorithm which will sum two
binary numbers presented in the form a+b=;
for example

                111110+011=

Each step shows an application of the list
of rules until the final sum is completed
and the auxiliary symbols are gone.]]

(a0:0a)          (a1:1a)          (b0:0b)
(b1:1b)          (0*:1)           (1*:*0)
(*:1)            (0a:=0)          (0b:=1)
(1a:=1)          (1b:*=0)         (0+:+a)
(1+:+b)          (+a:0=+)         (+b:1=+)
(+:)             (=:)
```

Each step in the execution of a program produced by `MKVCOM` shows the result of the last transformation applied to the given string. The generated programs provide the means for interrupting very long or infinite loops, which are always described in the sign-on message.

Post Productions

Post productions strongly resemble the proofs that have traditionally characterized high school courses in Euclidean geometry. Some initial premise—an axiom—is to be transformed by a chain of substitutions into a conclusion—a theorem. A distinguishing feature of Post's rules of transformation is that the text, or phrases to be manipulated, are not spelled out explicitly. Rather, they are defined by their context, that is, by the phrases immediately preceding or following them.

Post's ideas are to be found in a couple of journal articles[Po43, Po46]. A really excellent presentation of his work occupies a great part of Minsky's book[Mi67], derived from his own lecture notes at MIT.

It is possible to define a system, which does not strictly follow Post's original plan, but which is nevertheless algorithmic and which illustrates many of the ideas and captures much of the spirit behind his formalization of computation. Such a program was a precursor to CONVERT. Yet a different approach to a computational scheme led A. A. Markov to his algorithms.

A set of Post productions is a list of antecedent-consequent pairs which are placed in a file with extension `PST` and whose general form is

$$(g_0 v_1 g_1 \cdots v_n g_n, h_0 u_1 h_1 \cdots u_m h_m)$$

where every v_i and u_j denote a variable and each g_l and h_k denote a constant string. Each variable is denoted by the symbol <*p*>, where p is an integer between 0 and 28.

In the antecedent, no variable may appear more than once and no inner g_i may be null; in the consequent only variables appearing in the antecedent may be used. The constant strings may contain any ASCII characters between the space (SP) and the tilde (~), except round parentheses, angle brackets comma, apostrophe or double quotes. g_0 and g_n may be null, as may be any of the h_k in the consequent. Only one attempt is made to bind each of the variables (using the shortest possible string) and a rule fails if any of the constant strings in the antecedent is not found at the appropriate place. If g_n is null the last variable (v_n) is bound to the remaining suffix of the text being transformed; otherwise the text must have g_n as a suffix. Rules are tried in order (Markov-style) and the process is reapplied to the transformed string; it terminates only when no rule applies or the user interrupts it by pressing `Ctrl-X`.

What follows is a set of productions which, like the example in the previous section, perform the sum of two binary numbers.

```
[BINSU.PST]
[Post productions can deal with the digits
to be summed in a single step even though
they are widely separated.  In a Markov
algorithm it is necessary to move one of
```

```
them until it is in contact with the other
before the addition table can be applied.]

[[
A Post Production scheme which will sum two
binary numbers written in the form a+b=;
for example:
              111110+011=
]]

       (<1>0+<2>0=<3>,<1>+<2>=0<3>)
       (<1>0+<2>1=<3>,<1>+<2>=1<3>)
       (<1>1+<2>0=<3>,<1>+<2>=1<3>)
       (<1>1+<2>1=<3>,<1>+<2>*0<3>)
       (<1>0+<2>0*<3>,<1>+<2>=1<3>)
       (<1>0+<2>1*<3>,<1>+<2>*0<3>)
       (<1>1+<2>0*<3>,<1>+<2>*0<3>)
       (<1>1+<2>1*<3>,<1>+<2>*1<3>)
       (<1>+<2>=<3>,<1><2><3>)
       (+<2>*<3>,0+<2>*<3>)
       (<0>+*<3>,<0>+0*<3>)
       (+*<3>,1<3>)
```

During execution, the program shows the string resulting from each applied transformation.

Turing Machines

A Turing machine is a fundamental concept in the theory of computation. Although of little practical use, it forms a model of computation which is so simple and definite that theorems can be proven about its operation allowing rigorous conclusions to be drawn about what is computable and what is not. It is Church's thesis that anything which is computable is computable by a Turing machine; a statement which cannot be proven but which has never been refuted after very extensive analyses of what might be meant by computation and ways of performing it. (According to Minsky, the thesis, in terms of his machine model, is Turing's; Church was interested in the general character of an effective computation.)

One of the best references to Turing machines and their operation is again Minsky's book[Mi67]; Hopcroft and Ullman[Ho79] give demonstrations of the equivalence of several modifications to the basic machines (multiple tapes, multidimensional tapes, semiinfinite tapes, etc.). The original article, which is often cited for the definition of a Turing machine, is [Tu36].

A Turing machine is described by a set of quintuples. Each quintuple has the following form:

$$(q,s,s',q',d)$$

indicating that if the machine is in state q and its read/write head is placed over a square containing symbol s, it is to replace s by s', make a transition to state q' and move the head in the direction indicated by d.

States q and q' may be denoted with any combination of digits and letters; as symbols s and s' any ASCII character between the space (SP) and the tilde (~) may be used, except the round parentheses, the comma, the apostrophe, the double quote and the circumflex accent (^). The direction d may be a `+` (to move the read/write head to the right), a `-` (to move it to the left), or `0` (to stay on the same square). The state name `H` is distinguished as the Halt state; a machine also halts if it finds itself with a state-symbol pair (q, s) for which no quintuple is available.

Having placed a set of quintuples in a file with extension `TNG`, the first state of the first quintuple is taken to be the start state of the machine. At the start of execution, a circumflex accent in the initial tape indicates the head is to be positioned at the character following it; if no accent appears, the head will be assumed to be positioned on the leftmost character of the given string. Empty tape squares are assumed to contain blanks (SP).

Once more consider the binary sum example:

```
[BADD.TNG]
[[
This Turing Machine will sum two binary
numbers. Its initial tape should have the
form =a+b=, with the head positioned at
the right equal sign. For example,

              =111110+011^=
]]

  (Q0,=, ,dig,-)
  (dig,0,=,zle,-)      (dig,1,=,ole,-)
  (dig,+,+,lef,-)      (zle,0,0,zle,-)
  (zle,1,1,zle,-)      (zle,+,+,zad,-)
  (ole,0,0,ole,-)      (ole,1,1,ole,-)
  (ole,+,+,oad,-)      (zad,a,a,zad,-)
  (zad,b,b,zad,-)      (zad,0,a,rig,+)
  (zad,1,b,rig,+)      (zad,=,a,new,-)
  (oad,a,a,oad,-)      (oad,b,b,oad,-)
  (oad,0,b,rig,+)      (oad,1,a,car,-)
  (oad,=,b,new,-)      (car,0,1,rig,+)
  (car,1,0,car,-)      (car,=,1,new,-)
  (new, ,=,rig,+)      (rig,0,0,rig,+)
  (rig,1,1,rig,+)      (rig,a,a,rig,+)
  (rig,b,b,rig,+)      (rig,+,+,rig,+)
  (rig,=, ,dig,-)      (lef,0,0,lef,-)
  (lef,1,1,lef,-)      (lef,a,a,lef,-)
  (lef,b,b,lef,-)      (lef,=, ,fin,+)
  (fin,0,0,fin,+)      (fin,1,1,fin,+)
  (fin,a,0,fin,+)      (fin,b,1,fin,+)
  (fin,+, ,H,+)
```

During the simulation of a Turing machine, the generated program shows the a section of the tape centered

at the square under the head (which is distinguished by separating it from the rest by a blank on either side), and the state of the finite control to the right of the tape section.

LISP

LISP is an invention of John McCarthy's dating from the late 1950's at MIT. LISP is based on Alonzo Church's lambda-calculus, and is fundamentally recursive in nature. It primarily manipulates text, which leads to the idea of defining it in its own notation, after the fashion of a Universal Turing Machine. The resulting Universal function EVAL gives a self-consistent definition of LISP within LISP.

Unfortunately, due to its low efficiency of execution and the predisosition of users for iterative programming, pure recursion proved to be very cumbersome for LISP. To meet these objections, the "program feature" was introduced. The development of `REC` was a direct consequence of the desire to design an aesthetic substitute for the "program feature", having evolved from the related concept of "operator predicates".

The fundamental reference to LISP is [McC60]; for a long time the only other reference was the user's manual, [McC62]. Nowadays, of course, there are quite a few books on LISP.

An important aspect of LISP 1.5 and its successors as implemented at MIT and elsewhere was the physical representation chosen for lists—a binary tree with pointers to list elements and eventually constants such as atoms and numbers. The translation generated by `LSPCOM` ignores this aspect entirely, so that lists are strings whose parentheses are balanced anew with each operation that is carried out. This makes no difference whatsoever in the linguistic aspects of the program, but of course renders it entirely impractical for the execution of truly large programs. To preserve the full aspects of LISP, it would only be necessary to make a version of the built-in primitive functions which would use binary trees rather than chains.

The program implemented here is EVAL, not EVALQUOTE nor APPLY, as defined in [McC62], nor some other variant on the theme. This means that one would present, for instance, `(cons (quote a) (quote (1 2 3)))` rather than the version `cons (a (1 2 3))` that EVALQUOTE would expect, or the additional ALIST which APPLY would require.

The version of LISP recognized by `AUTOMATA` has the following limitations:

- The file must consist of a set of function definitions, the last of which must be called `*`, and which provides the main entry point.
- User-defined function names may only be single upper- or lower- case letters.
- Atoms representing variables (to be associated by `lambda`) may only be integers between 0 and 28.

The following predicates, functions and forms are recognized:

append	and	atom	bl	caar	cadr
car	cdar	cddr	cdr	cond	cons
cr	eq	if	lambda	lf	list
lp	not	null	or	print	qu
quote	read	rp	tb		

Each function has the form

$$(n\ \ f)$$

where n is the name (a letter or `*`), and f is the form used to compute the function. In the form `((lambda` v f`)` u`)`, v is the list of variables (zero or more parenthesis-enclosed, blank-separated integers between 0 and 28), f is the form defined and u is the list of objects to be associated to the variables in list v.

At the start of execution, the generated program prompts for a string, which the main program will be able to use only if it has been defined in terms of a `lambda` with *a single* variable (as in the example below).

Consider yet again the symbolic sum of binary numbers:

```
[BINADD.LSP]
[[
Binary sum in LISP:  When "lisp>" appears,
type in the first number as a list of ones
and zeros; when "read>" appears, type the
second number in likewise fashion.
]]

[calculate a binary sum -
 use reversed digits]
(b (lambda (0 1)
           (r (c (r 0 (list)) (r 1 (list)))
                    (list)) ))

[reverse a list]
(r (lambda (0 1) (if
    (eq 0 (quote ())) 1
    (r (cdr 0) (cons (car 0) 1))) ))

[sum low order bits, then rest]
(c (lambda (0 1) (cond
    ((null 0) 1)
    ((null 1) 0)
    ((and) (cons (d (car 0) (car 1))
               (c (list (e (car 0) (car 1)))
                        (c (cdr 0) (cdr 1)))
```

```
    ))
   )))

[sum of two bits]
(d (lambda (0 1) (cond
    ((eq 0 (quote 0)) 1)
    ((eq 1 (quote 0)) 0)
    ((and) (quote 0))
    )))

[carry bit]
(e (lambda (0 1) (cond
    ((eq 0 (quote 0)) (quote 0))
    ((eq 1 (quote 0)) (quote 0))
    ((and) (quote 1))
    )))

[main program]
(* (lambda (0) (b 0 (read))))
```

Regular Equations

A deterministic finite automaton may be expressed as a system of "linear" equations in which each state is represented by an unknown, the coefficients are the characters on which transitions occur and "multiplication" and "addition" are the concatenation and alternation operators of regular expressions, respectively.

For example, if an automaton has three states A, B and C, such that state A has transitions to state B on symbol `1`, and to state C on symbol `0`, the equation for A will include terms `1`B and `0`C. The equation for A will also include constant terms for each transition ending in a final state (`1` if B is final, `0` if C is final, the null string, denoted by `$`, if A itself is final). The equation for the initial state heads the list. For simplicity, the alphabet is limited to $\{0, 1\}$; states are denoted by single upper- or lowercase letters.

The system is solved by backward substitution, using Arden's lemma[Ar60, Ba75]: If α and β are regular expressions such that the language denoted by α does not contain the null string, then the solution to

$$S = \alpha S \mid \beta$$

is $S = \alpha^* \beta$.

The solution produced is the regular expression for the state whose equation appears first in the file (whose extension must be `RSY`); if that state is the start state of the automaton, the resulting regular expression describes the language accepted by the automaton. This expression is written in a file with extension `RXP`, which may in turn be compiled to generate a parser in `REC`.

Due to the growth in complexity of the expressions (which tends to be exponential) as the backward substitution procedes, this solver is practical only for automata with a few states (up to seven or eight).

The following is an example of the sort of data expected by `AUTOMATA` in a `RSY` file. Each equation must appear on its own line, with no embedded blanks.

```
[EJEM.RSY]
[[
Automaton from exercise 2.13(a)
from the book by Hopcroft & Ullman
]]

a=0a|1b|0|$
b=0c|1b
c=0a|1b|0
```

The editor

The screen aditor, based on services provided by the BIOS of the IBM PC and compatibles through the INT 16 instruction, includes the following commands:

Key	*Effect*
`[Ctrl-Q]`	Quit edit (no changes saved)
`[Ctrl-E]`	End edit (save changes)
`[↑]`	Cursor up
`[→]`	Cursor right
`[←]`	Cursor left
`[↓]`	Cursor down
`[Page Up]`	Advance a page
`[Page Down]`	Back up one page
`[Home]`	To beginning of file
`[End]`	To the end of the file
`[F3]`	To beginning of line
`[F4]`	To end of the line
`[Delete]`	Delete character under cursor
`[Backspace]`	Delete character preceding cursor
`[Ctrl-Z]`	Delete from cursor to end of line
`[Ctrl-X]`	Delete line
`[F1]`	Search for a string
`[Alt-F1]`	Cancel search string
`[F2]`	Search and replace
`[Alt-F2]`	Cancel search-and-replace strings
`[Insert]`	Toggle text addition modes (insert/overwrite)

Conclusions

The `REC` programs produced by the compilers in the package are not meant to be efficient; rather the emphasis is upon their pedagogical value, in that the different automaton types are readily programmable and modifiable, and in the ease and clarity of their presentation of the working of the corresponding automaton. During the simulation of a given automaton, it is possible to slow

down the speed of the simulation, to examine in detail its moves, or to speed it up if one wants to skip quickly over a sequence of steps.

The use of CONVERT, with its paradigmatic style based on transformation rules, allowed writing the compilers in a relatively short time; several subroutines were reused across the set of compilers. The virtual machine afforded by `REC`, with its stack, workspace and table of variables, simplified the task of translating the different automaton types.

The screen editor, although adequate for the package's purposes, could be extended; for example, commands to copy and move blocks of text or to move the cursor by words or paragraphs might be added.

The programs fit in a single 5.25" DSDD floppy disk, including the `REC` compiler, the source programs in CONVERT, the CONVERT compiler itself, at least one sample source file for each of the compilers, and the TEX source for this paper.

References

[Ar60] D. N. Arden, "Delayed logic and finite state machines," *Theory of Computing Machine Design*, pp. 1–35, Univ. of Michigan Press, Ann Arbor, Mich. (1960)

[Ba75] R. C. Backhouse and B. A. Carré, "Regular Algebra Applied to Path-finding Problems," *Journal of the Institute for Mathematics and its Applications* **15**, 161–186 (1975)

[Ci70] 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)

[Ci85] G. Cisneros y H. V. McIntosh, "Introducción al lenguaje de programación Convert," *Acta Mex. Cienc. Tecnol.* **3**, 9 (Ene-Mar), 65–74 (1985). English translation published as "Introduction to the programming language Convert", *SIGPLAN Notices* **21**, 4 (Apr), 48–57 (1986)

[Ci86a] G. Cisneros and H. V. McIntosh, REC and Convert compilers for MS-DOS, PC/Blue library, Vols. 211 and 212, New York Amateur Computer Club (1986); also available as #MS29 from Micro Cornucopia.

[Ci86b] G. Cisneros y H. V. McIntosh, "Notas sobre los lenguajes REC y Convert", Departamento de Aplicación de Microcomputadoras, Instituto de Ciencias, UAP, Puebla, México (1986)

[Ga71] R. Carlos García Jurado M., "Un REC visual para la PDP-15 en comunicación con la PDP-10," B.Sc. Thesis, ESFM, Instituto Politécnico Nacional, México (1971)

[Gu65] A. Guzmán Arenas, "CONVERT," B.S.E.E. Thesis, ESIME, Instituto Politécnico Nacional, México (1965)

[Gu66] A. Guzmán and H. V. McIntosh, "CONVERT," *Commun. ACM* **9**, 8 (Aug), 604–615 (1966)

[Ho79] J. E. Hopcroft and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation," Addison-Wesley, Reading, Mass. (1979)

[Kl56] S. C. Kleene, "Representation of events in nerve nets and finite automata," *Automata Studies (Annals of Mathematics Studies, No. 34)*, Princeton University Press, Princeton, N.J. (1956)

[Ma54] A. A. Markov, "Theory of Algorithms," Works of the Mathematical Institute "V. A. Steklov", Vol. 42, Academy of Sciences of the U.S.S.R., Moscow (1954) [English translation: The Israel Program for Scientific Translations; U.S. National Technical Information Service document No. TT60-51085]

[McC60] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine," *Commun. ACM* **3**, 4 (Apr), 184–195 (1960)

[McC62] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, "LISP 1.5 Programmer's Manual," The MIT Press, Cambridge, Mass. (1962)

[McC43] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bull. Math. Biophysics* **5**, 115–133 (1943)

[McI68] H. V. McIntosh, "A CONVERT compiler of REC for the PDP-8," *Acta Mex. Cienc. Tecnol.* **2**, 1 (Ene-Abr), 33–43 (1968)

[McI84] H. V. McIntosh, "REC applications including CNVRT compiler," SIG/M library, Vol. 166, Amateur Computer Group of New Jersey (1984)

[McI85] H. V. McIntosh and G. Cisneros, "REC and Convert compilers for CP/M," SIG/M library, Vols. 213, 214 and 215, Amateur Computer Group of New Jersey (1985)

[McI89] H. V. McIntosh and G. Cisneros, "The programming languages REC and Convert," *SIGPLAN Notices* **25**, 7 (July), 81–94 (1990)

[Mi67] M. Minsky, "Computation: Finite and Infinite Machines," Prentice-Hall, Englewood Cliffs, N.J. (1967)

[Po43] E. L. Post, "Formal reductions of the general combinatorial decision problem," *Amer. J. Math.* **65**, 197–268 (1943)

[Po46] E. L. Post, "A variant of a recursively unsolvable problem," *Bull. Amer. Math. Soc.* **52**, 264–268 (1946)

[Se67] Raymundo Segovia Navarro, "CONVERT en el diseño de procesadores," B.S.E.E. Thesis, ESIME, Instituto Politécnico Nacional, México (1967)

[Tu36] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proc. London Math. Soc. (Ser. 2)* **42**, 230–265 (1936)

[Va71] José Luis Varas Araujo, "Compilador REC en lenguaje COMPASS para la computadora CDC-6400," B.Sc. Thesis, ESFM, Instituto Politécnico Nacional, México (1971)