MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence
Memo. No. 149.

January 1968.

REC/8

A CONVERT COMPILER OF REC FOR THE PDP-8

Harold V. McIntosh[*]

* ESCUELA SUPERIOR DE FISICA Y MATEMATICAS
INSTITUTO POLITECNICO NACIONAL
MEXICO 14 D.F., MEXICO.

ABSTRACT

REC/8 is a CONVERT program, realized in the CTSS LISP of Project MAC, for compiling REC expressions into the machine language of the PDP-8 computer. Since the compilation consists in its majority of subroutine calls (to be compiled, after removal of LISP parentheses by MACRO-8) the technique is applicable with trivial modification to any other computer having the subroutine jump and indirect transfer instructions. The purpose of the program is both to compile REC expressions and to illustrate the workings of the REC language, and accordingly a description of this language is given. It contains operators and predicates; flow of control is achieved by parentheses which define subexpressions, colon which implies iteration, and semicolon which terminates the execution of an expression. Predicates pass control to the position following the next colon or semicolon, allowing the execution of alternative expression strings.

REC (REGULAR EXPRESSION COMPILER) is a programming language of simple structure developed originally for the PDP-8 computer, but readily adaptable to any other general purpose computer. It has been used extensively in teaching Algebra and Numerical Analysis in the ESFM, even for programming hand calculations with the Friden electronic desk calculator. In rather vague terms, it derives its appeal from the fact that computers can be regarded in one way or another as Turing Machines with very elaborate built-in shortcuts to eliminate the grotesque inefficiency of manipulating individual bits on a single linear tape. A turing Machine consists of a finite state machine acting as the control of a tape memory; finite state machines in turn are conveniently described by regular expressions. The REC notation is a manner of writing regular expressions mor amenable to programming the Turing Machine which they control. If one does not wish to think so strictly in terms of Turing Machines, REC expressions still provide a means of defining the flow of control in a program, which is quite convenient in many applications.

Let Σ be an alphabet, which presumably would not contain among its letters the operational signs which we shall introduce. We then define a REC expression recursively in the following manner.

i) λ is a REC expression

ii) () is a REC expression

iii) if $\sigma \epsilon \Sigma \cup \{: \; ; \; \circ\}$, σ is a REC expression

iv) if α and β are REC expressions, so is $\alpha\beta$

v) if α is a REC expression, so is (α)

The operational signs are used as follows. Parentheses are used to denote a single expression. Concatination is implied by writing expressions in sequence. Colon [:] implies iteration of all the expression which precedes the colon. Semicolon [;] terminates the concatination of a string. The large period [∘] indicates a choice between continuing to concatinate the following expressions or to pass over them until the next following colon or semicolon (if any) of the same parenthesis level is reached. Such a choice is always implied following a parenthesized expression.

It is to be noted that parentheses have a very technical use in REC expressions, and are more than simple signs of grouping. Thus, since concatination is associative, it is always written in its extended form without parentheses. When some grouping is desired to be shown, some other symbol, such as square brackets, should be used. The non-associativity of REC parenthesization is often exploited to achieve some economy or simplification of expression.

To see the correspondence between regular expressions and REC expressions, we first show how any regular expression is to be written as a REC expression.

$$\Phi \to ()$$

$$\lambda \to \lambda$$

$$\sigma \to \sigma$$

$$\alpha\beta \to \alpha\beta$$

$$\alpha \cup \beta \to (\bullet\alpha;\beta;)$$

$$\alpha^* \to (\bullet\alpha:;)$$

For the converse process of writing the regular expression corresponding to a REC expression, it is more convenient to show how to use a REC expression to construct a transition system, whose regular expression (or class of equal regular expressions) may then be deduced. The algorithm is as follows, recursively defined.

1) For every REC expression there will be an initial state and two final states, labelled T and F.

2) A REC expression is to be read from left to right, but any quantity appearing within parentheses is to be treated as a single expression, recursively, with additional rules governing how to join its initial and final states to those outside the parentheses.

3) If $\sigma \in \Sigma$ is seen, draw an arrow labelled σ from the last state to a new state.

4) If λ is seen, do the same with a spontaneous transition.

5) if : is seen, draw an arrow representing a spontaneous transition back to the initial state.

6) If ; is seen, draw an arrow representing a spontaneous transition to the final state T.

7) Whenever either : or ; is seen, a new state should be formed.

8) If $\bullet$ is seen, draw an arrow representing a spontaneous transition to the state immediately following the next : or ;, if any; otherwise to the final state T.

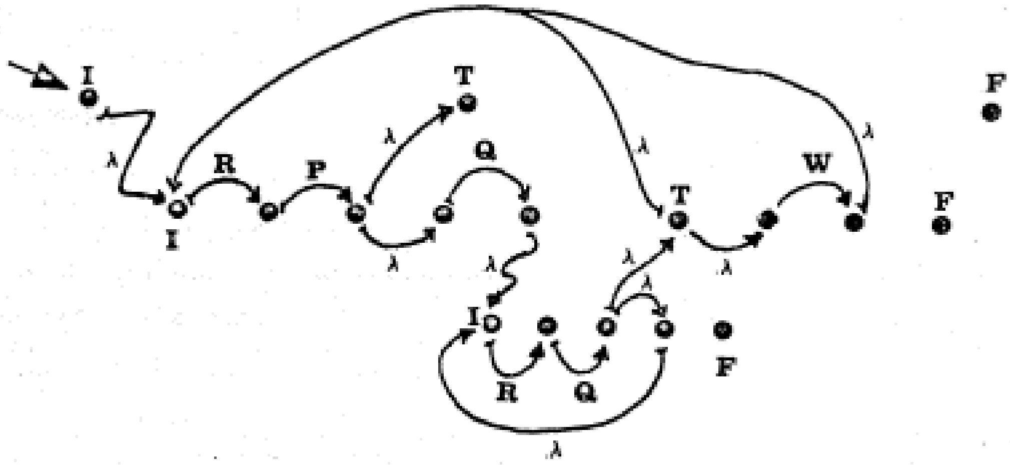
9) If a parenthesized expression is seen, apply the entire algorithm to the parenthesized expression. When this is done, draw an arrow representing a spontaneous transition from the last state to the initial state of the enclosed expression. The last state written is the final state F of the enclosed expression, and is to be connected by a spontaneous transition to a subsequent state in exactly the same manner as had an ∘ been seen. The final state T is to be used as the current state in continuing to process the outer level.

10) The last state to be written is the final state F of the outer expression; or should be connected to it by a spontaneous transtion if the latter has already been prepared. The final state T is the accepting state of the transition system.

As an example of the process, let us consider the transition system which we would produce from the REC expression

(R P∘ ; Q∘ (R Q∘::) :W:)

by following the above rules.



It will be seen that the three F states are all isolated because according to this particular REC expression there is no way to arrive at any of them.

It will be noted that the REC expressions which are derived from regular expressions by the prescription we have offered form a limited class among the possible REC expressions. In part this is due to a desire to leave the semantics of the REC expressions relatively weak, even though it admits a great number of expressions which would produce useless transition diagrams; for instance we do not exclude the sequence :::::.

But there is also the consideration that although regular expressions are defined with binary connectives, such as union and concatination, these connectives are associative and are generally written in parenthesis free form. Although REC parentheses are not associative, there nevertheless exist convenient n-ary forms equivalent to their corresponding binary forms. For example, in a triple union one could write

(A u B) u C as (∘(∘A;B;);C;)
A u (B u C) as (∘A;(∘B;C;);)

but preferable to both is writing

A u B u C as (∘A;∘B;C)

with a similar notation corresponding to a more extensive union. In fact, such merit as there might be to the REC notation arises from the fact that although it might be somewhat cumbersome to make direct transcriptions of regular expressions, there will be a consequential class of expressions which we will wish to write whose REC form will be simpler and more convenient than the corresponding regular expressions. Thus the correspondences which we have established serves to demonstrate that the totality of REC expressions is no more nor less general than the totality of regular expressions.

Since the intention of REC expressions is to control the operation of a general purpose computer (or more specifically a Turing Machine), we will expect the letters of the REC alphabet to represent individual operations of which the machine is capable. For this reason the letters will be called <u>operators</u>. Words of the REC alphabet will then correspond to sequences of operations, carried out in the order given. The transition system derived from a REC expression will then accept a word of this alphabet if it corresponds to a possible series of operations which could be carried out during the calculation in question. In the case of a Turing Machine, the operators will be to write a symbol, compare a symbol, move the tape left or move the tape right. But the operators will have to be chosen according to the circumstances.

In reality we are not so much interested in recognizing a possible calculation as in prescribing the particular one which we want among all those possible. It is for this purpose that the large period [∘] was introduced, which is related to the operation of union in a regular expression. At each place in a REC expression where ∘ occurs, there is a spontaneous transition in the transition diagram. indicating the possibility of a selection among two alternatives; to continue the regular sequence, or to start a new one by following the spontaneous transition. To specify a particular word among all those represented by a given REC expression, it is only necessary to specify this choice at each place where it becomes possible. We might even assume that there are special operators whose purpose is to make this choice. They are called *predicates*, and will always combine the symbol . implicitly. Thus a predicate is a combination of an appropriate operator followed by the symbol .. We will moreovre say that a predicate takes the value *true* or *false* according to whether the decision is made to continue in the regular sequence or to follow the spontaneous transition pas the nearest colon or semicolon. Every parenthesized expression is automatically assumed to be a predicate, although analysis may show that it is only capable of assuming one of the two possible values. Such was the case in our example.

The transition diagrams of REC expressions have two final states to accomodate their usage as predicates. Thus a calculation definitely fails, definitely succeeds, or else is in progress. Moreover the REC notation has been particularly chosen to facilitate the formation of Boolean combinations of its subexpressions. Thus the combination AND of the predicates a, b, c, ..., n is written

(abc ... n;),

a notation which is valid for any number of arguments. Thus (;) always is a true predicate, whilst a = (a;).

The combination OR of these same predicates would be written

(a; b; c; ...; n;),

which again holds for any number of arguments. () is a predicate which is always false, and as before, (a;) = a.

The complement of the predicate x is written

(x).

We accordingly always have x = ((x)).

A typical REC expression will begin with a series of operators, followed by a predicate which will decide typical questions such as whether the calculation is finished and be followed by ;, or whether to repeat the whole procedure and be followed by :. When these conditions fail, there will follow further calculation, expressed by a series of operators, and yet another predicate. One executes as much of a string as he can until he meets a delimiter, and as many strings as necessary to meet a terminal condition. One practical caution which has to be observed is that if several predicates occur in a string, and one has reached the end of the string, the AND of all these predicates is true. if one arrives beyond a colon or semicolon, indicating the string has failed, he only knows the AND has failed, but not which individual predicate. This requires either a new test of some of the predicates, or a more cautious rewriting of the REC expression. It is one situation in which one sometimes wishes there were a more direct control of the flow of control in a REC expression; perhaps by means of labels and "GO TO's."

To give some very simple examples of the application of REC, let us bear in mind the PDP-8 computer, which has a teletype coded for 64 ASCII characters in direct communication with the central processor. Let R be the operator which reads one such character, either from paper tape or punched by hand on the keyboard, and W be the operator which sends one such character to the teletype. The characters are kept in a workspace (the accumulator, say), and we may imagine 64 operators of the type "x which place the character x in this workspace erasing the previous contents, as well as 64 predicates =x which test the workspace for equality to the character x.

The REC expression

```
(R =!; W " W:)
```

will doublespace everything which it reads, until the exclamation point is encountered and it terminates operation.

Let us say that we wish to ignore all text which occurs between two stars. An appropriate expression will be

```
(R =!; =* (R =*;:) :W:)
```

and again it will terminate when an exclamation point is encountered in the printing text.

By including operators for the binary conversion of decimal input and output, the arithmetic operations, and a test for negative numbers, one could formulate REC expressions for arithmetic calculations. The domain of applicability of REC depends upon its complement of operators and predicates; however at present it is only the control structure which interests us.

Although we are describing a compiler of REC for the PDP-8, the description is applicable to the majority of machines because the compilation is made entirely in terms of subroutine calls, except for the part which corresponds to REC's own flow of control, which is realized for the most part by appropriate transfers.

In the PDP-8, a subroutine call is made by means of the instruction JMS (Jump to Subroutine). Let us suppose we have the coding configuration

```
X,   JMS Y

     ...
     ...

Y,   00
     ...
     JMP I Y
```

When the instruction JMS Y, located at address X, is executed, the address X+1 is stored at Y, and transver is made to Y+1. When the subroutine is terminated, this is done by the instruction JMP I Y, an indirect transfer to Y which is a transfer to X+1, so that the original program is resumed in sequence.

Data of use to the subroutine Y may be located at addresses X+1, X+2, and so on, and may be accessed indirectly through the address stored at location Y. By applying the instruction ISZ Y, (Increment and skip on zero), this data may be gathered item by item. Moreover, the subroutine Y can serve as a predicate, since an ISZ preceding the return jump can cause a skip to X+2 rather than a return to X+1.

In this way, the predicate, =x, may be treated as a composite predicate, formed from a general subroutine EQ, which uses the character x as a parameter in the calling sequence. =x would then compile into

```
JMS EQ
=
(return false)
(return true)
```

Clearly, this pattern accounts for predicates with multiple parameters, including none. the false return will contain a transfer, corresponding to the spontaneous transition of the transition diagram which the REC expression defines, while for the true return there will occur further subroutine jumps corresponding to subsequent operators.

With these preliminaries we may now turn to the CONVERT program REC, an annotated listing of which we give below.

```
        DEFINE ((

    (REC (LAMBDA (L) (PRINTLIST (CONVERT
    (QUOTE (
        OP       PAV      (=OR= R W)
```

For the purposes of this program, three classes of letters are distinguished: Operators (OP), Predicates (PR) and compound predicates (CP). In each category its members are listed, and treated as PAV's by the CONVERT program.

```
        PR       PAV      (=OR= Q)
        CP       PAV      (=OR= EQ QU)
        ))
    (QUOTE (
        X (XXX)
        ))
    L
    (QUOTE (*O (
        (PR            ((JMS PR) (JMP FA)))
```

Predicates are compiled as a subroutine call followed by a transfer to FA. FA is the heading corresponding to the FALSE exit of the segment under compilation, this transfer is skipped over when the predicate is true.

```
        (OP            ((JMS OP))
```

Operators are compiled by a simple subroutine call.

```
        ((CP X)        ((JMS CP) (X) (JMP FA))
```

Compound Predicates are compiled as Predicates, but their parameter is included as part of the calling sequence.

```
        ((**)          ((JMP OF) FA))
```

The CONVERT program is written in such a way that it does not distinguish CDR of a list from a list. However, these have to be processed differently, and are therefore distinguished by a double asterisk placed in front of a fragment which has arisen as CDR of a list. When only the double asterisk is left, the end of the list has been reached, the spontaneous transition (JMP OF) corresponding to the fact that each parenthesized REC expression is regarded as a predicate is inserted, and the heading FA is placed, since we have now arrived at the first state outside the parenthesis to which all false exits in the last segment must procede.

```
        ((** CO XXX)  ((JMP HE) FA (*SKEL* FA EXPR =GNSY= (=REPT= (** XXX)))))
```

When, in examining a REC expression element by element, we arrive at a colon (CTSS DOES NOT LET US WRITE ALL CHARACTERS, AN IDIOSYNCRACY OF THE LISP INPUT ROUTINE), we write a spontaneous transition to the initial state (JMP HE), note the false exit point of all predicates in the previous segment, and define a new false exit point for the ensuing segment. The analysis continues with the remainder of the REC expression, ** serving as a signal that we do not deal with a new expression..

```
        ((** SC XXX)  ((JMP TR) FA (*SKEL* FA EXPR =GNSY= (=REPT= (** XXX)))))
```

When a semicolon is encountered, a spontaneous transition is made to the TRUE final state (JMP TR), the false exit point of all predicates in the previous segment is noted, and a new false exit point is established for the ensuing segment. The analysis then procedes with the remainder of the expression.

```
        ((** X  XXX)  ((*REPT* X) (*REPT* (** XXX))))
```

If neither delimiter is encountered, we compile the CAR and then the CDR of the expression. Car's and CDR's are not treated uniformly because a new initial state has to be established for each subexpression, but <u>not</u> for each CDR.

```
        ((===)         (=SKEL=  HE EXPR =GNSY=    TR =EXPR= =GNSY=   OF EXPR FA
                       (HE (*SKEL*   FA EXPR =GNSY=   (=REPT= (** *SAME*))) TR))
```

• In compiling a parenthesized expression, provision must be made for the initial state, TRUE final state, and FALSE final state, all of which are defined as GENSYM's. These labels must be included at appropriate points in the compiled code.

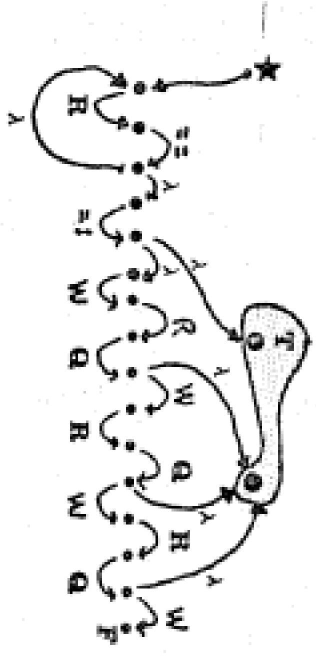
```
        )))
        ))))

    (PRINTLIST (LAMBDA (X) (PROG (Y) (SETQ Y X) (CLOCK ()) A (PRINT (CAR Y))
    (SETQ Y (CDR Y)) (COND ((NULL Y) (RETURN (CLOCK T)))) (GO A))))
```

PRINTLIST is an auxiliary function which allows listing the compiled program with one PDP-8 instruction per line, rather than as a compact list in the usual manner that LISP would print a result.

```
        ))
```

As an example of the operation of REC we may consider the following example. (REC L) is a function whose argument is the REC expression which is to be compiled. On account of inherent limitations in the orthography of the CTSS LISP input routine, certain substitutions had to be made: SC for ;, CO for :, (EQ X) for =X, (QU X) for "X.

```
rec ((r (eq =) co (eq :) sc w r q w r q w r q w))
G03163          Initial Point
(JMS R)         R
(JMS EQ)        ==
(=)             parameter
(JMP G03165)    false
(JMP G03163)    :
G03165          false exit of last segment
(JMS EQ)        =:
(:)             parameter
(JMP G03166)    false
(JMP G03164)    ;
G03166          false exit of last segment
(JMS W)         W
(JMS R)         R
(JMS Q)         Q is an arbitrary predicate
(JMP G03167)    false exit
(JMS W)         W
(JMS R)         R
(JMS Q)         Q
(JMP G03167)    f
(JMS W)         W
(JMS R)         R
(JMS Q)         Q
(JMP G03167)    f
(JMS W)         W
(JMP FA)        exit from last segment to FALSE final state
G03167          continuation on higher level, exit of F's in last segment
G03164          TRUE final state, exit of all semicolons
5               (time of execution)
```



The program which is generated is incomplete in the sense that it itself should be finished off as a subroutine, with a blank entry point bearing an appropriate label, and terminated with appropriate ISZ's and JMP I's.

For ease of reference we conclude with an unannotated listing of the program.

```
     DEFINE ((

(REC (LAMBDA (L) (PRINTLIST (CONVERT
(QUOTE (
     OP        PAV       (=OR= R W)
     PR        PAV       (=OR= Q)
     CP        PAV       (=OR= EQ QU)
     ))
(QUOTE (
     X (XXX)
     ))
L
(QUOTE (*0 (
     (PR                ((JMS PR) (JMP FA)))
     (OP                ((JMS OP)))
     ((CP X)            ((JMS CP) (X) (JMP FA)))
     ((**)              ((JMP OF) FA))
     ((** CO XXX)       ((JMP HE) FA (*SKEL* FA EXPR =GNSY= (=REPT= (** XXX)))))
     ((** SC XXX)       ((JMP TR ) FA (*SKEL* FA EXPR =GNSY= (=REPT= (** XXX)))))
     ((X* X  XXX)       ((*REPT* X) (*REPT* (** XXX))))
     ((===)             (=SKEL=   HE EXPR =GNSY=   TR EXPR =GNSY=   OF EXPR FA
                        (HE (*SKEL*   FA EXPR =GNSY=   (=REPT= (** *SAME*))) TR)))
     )))
     ))))

(PRINTLIST (LAMBDA (X) (PROG (Y) (SETQ Y X) (CLOCK ()) A (PRINT (CAR Y))
(SETQ Y (CDR Y)) (COND ((NULL Y) (RETURN (CLOCK T)))) (GO A))))

     ))
```

REFERENCES

REC:

Joseph E. Grimes and Harold V. McIntosh, "SYMBOL MANIPULATION WITH REC/S" (unpublished) 1967.

Harold V. McIntosh, "REGULAR EXPRESSIONS," Lecture notes for Mathematical Logic II (1968) (unpublished) ESFM.

----- "REGULAR EXPRESSION COMPILER, PROGRAM LISTING," (unpublished) (1966).

----- "MEMORANDUM: REC/A (ARITHMETICVVERSION OF REC)," (unpublished) (1966)

----- "MEMORANDUM: REC/T (TAPE HANDLING VERSION OF REC)," (unpublished) (1967).

----- "MEMORANDUM: MODIFICATIONS TO REC/T," (unpublished) (1967).

Adolfo Guzman and Evodio Lopez, "PROGRAM LISTING FOR IBM-1130 REC," (unpublished) (1967).

CONVERT:

Adolfo Guzman and Harold V. McIntosh, "CONVERT," Communications of the Association for Computing Machinery 9 604-615 (1966).

Harold V. McIntosh and Adolfo Guzman, "A MISCELLANEY OF CONVERT PROGRAMMING," Project MAC Artificial Intelligence Group Memo 130 (April 1967).