INR: A Program for Computing Finite Automata[*]

J. Howard Johnson
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

8 JUL 1988 11:27

Abstract

INR is a program that computes finite state automata and transducers from provided regular and rational expressions.

1 Introduction

INR is a program that constructs finite state automata and transducers from generalized rational expressions. Although originally developed with to support research into approximate string matching [Joh83] it has proved useful for a number of other purposes.

The program operates in a traditional read/evaluate/print mode, that is, it repeatedly reads an expression from the input stream, computes the automaton required, and displays a result. For example, if we wish to find the minimized automaton over the alphabet $\{a, b\}$ that recognizes words that contain at least one "a" and at least one "b". This can be described using the extended regular expression indicating the intersection of words containing a and words containing b:

$$\{a, b\}^* a \{a, b\}^* \wedge \{a, b\}^* b \{a, b\}^*.$$

This expression can be presented to INR in the following form:

```
    {a,b}* a {a,b}* & {a,b}* b {a,b}*;
```

INR will print back

[*] This work was supported by the Natural Sciences and Engineering Research Council of Canada, Grant No. A0237.

```
DFA MIN States: 5      Trans: 9      Tapes: 1  Strg: 1 K

(START) a 2
(START) b 3
2 a 2
2 b 4
3 a 4
3 b 3
4 -| (FINAL)
4 a 4
4 b 4
```

indicating that a minimized deterministic one-tape finite automaton has been computed with 5 states and 9 transitions. It requires 1 Kbyte (1024 bytes) of memory in the representation used by INR. After this the transitions of the automaton are presented in the form

source-state   *label*   *target-state*

so that

```
3 a 4
```

indicates a transition from state 3 to state 2 which reads an *a*. The start state is denoted by the special state name (START) and final states are indicated by a transition on end-marker (-|) to the dummy state (FINAL). In the above example

```
4 -| (FINAL)
```

indicates that 4 is a final state.

If we want to enumerate some of the words recognized by this automaton we can specify the following command to INR:

```
{a,b}* a{a,b}* & {a,b}* b {a,b}*:;
```

The beginning of the result typed would then be:

```
a b
b a
a a b
a b a
a b b
b a a
b a b
b b a
a a a b
```

```
a a b a
a a b b
a b a a
a b a b
a b b a
a b b b
b a a a
 ...
```

The enumerator prints the first few words of the infinite sequence terminating when it reaches a preset limit.

Another interactive use for INR involves testing whether two regular sets are the same. For example consider the set of all words over the alphabet $\{a, b\}^*$ that are not either all a's or all b's. This can be denoted using the extended regular expression

$$\{a, b\}^* - (\{a\}^* \cup \{b\}^*).$$

This can be indicated to INR as:

```
{a,b}* - ( a* | b* );
```

To check that these are the same sets, we can compute the symmetric difference of the two regular languages and verify that is empty. The symmetric difference (exclusive or) operator for INR is the symbol `!`.

```
( {a,b}* a {a,b}* & {a,b}* b {a,b}* )
  ! ( {a,b}* - ( a* | b* ) );
```

This yields the result

```
Empty Automaton
```

indicating that the symmetric difference is empty, that is, that the two indicated languages are identical.

In the first example a generalized regular expression was entered and an automaton was printed back. Instead of this, the automaton could have been written to a file "temp" in the form shown above:

```
{a,b}* a {a,b}* & {a,b}* b {a,b}* :pr temp;
```

Alternatively, it could be written to the file "temp" in a condensed format which usually requires about half of the space and can be loaded more quickly.

```
{a,b}* a {a,b}* & {a,b}* b {a,b}* :save temp;
```

It is also possible to assign it to an internal variable named "temp":

```
temp = {a,b}* a {a,b}* & {a,b}* b {a,b}*;
```

In each of these cases it is then available for computation in further expressions allowing for a considerable saving in typing. The value can be retrieved from the file temp in either `:pr` form or `:save` form by means of the sub-expression `:load temp`. The value from a variable can be retrieved from an internal variable by simply using the variable name itself (if that name has not been used as a transition label).

INR is very often run in a non-interactive (batch) mode using variables to build up larger and larger automata followed by a `:pr` or a `:save` to store the result into a file. For large automata this is the preferred approach since INR provides no mechanisn for recalling past expressions or for editing them.

2 Lexical Conventions

2.1 Symbols

At the lowest level, the input is composed of symbols, delimiters and comments.

Symbols are made up of any non-empty sequence of characters from the set of upper and lower case letters, digits, and the following special characters `.` and `_`. To specify a symbol not satisfying this restriction, simply put back-quote (grave accent) characters (‘) before and after it. The string that results after the back-quotes are stripped off then constitute the symbol. If the desired symbol contains back-quotes they can be escaped with a preceding backslash.

The following characters are recognized as delimiters:

```
! " $ % & ( ) * + , - / : ; < = > ? @ [ \ ] ^ { | } ~
```

All of these characters except `" % < > ~` indicate operations or are used in the specification of operations on automata.

Comments are indicated by a `#`. All text from this character to the next newline is treated as a comment and ignored by the parser.

Within expressions, symbols are interpreted as variables, as primitive tokens (elements of the underlying alphabet), or as filenames. Two symbol tables are maintained to keep track of variable/token usage. The variable symbol table contains all symbols that have been assigned values in assignment statements. The token symbol table contains all symbols that have been previously used as tokens. The printable characters along with the horizontal tab and newline characters are loaded into this table since they are usually used as tokens. When a symbol is recognized by the scanner, it is identified as a variable if it does not occur in the token symbol table. If the symbol occurs in the variable symbol table, that value is used. Otherwise the indicated file is read. The value is the automaton so obtained.

Any symbol beginning with a digit followed by a dot is immediately identified as a token. The digit identifies which tape to which the token is assigned. In the one-tape case, the number is “0”. Thus any symbol can be identified as being a token by preceeding it with the appropriate digit (usually “0”) and a dot. The value of the expression indicated by the symbol is an automaton that recognizes that single token.

If neither of these two cases occurs, the symbol is looked up in the two symbol tables to determine its past usage. If it occurs in only one of the symbol tables that determines whether it is a variable or a token. If it occurs in neither or both then it is assumed to be a token. If in both, a warning diagnostic is given. A warning diagnostic is also provided when an assignment is made to a symbol in the token table.

In order to display the symbol tables there are user level commands. The command “:list” displays the variable symbol table and “:alph” displays the token symbol table. One will immediately notice that the variable symbol table contains the symbol `_Last_`. This refers to the last expression requested that was not assigned to a variable.

3 Generalized Regular Expressions

This section outlines the method for building expressions which define regular languages. This corresponds to the one-tape automaton case for INR. In these situations, INR will compute a minimized deterministic finite automaton (DFA MIN) and display the result. More mathematically inclined readers will notice an anomolous use of set notation that is common in the formal language theory literature and continued here to avoid pedantry and cumbersome notational problems. A string containing one letter is ambiguously written as the letter and a set containing one element is ambiguously written as its element. With this understanding `a` ≡ `{a}` ≡ `{’a’}`.

3.1 Token

A token is a primitive element. It is represented by a symbol that has not been assigned a value as a variable and denotes the automaton that accepts that token and then quits.

```
    abc;                                    (START) abc 2
 or ‘abc‘;                                  2 -| (FINAL)
```

3.2 Variable

A variable is a symbol that has been previously been assigned a value using an assignment statement.

```
    var = abc; var;                         (START) abc 2
                                            2 -| (FINAL)
```

Variables should use names that are clearly distinguishable from tokens. For example, they should be at least two letters in length to avoid conflict with token names.

3.3 String

A string is a possibly empty sequence of tokens and denotes the automaton that recognizes only that sequence. Strings are indicated by writing the tokens in the sequence in which they occur. They normally require spaces between adjacent tokens because of the lexical rules for distinguishing symbols. Thus `a b c` indicates a string with three one-letter tokens whereas `abc` indicates a single token.

Since one-letter tokens are very useful to denote the printable characters of the ASCII (or EBCDIC) character set an alternative notation is provided to indicate strings of one-letter tokens. Any such string can be written without blanks but preceded and followed by single forward quotes (’):

```
     a b c;                                (START) a 2
or   'abc';                                2 b 3
                                           3 c 4
                                           4 -| (FINAL)
```

Any sequence of characters including operators may be used within a string with the following exceptions: `\'` indicates a single forward quote character, `\\` indicates a single backslash character, `\n` indicates a newline character, and `\t` indicates a tab character. Note that strings of one character are a good way of entering single letter tokens that are otherwise treated as operators or white space by the lexical scanner. Thus the token containing exactly one blank can be specified using the notation `' '`, newline can be specified by `'\n'`, and the plus character by `'+'`. Although backquotes also work in these cases, a greater uniformity of notation can be achieved if many strings also occur.

3.4 Empty String

The empty string is indicated by `^`. This symbol is used because of its vague similariy to the Greek letter Λ that is often used for this purpose. Note that in EBCDIC this character has a graphic representation of ¢. Alternative representations is `()` (empty parentheses) or `''` (empty forward quotes).

```
     ^;                                    (START) -| (FINAL)
or   ();
or   '';
```

3.5 Finite Set of Strings

Finite sets of strings are indicated using the standard brace/comma notation. Thus {’ab’,’aa’,’a’} denotes an automaton recognizing a three strings ’ab’, ’aa’ and ’a’.

```
    {’ab’,’aa’,’a’};                        (START) a 2
                                            2 -| (FINAL)
                                            2 a 3
                                            2 b 3
                                            3 -| (FINAL)
```

3.6 Empty Set

The empty set is indicated by {}.

```
    {};                                     Empty Automaton
```

3.7 Alternation (Set Union)

The operator for alternation (OR) is expressed using the vertical bar |.

```
    {’abc’}|{’def’};                        (START) a 2
                                            (START) d 3
                                            2 b 4
                                            3 e 5
                                            4 c 6
                                            5 f 6
                                            6 -| (FINAL)
```

Note that the concatenations are done before the alternation since concatenation has a higher priority than alternation. Parentheses may be used to enforce a different order of evaluation:

```
    a b (c | d) e f;                        (START) a 2
 or a b c e f | a b d e f;                  2 b 3
                                            3 c 4
                                            3 d 4
                                            4 e 5
                                            5 f 6
                                            6 -| (FINAL)
```

Note also that the brace-comma set notation is always equivalent to parentheses with |. This departs slightly from standard set theory but is a natural generalization in this situation. Thus the above automata can also be denoted by a b { c, d } e f.

3.8 Concatenation

Concatenation is represented by adjacency. That is, there is no explicit concatenation operator although ^ can be used if desired.

```
    { a, b } { c, d e };                  (START) a 2
or  { a, b }^{ c, d e };                  (START) b 2
or  {'ac','ade','bc','bde'};              2 c 3
                                          2 d 4
                                          3 -| (FINAL)
                                          4 e 3
```

Note that strings, as discussed before, are just the concatenation of their constituent tokens.

3.9 Kleene Plus and Star

The concatenation closure operators (Kleene + and Kleene $*$) are represented by the corresponding unary postfix operator.

```
    (a b)+;                               (START) a 2
or  'ab'+;                                2 b 3
                                          3 -| (FINAL)
                                          3 a 2

    (a b)*;                               (START) -| (FINAL)
or  'ab'*;                                (START) a 2
                                          2 b (START)
```

The difference is that the second form includes the empty word. Note that parentheses are used to group together the argument to + and *. This is because these two operators are considered to have higher priority than any of the others.

```
    a b*;                                 (START) a 2
or  a (b*);                               2 -| (FINAL)
                                          2 b 2
```

3.10 Optional

The operator indicating zero or one occurrences of a set is denoted by `?`. This can be used to identify optional components.

```
    a b?;                                 (START) a 2
                                          2 -| (FINAL)
                                          2 b 3
                                          3 -| (FINAL)
```

The operators `+`, `*`, and `?` are computed from left to right.

3.11 Concatenation Power

To express the concatenation of a set with itself exactly k times the expression may be followed by a `:` and the value k. Thus `R1 :2` is the same as `R1 R1` and `R1 :3` the same as `R1 R1 R1`.

```
     {a,b} :2;                          (START) a 2
                                        (START) b 2
                                        2 a 3
                                        2 b 3
                                        3 -| (FINAL)
```

To express the concatenation of a set with itself k or fewer times the notation `R1? :`k can be used. To express the concatenation of a set with itself more than k times the set difference can be used: `R1* - (R1? :`k`)`.

3.12 Concatenation Quotient

If concatenation is interpreted as a multiplication of strings so that `'abc'` · `'def'` = `'abcdef'` we can define a right quotient operator / as `'abcdef'` / `'def'` = `'abc'` and a left quotient operator \ as `'abc'` \ `'abcdef'` = `'def'`. The (left or right) quotient of sets is defined as the union of their elementwise quotients.

```
     'ab' \ 'abcdef' / 'ef';            (START) c 2
or   'cd';                              2 d 3
                                        3 -| (FINAL)
```

3.13 Set Intersection

The automaton recognizing the intersection of two regular languages is indicated by the `&` operator.

```
     {a,b}* & b {a,c}*;                 (START) b 2
or   b a*;                              2 -| (FINAL)
                                        2 a 2
```

3.14 Set Difference

The operator `-` indicates the difference of two two regular languages.

```
     {a,b}* - b a*;                     (START) -| (FINAL)
or   (^ | a | b a* b) {a,b}*;           (START) a 2
                                        (START) b 3
                                        2 -| (FINAL)
                                        2 a 2
                                        2 b 2
                                        3 a 3
                                        3 b 2
```

3.15 Symmetric Difference (Exclusive Or)

The symmetric difference of two languages is the set of words that are in one of the languages but not the other and will be indicated by the `!` operator. Thus $R_1 ! R_2 \equiv (R_1 - R_2) \cup (R_2 - R_1)$.

```
    {a,b}* ! {a,c}*;                      (START) a (START)
or  a* ( b {a,b}* | c {a,c}* );           (START) b 2
                                          (START) c 3
                                          2 -| (FINAL)
                                          2 a 2
                                          2 b 2
                                          3 -| (FINAL)
                                          3 a 3
                                          3 c 3
```

Note that the symmetric difference of a language with itself is the empty set. Thus this operation can be used to check the equivalence of two regular languages.

3.16 Shuffle

The shuffle of two words is the set of words formed by alternatively taking some letters from each of the words. Thus the shuffle of the words `'ab'` and `'cd'` is the set `{'abcd','acbd','acdb','cabd','cadb','cdab'}`. The shuffle of two sets of words is the union of the shuffles of all pairs of words such that one word is taken from each set. The shuffle operator is indicated by the symbol `!!` which mildly suggests the symbol Ш that is often used for this purpose.

```
    'ab' !! 'cd';                         (START) a 2
or  {'abcd','acbd','acdb',                (START) c 3
      'cabd','cadb','cdab'};              2 b 4
                                          2 c 5
                                          3 a 5
                                          3 d 6
                                          4 c 7
                                          5 b 7
                                          5 d 8
                                          6 a 8
                                          7 d 9
                                          8 b 9
                                          9 -| (FINAL)
```

3.17 Active Alphabet

The active alphabet of an automation is the set tokens that can actually occur in a word in the language.

```
    ’abcd’ / ’d’ :alph;                (START) a 2
or  {a,b,c};                           (START) b 2
                                       (START) c 2
                                       2 -| (FINAL)
```

3.18 Reverse

The reverse of a string ’abcd’ is the string ’dcba’. The reverse of a set of strings is the set of their reverses.

```
    {’abc’,’abd’} :rev;                (START) c 2
or  {’dba’,’cba’};                     (START) d 2
                                       2 b 3
                                       3 a 4
                                       4 -| (FINAL)
```

3.19 Prefixes

The prefixes of ’abcd’ is the set { ^, ’a’, ’ab’, ’abc’, ’abcd’ }. The prefixes of a set of strings is the union of the prefix sets for each word. Note that R1 :pref is an alternate (more efficient) way of indicating the set R1 / (R1 :alph)*.

```
    {’abc’,’abd’} :pref;               (START) -| (FINAL)
or  {’abc’,’abd’} / {a,b,c,d}*;        (START) a 2
or  {^,’a’,’ab’,’abc’,’abd’};          2 -| (FINAL)
                                       2 b 3
                                       3 -| (FINAL)
                                       3 c 4
                                       3 d 4
                                       4 -| (FINAL)
```

3.20 Suffixes

The suffixes of ’abcd’ is the set { ^, ’d’, ’cd’, ’bcd’, ’abcd’ }. The suffixes of a set of strings is the union of the suffix sets for each word. Note that R1 :suff is an alternate way of indicating the set (R1 :alph)* \ R1.

```
    {’abc’,’abd’} :suff;               (START) -| (FINAL)
or  {a,b,c,d}* \ {’abc’,’abd’};        (START) a 2
or  {^,’c’,’d’,’bc’,’bd’,              (START) b 3
      ’abc’,’abd’};                    (START) c 4
                                       (START) d 4
                                       2 b 3
                                       3 c 4
                                       3 d 4
                                       4 -| (FINAL)
```

The factors (subwords) of a language can be obtained by using finding the prefixes of the suffixes: `R1 :suff :pref`. For large sets this is preferable to the equivalent but much more expensive `R1 :pref :suff`.

3.21 Complement

Since the alphabet of interest is inferred from the context, the complement must be defined with respect to a specific universe. Thus there are two basic forms `R1 :acomp` and `R1 :comp`. The first of these is equivalent to `(R1 :alph)* - R1` and the second is equivalent to `(SIGMA :alph)* - R1` where `SIGMA` contains the reference alphabet.

```
    b a* :acomp;                       (START) -| (FINAL)
or  SIGMA = {a,b}; b a* :comp;         (START) a 2
or  {a,b}* - b a*;                     (START) b 3
or  (^ | a | b a* b) {a,b}*;           2 -| (FINAL)
                                       2 a 2
                                       2 b 2
                                       3 a 3
                                       3 b 2
```

3.22 Stretching

An operation that is sometimes useful involves replacing each letter in the word by a sequence of k of that letter. This can be done using the `$` operator followed by a string of k zeros.

```
    'abc' $ 0 0;                       (START) a 2
or  'aabbcc';                          2 a 3
                                       3 b 4
                                       4 b 5
                                       5 c 6
                                       6 c 7
                                       7 -| (FINAL)
```

3.23 Priorities of Operators

Within a pair of matching parentheses or braces the expression is computed in the following order:

1. Kleene $+$, Kleene $*$ and Optional operators are evaluated from left to right on their single argument.
2. Concatenations
3. One pair of matching left and right quotients are performed if they exist.

4. Set intersection and set difference operations are performed from left to right.

5. Alternation (set union), symmetric difference, and shuffle operations are performed from left to right.

6. One stretch operation

7. The colon unary operations concatenation power, :alph, :rev, :pref, and :suff are performed left to right.

8. The alternation implied by commas within braces is performed last.

Note that the colon unary operations will often require surrounding parentheses in a larger expression.

Note that parentheses or braces can be used to perform grouping when the priorities of operators would imply an incorrect interpretation. Use of parentheses is usually recommended unless a set constructor is needed for clarity. However, note that commas within parentheses have another meaning to be discussed in a later section. Another form of grouping is performed by string quotes. Thus `'ab'*` is equivalent to `(a b)*` and not to `a b*`.

4 Generalized Rational Expressions

A binary relation is a set of ordered pairs of elements. Thus just as a language can be regarded as a set of words, we can define a binary language relation as a set of pairs of words. Binary language relations are often called transductions since they indicate a transformation or translation process. Of particular interest are transductions which can be performed by a finite state machine. These are called finite state transductions or simply finite transductions. A two-tape finite automaton or finite transducer determines whether a pair of words is in the transduction in the following way: The two words under consideration are placed on two input tapes followed by special end-marker characters and the machine is put into its start state. The machine then picks an operation (non-deterministically) which may involve reading some number of characters from either (or both or neither) of the two input tapes, advancing the read heads of the two tapes and making a transition to a new state. The pair of words is accepted if there exists such a sequence of steps such that all of both input tapes are read and the transducer is left in a final state.

As described above the machine is considered to have two input tapes. However, one of the input tapes can be made into an output tape. In this case the transducer is viewed as reading an input word and writing an output word that is non-deterministically chosen from the set of words related to the input word. Still another interpretation is to consider the input word as determining a set of output words that are related to it.

The above description of finite transducers implies a non-deterministic choice of the next transition to make. The class of relations that result when the transducer is made deterministic, either with two input tapes or with an input tape and an output tape are proper subclasses of finite transductions. However, it is possible to remove much of the non-determinism using the idea of a characterizing language.

Before defining a characterizing language we will mark the alphabets with either a 0 or a 1 to indicate whether they are to be read from tape 0 or tape 1. Thus a will occur in the forms a_0 and a_1. We can then define languages over this extended alphabet. From each language we can then obtain a relation by projecting out the tape 0 letters and the tape 1 letters as the two components of the ordered pair. A language then (strongly) characterizes a relation if this projection process on the language yields exactly the given relation. Note that each word in the characterizing language is a shuffle of related marked words.

It is always possible to find a regular characterizing language for any finite transduction although the choice is not unique and cannot be made canonical without restricting the class of languages or being uncomputable. Thus a convenient way of describing finite transductions is by means of some regular characterizing language. Then the usual optimizations and transformations which apply to regular languages can then be applied without changing the implied transduction. This is exactly what INR does. Instead of a_0 and a_1, INR uses the notation `0.a` and `1.a`.

Regular languages are defined in other ways than by their automata however. For example regular expressions, especially when extended to include the Boolean operations are a very expressive way to present regular languages. In the case of the union operation, this behaviour carries over to relations since the relation defined as the union of two characterizing languages is always the union of the relations defined by the two characterizing languages. However, there is a slight problem with the other Boolean operations when they are applied to relations though their characterizing languages. The result is not usually the same as applying the Boolean operation to the relation itself. In fact, the operations intersection, set difference, symmetric difference, and complement can yield transductions that are not finite state.

Note that we could also define ternary relations as sets of ordered triples of elements and introduce ternary finite transductions. We can define characterizing languages by using marking the letters with numbers 0, 1, or 2. Again there will always be a regular characterizing language, and similar closure properties. The process can, of course, be continued to relations of any degree. INR currently restricts the degree to 10: `0.a`, `1.a`, . . ., `9.a`.

Finite transductions are also called rational relations or rational transductions since they are closed under the rational operations: concatenation, union, and concatenation closure (Kleene plus). They are also closed a number of other useful operations which will be described. This section outlines the additional operators provided with INR which allow the definition of rational relations.

4.1 Token

To indicate that a token is associated with a particular tape, precede it by the tape number and a period.

```
    1.abc;                                 (START) 1.abc 2
or  `1.abc`;                               2 -| (FINAL)
```

Note that `` 1.`abc` `` will not work because the lexical scanner parses this as `` 1.` `` followed by `abc` followed by an unmatched `` ` ``.

4.2 Variable

A variable may be assigned a rational relation in the same way that it can be assigned a regular set.

4.3 String

Strings are as for regular languages except for the need to shift them from tape zero to some other tape. To define a string on tape 1, there are four basic techniques depending on the situation.

1. Build the string from tokens which specify the tape. Then `'abc'` on tape 1 becomes `1.a 1.b 1.c`.

2. Use (parenthesis/comma) tuple notation. Then `'abc'` on tape 1 becomes `( ^, 'abc' )` or simply `(,'abc')`.

3. Use the tape shifting operator `[]`. Then `'abc'` on tape 1 becomes `['abc']`.

4. Use the tape renumbering operator `$`. Then `'abc'` on tape 1 is `( 'abc' $ 1 )`.

```
    1.a 1.b 1.c;                           (START) 1.a 2
or  (,'abc');                              2 1.b 3
or  ['abc'];                               3 1.c 4
or  ('abc' $1);                            4 -| (FINAL)
```

To indicate a string on tape 2 a similar technique can be used:

```
    2.a 2.b 2.c;                           (START) 2.a 2
or  (,,'abc');                             2 2.b 3
or  [['abc']];                             3 2.c 4
or  ('abc' $2);                            4 -| (FINAL)
```

The empty string is indicated as in the regular case by `^`, `()`, or `''` and is not associated with any tape.

4.4 Set Operations

Rational relations are closed under union and the syntax and behaviour is as for regular languages. However, they are not closed under intersection, difference, symmetric difference or complement so INR interprets these four operations as applying to the characterizing languages that represent the rational relations and a warning is printed. Some advanced uses of INR involving a careful use of this feature will be discussed in a later section.

4.5 Concatenation Operations

The operations concatenation, Kleene plus and star, optional and concatenation power are as in the regular case. Concatenation quotient is not defined. On the other hand, `:alph`, `:rev`, `:pref`, and `:suff` are available and apply to the characterizing language.

4.6 Tuple Formation (Cartesian Product)

The tuple notation consisting of parentheses and commas can be used for more than just strings. It computes the number of tapes used by the left operand, adds that number to each tape number of the right operand and finaly concatenates the two machines together in the indicated order. Thus, as far as the characterizing languages are concerned, the comma operator functions as a concatenation following a tape shift.

```
    ( a*, b* );                         (START) -| (FINAL)
or  0.a* 1.b*;                          (START) 0.a (START)
                                        (START) 1.b 2
                                        2 -| (FINAL)
                                        2 1.b 2
```

4.7 Tape Shifting

The operator `[]` can be used to cause all of the tapes to be incremented by one.

```
    a* [b*];                            (START) -| (FINAL)
or  ( a*, b* );                         (START) 0.a (START)
or  0.a* 1.b*;                          (START) 1.b 2
                                        2 -| (FINAL)
                                        2 1.b 2
```

```
    [b*] a*;                            (START) -| (FINAL)
or  (, b* ) a*;                         (START) 1.b (START)
or  1.b* 0.a*;                          (START) 0.a 2
                                        2 -| (FINAL)
                                        2 0.a 2
```

These two automata describe the same rational relation although, of course, the characterizing languages are different.

4.8 Tape Projection

The $ operator can be used to reorganize the tape structure of a transducer. One such operation is the selection (projection) of some of the tapes with the implied removal of the others. Rational relations are closed under all such projections. For example, for a binary transduction R1, it is often useful to construct an automaton which recognizes the domain of the transduction or the range. The domain is projected by INR using the notation `R1 $ 0` and the range by the notation `R1 $ 1`.

```
    (a,b)* $ 0;                       (START) -| (FINAL)
or  a*;                               (START) a (START)
```

```
    (a,b)* $ 1;                       (START) -| (FINAL)
or  b*;                               (START) b (START)
```

Projections of ternary relations can also be done. Thus if R2 is a ternary transduction `R2 $ (0,2)` will select tapes 0 and 2 renumbering them as 0 and 1.

```
    (a,b,c)* $ (0,2);                 (START) -| (FINAL)
or  (a,c)*;                           (START) 0.a 2
                                      2 1.c (START)
```

4.9 Tape Renumber

The $ operator can also be used to cause a renumbering of the tapes. For example, the inverse of a binary relation can be formed by applying the operator `$ (1,0)`.

```
    (a,b)* $ (1,0);                   (START) -| (FINAL)
or  ([a] b)*;                         (START) 1.a 2
                                      2 0.b (START)
```

Projection and renumbering can be done together by a natural extension of the basic idea.

```
    (a,b,c)* $ (2,0);                 (START) -| (FINAL)
or  (a,c)*;                           (START) 1.a 2
                                      2 0.c (START)
```

4.10 Stretching

A third use the `$` operator is stretching. A simple example of this was demonstrated in the last section and the idea can be generalized to any tape.

```
    (a,b)* $ (0 0,1 1 1);               (START) -| (FINAL)
or  (a a,b b b)*;                       (START) 0.a 2
                                        2 0.a 3
                                        3 1.b 4
                                        4 1.b 5
                                        5 1.b (START)
```

A more useful type of stretching involves the conversion of an automaton that accepts a regular language to one that accepts a word in that language and copies it to the output tape.

```
    a b* $ (0,0);                       (START) 0.a 2
or  (a, a) (b, b)*;                     2 1.a 3
                                        3 -| (FINAL)
                                        3 0.b 4
                                        4 1.b 3
```

Note that the right argument is an automaton that recognizes a string of tokens of the form *digit.digit* and this may be created in any way. For example, if `$ (0,0)` is used often, a variable containing the value `(0,0)` could be defined to provide an improved readability in this case. Note also that how the stretching is done in the multitape case. The tape numbers are assigned in the order that they occur in the right argument.

```
    a b* $ [0] 0;                       (START) 1.a 2
or  ([a] a) ([b] b)*;                   2 0.a 3
                                        3 -| (FINAL)
                                        3 1.b 4
                                        4 0.b 3
```

4.11 Composition

The composition operator is denoted by `@`. The last tape of the left operand is joined with the first tape of the right operand. This yield the usual composition of relations when applied to binary relations and the usual "apply" when the left operand is over one tape and the right operand is a binary relation.

```
    (a,b)* @ (b,c)*;                    (START) -| (FINAL)
or  (a,c)*;                             (START) 0.a 2
                                        2 1.c (START)
```

If the left argument is a regular language then the compose operator computes the subset of the range that is the image of this set. This is a generalization of the application of functions to domain values.

```
    'aa' @ (a,b)*;                          (START) b 2
or  'bb';                                   2 b 3
                                            3 -| (FINAL)
```

The compose operator can also be used to compute the subset of the domain that maps to a subset of the range. This is really an apply of the inverse relation.

```
    (a,b)* @ 'bb';                          (START) a 2
or  'bb' @ ((a,b)* $ (1,0));                2 a 3
or  'aa';                                   3 -| (FINAL)
```

4.12 Join

The join operator is denoted by `@@`. It has the same effect as composition except that the last tape of the left operand is represented in the output. Thus the composition operator is equivalent to a join followed by a projection that removes the tape over which the join was done. The join operator can be used to restrict the domain or range of a transduction.

```
    'aa' @@ (a,b)*;                         (START) 0.a 2
or  (a,b)* @@ 'bb';                         2 1.b 3
or  (a,b) :2;                               3 0.a 4
                                            4 1.b 5
                                            5 -| (FINAL)
```

4.13 Composition Power

The k-fold composition of a binary rational relation is denoted by `:(`k`)`. That is, the number is written in parentheses.

```
    (a,'aa')* :(2);                         (START) -| (FINAL)
or  (a,'aa')* @ (a,'aa')*;                  (START) 0.a 2
or  (a,'aaaa')*;                            2 1.a 3
                                            3 1.a 4
                                            4 1.a 5
                                            5 1.a (START)
```

4.14 Extending Or (ElseOR)

Often it is desirable to extend one function by a second function, that is, extend the domain to include that covered by the second function without disturbing the values assigned by the first function within its domain. This operator is denoted by the symbol `||`. Thus if f and g are functions then f `||` g will agree with f whenever the argument is in *domf*. Otherwise it will agree with g. Note that `R || S` is simply a shorter form of `R | ( (S$0)-(R$0) @@ S )`

```
    (a,c) || (a,b)*;                    (START) -| (FINAL)
or  ^ | (a,c) | (a,b)(a,b)+;            (START) 0.a 2
                                        2 1.c 3
                                        2 1.b 4
                                        3 -| (FINAL)
                                        4 0.a 5
                                        5 1.b 6
                                        6 -| (FINAL)
                                        6 0.a 5
```

4.15 Compute Subsequential

One interesting subcase of finite transductions is the deterministic transductions, that is, the transductions that can be performed in one left to right pass without backtracking or non-deterministic choices with only a finite number of possible states. Thus it is desirable to identify when this can be done and construct a deterministic transducer when it does. This is performed by the `:sseq` operator.

```
    (a,c) || (a,b)* :sseq;              (START) 0.-| 2
                                        (START) 0.a 3
                                        2 1.-| 4
                                        3 0.-| 5
                                        3 0.a 6
                                        4 -| (FINAL)
                                        5 1.c 2
                                        6 1.b 7
                                        7 1.b 8
                                        8 0.-| 2
                                        8 0.a 7
```

Note that `(START)`, `3` and `8` are read states since they must read exactly one character (or end-marker) from tape zero. States `2`, `5`, `6` and `7` are write states. They can only write and the choice is determined. Note also that explicit end-markers are introduced. The existence of explicit end-markers can confuse some of the operations that use the characterizing language idea. End-markers can be deleted by using the `$` operator. It removes explicit end-markers from any tape.

5 Coercing and Displaying Operators

5.1 Coercing Operators

Internally *INR* operates on the following laziness principle. Each automaton can be in one a number of modes depending on how much work has been done on it. At one extreme, the automaton is a collection of transitions appearing any order, and possibly with duplicate transtions or inaccessible states. At the other extreme, the automaton is a minimized deterministic automaton with its transitions sorted and unduplicated. Each operation then automatically coerces its arguments to the required form by calling the appropriate routine and sets the mode of its result value to a appropriate value. For example, union takes two sets of transitions and after renaming combines them into one set. On the other hand, intersection requires that its operands be minimized deterministic automata and produces as a result a not necessarily minimized deterministic automaton. For debugging and education purposes, the coercing operator can be explicitly used:

:nfa Sort and unduplicate transitions.

:trim Remove states that are unreachable. (Reduce)

:lameq Remove lambda equivalent states.

:lamcm Combine lambda implied states.

:closed Form lambda closure.

:dfa Form deterministic machine. (Subsets Construction)

:min Minimize the DFA.

Thus `<expression> :trim;` will print the trim (reduced) automaton corresponding to the expression. Note that coercing is one way and that some operators are smart about promoting their results.

5.2 Displaying Operators

The following operators cause some form of printing:

:pr Print in readable format.

:save Save in condensed format.

:report Write a one line report.

:enum Enumerate words.

:card Count number of words and print.

:length Print length of shortest word.

The operation `:pr` may be followed by a filename in which case the automaton is printed into that file. The operation `:save` must be followed by a filename indicating the desired file. The operation `:enum` may be followed by a positive number indicating the limit on printing desired. The number bounds the length of the longest word (in tokens) and the number of tokens to be printed. Thus 1000 means that no word longer than 1000 characters will be printed and that after 1000 tokens have been displayed no further words will be started.

5.3 Input Operator

To cause an automaton to be loaded the operator `:read` followed by a filename can be used. This will load a file in either `:pr` or `:save` format.

6 Statement Forms

6.1 Evaluate Statement

An expression of the form `<expression>;` requests the evaluation of the expression and the displaying of the resulting automaton. The variable `_Last_` is updated by this statement. If the last operation performed is a coercing operation, then the automaton is printed. If the last operation performed is not a coercing operation or a printing operation, then the automaton is automatically coerced to `:min` and printed. Thus the default is that a minimized deterministic automaton is printed.

6.2 Enumerate Statement

An expression of the form `<expression>:;` requests the evaluation of the expression and the enumeration of the words recognized by the resulting automaton. The variable `_Last_` is also updated by this statement.

6.3 Assignment Statement

Variables are simply symbols for which a value has been assigned. Thus form of the assignment is: `<symbol> = <expression>;` Any later references to the symbol will refer to the set assigned to it. This is particularly useful for large expressions since often repeated computations can be eliminated.

There is a built-in variable `_Last_` which is assigned to the last evaluated expression that was not directly assigned. At any time a report on the assigned variables may be requested by the command `:list`.

6.4 Command Statement

A number of commands that cause certain actions to be performed are provided.

:alph; Display token symbol table.

:free; Display status of free lists.

:list; Display variable symbol table.

:noreport; Turn off (verbose) debug tracing.

:pr; Save all variables in files with the same names.

:quit; Terminate session.

:report; Turn on (verbose) debug tracing.

:save; Save all variables in files (condensed) with the same names.

:time; Display time since last :time call (VAX only).

7 Internals

The internal workings of *INR* is achieved through the use of an abstract data type whose values are automata. A large number of operations are defined on objects of this type which yield other automata. Thus, for example, the operation union is implemented internally as a function A_union which accepts two automata as arguments and returns another.

In order to provide better access to the operations, an interface was designed and implemented using "yacc", a LALR(1) parser generator provided with UNIX. The result is a user-friendly method of specifying transformations on automata. However, since the user commands are in the form of generalized rational (or regular) expressions, the impression is that of translation from expression form to automaton form. However, there is no represention of expressions internally and even strings and sets are coded as automata.

The form of the data structure is a ternary relation indicating the from state, the transition label, and the to state. Special states `(START)` and `(FINAL)` and special transitions `^^` and `-|` simplify the processing. The abstract data type also contains a mode indicator that indicates the coercing level. This allows the lazy evaluation mechanism to work and avoids a lot of redundant computation.

References

[Joh83] J. Howard Johnson. *Formal Models for String Similarity.* PhD thesis, University of Waterloo, 1983. Available as University of Waterloo Research Report CS–83–32 (128 pages).