Rule-based Programming in FLEX

written by R. Shapiro
18 July 1986

Table of Contents

1. Introduction

This document describes the use of the rule-based programming system FLEX. It is presumed that the reader is familiar with the notion of rule-based programming. Knowledge of some such system (e.g., LOOPS or MYCIN) is useful as well. Working knowledge of Lisp-Machine Lisp (or at least some Lisp dialect) is essential.

2. Rule packets

Rules defined in FLEX are organized into rule-packets, following the LOOPS model of rule-based programming. A rule packet is a description which consists of a set of rules, a set of local variables, and a list of arguments. Certain varieties of rule packets (see 2.2) also contain a list of other packets (called sub-packets). The rules of a given packet can reference any of the local variables or arguments of the packet which owns them (as well as the variables of packets of which the owning packet is a sub-packet).

The rule packet itself is just a description. It is not something which can be invoked in any sense. Rather, it is used to create instances of itself, which *can* be invoked. What it means to invoke a rule packet instance is determined by the type of the defining packet. In general, a packet instance can be thought of as a context with state (i.e., local variables) which will run the rules which it contains.

The two basic varieties of rule packets are forward-chaining packets and backward-chaining packets, described in the two following sections.

2.1 Forward-chaining packets

The instances of a forward-chaining rule packet are invoked as though they were functions. That is, the name associated with an instance defines a function (or a method) which, when called, will try the rules which the defining rule packet contains.

There are four types of forward-chaining packets which try the rules in different ways.

DO-1 An instance of a DO-1 packet will run the rules in succession until one fires.

DO-ALL An instance of a DO-all packet will run each rule in succession.

WHILE-1 An instance of a WHILE-1 packet runs a double-iteration of the rules as follows. The inner loop is equivalent to do-1 (i.e., stops when a rule fires or when all the rules have been tried, whichever comes first). The outer loop runs as long as the while clause associated with the packet (see 2.3.1) is true and the packet

instance has not been explicitly stopped by one of the rules (a special function is provided for this explicit stopping).

WHILE-ALL This is like while-1, except that the inner loop is like do-all instead of do-1 (i.e., try each rule).

An instance of a rule packet returns multiple values as follows. The first value is NIL if no rule fired, non-NIL if any rule fired. The other values are the principal values (see 3.3) of the local variables of the packet (in their definition order).

2.1.1 Expansion of forward-chaining packets

An instance of a forward-chaining rule packet is equivalent to a dynamically constructed function. If, at some point, the rules and variables of the defining packet are considered fixed, then the instances can be converted from dynamic functions into their static equivalents, using the now fixed state of the packet. This process is called expansion. Such expanded functions can then be compiled, yielding highly efficient forms of forward chaining. FLEX provides several functions for performing this expansion and compilation.

(compile-packets! packet-list &optional expand-only)
This function will expand (and compile as well, unless expand-only is non-nil) the packets which are elements of packet-list. This function is useful when a list of packet instance names already exists. Such a list is **$expansion-packets** which, after loading a file containing packet definitions, will contain all of the packet instance names defined by that file.

(compile-packets &rest packet-names)
This is a macro which will compile the expanded form of the named packets.

(expand-packets &rest packet-names)
This is a macro which will expand the named packets.

(write-expanded-packets packet-list path)
This function will write into the file named by the path argument the expanded form of the packets which are elements of packet-list.

2.2 Backward-chaining packets and domains

Backward-chaining packets are used in the manner of MYCIN. That is, a hierarchy of packets is defined whose purpose is to infer values for special variables (called goal variables). The packet at the top of this hierarchy is called a domain. Packets are instantiated as needed, and values for variables are set either by querying the user or by running rules whose action is to set the variable in question. A single instantiation of a domain, and all of the subsequent instantiations of its subpackets and the resulting values of variables, is called a session. Calling the domain as a function starts a session -- the resulting instance hierarchy is returned as the result. At the end of a session the values of the domain's goals variables are printed.

2.2.1 Properties of packet variables

Local variables of backward-chaining rule packets need various properties so that any user-interaction concerning them (e.g., asking the user for a value) can go smoothly. The form for setting properties is as follows:

```
(defrulevar <packet-name> <symbol-name>
        { <property-name> <property-value> }* )
```

The properties which are useful are:

:prompt — Value should be a string or a list whose elements are the arguments to the function **format**. **Format** will be called with these arguments if the user is asked to supply a value for this variable.

:type — This specifies the type of value that the variable is allowed to take. The legal values for this property are **symbol** if there is a discrete set of symbolic values that the variable may take on, **string** if the variable can take as a value an arbitrary string, or **number** if the variable must take on numeric values. If the variable is to be set-valued (see 3.3.1), then the type name should be prefixed by set-of-, e.g. **set-of-symbol** or **set-of-numeric**. These last are called **set-types**.

:choices — If the variable has type **symbol**, then this property gives the list of legal symbolic values. The form of the property-value is a list of two element lists, one per legal choice. The two element lists for each choice consist of the symbol which is the possible value and the string which a user may type to indicate that value. This is a list similar in construction to the list of choices passed to the **fquery** function.

:list-choices — Value should be t or nil. If the user is asked to supply a value for this variable, and it has a discrete set of permissible values

(specified with the **:choices** property, above), the value of this property determines whether or not the list of permissible values will be displayed.

:trans — Value is either a string, or a list whose members are the arguments to a call to **format**. The string will be printed or **format** called with the list of arguments at the end of a session. Typically this is used to print the value of a goal variable.

:never-ask — Value should be t or nil (default is nil). If the value is t, the user will never be asked to supply a value for this variable.

:ask-first — Value should be t or nil (default is nil, except for lab-data variables for which it is t). If the value is t, this variable will be asked for before any attempt is made to use the rules to infer a value.

2.3 Defining a rule packet

A rule packet definition looks like this:

```
(defrule-packet <packet-name> <argument-list>
    { <rule-description-clause> }*
    { <rule-definition> }* )
```

where

<packet-name> — The name to be used for this packet. This is either a symbol, or a method function-spec (i.e., a list whose car is a flavor name and whose cadr is a message name). The latter form is only relevant for forward-chaining packets. See 2.1 for an explanation of the use of packet names.

<argument-list> — A list of symbols which will be arguments to instances of this packet. Only relevant for forward-chaining packets. Each element of this list is either a symbol or a pair of symbols. In the latter case, the first element of the pair is the argument and the second is its type.

<rule-description-clause> — A symbol or list which sets some property of this packet. See 2.3.1 for a list of legal clauses.

<rule-definition> — A rule definition (see 3.2). The packet-name argument should be left out, however. The packet being defined will be used instead.

2.3.1 Rule packet description clauses

Rule packets have various properties which may be set by providing description clauses in the definition. Each property has an associated keyword, and may also take arguments. The general form of a description clause is a list whose car is the relevant keyword and whose cdr is a list of arguments. In the case that a property takes no

arguments, the description clause is a list just containing the relevant keyword; in this case the keyword itself may be used as the clause (i.e., the symbol instead of a list containing the symbol).

The defined keywords and their arguments are as follows:

:type — One argument, a symbol. One of **backward** (see 2.2), **do-1**, **do-all**, **while-1**, **while-all** (see 2.1 for the last 4). If no type clause is present, **do-1** is the assumed type.

:rules — The arguments are an initial set of rules for this packet. As with rule-description-clauses, these are rule definitions without a packet name.

:documentation — One argument, a string, which is stored with the packet as descriptive information.

:properties — The argument should be an association list where, for each element, the CAR is the property and the CADR is the value. Two predefined properties are :PRE-CONDITION (the value is a predicate which must evalute to true for the packet to run) and :BASIS (the value is a string useful for documentating the reasoning which the packet is intended to perform).

:locals — Arguments are the local variables that instances of this packet will have. Each element of this list is either a symbol or a pair of symbols. In the latter case, the first element of the pair is the local and the second is its type. The values of the locals will be returned as a result of running the packet, in the order in which they appear in this clause.

:make-instances — Arguments are names for instances to be made of this packet. This is only relevant for forward-chaining packets. If a forward-chaining packet definition does not contain one of these clauses, then a new name is generated for the packet, and the name supplied as the packet name is used instead as the name of an instance.

:while — Argument is an arbitrary Lisp expression, used by a **while-1** or **while-all** packet to terminate iteration.

:instantiate-ask-first — Argument is a string, suitable for passing to **yes-or-no-p**. The first time this packet is instantiated in a given context, this is the prompt that will be used. Any subsequent instantiations will use the **:instantiate-ask-rest** string. Both of these are only relevant for backward-chaining packets.

:instantiate-ask-rest — See **:instantiate-ask-first**, above.

:instantiate-when — This is used in conjunction with **:instantiate-ask-first** and **:instantiate-ask-rest**, as follows. If the argument is the symbol **:once-only**, then the packet will be instantiated once without any

interaction with the user. If the argument is the symbol **:ask**, then instantiation will be controlled through interaction with the user (using the prompts defined by the instantiate-ask properties). If the argument is a list whose car is **:ask-if**, then the cadr of the argument is evaluated. If this value is non-nil, then the instantiation will be controlled through interaction with the user, using the questions defined by the instantiate-ask properties. Otherwise, the argument is a Lisp form which is evaluated to determine instantiation (i.e., do it if the value is non-nil).

:lab-data — Arguments are local variables for the packet. This differs from **:local** only in that lab-data local variables will never be inferred (i.e., they must be given values through interaction with the user). Only relevant for backward-chaining packets.

:sub-packets — Arguments are names of packets which are sub-packets of the packet being defined. These should have been defined already. Only relevant for backward-chaining packets.

:super-packets — Arguments are packets that this packet will be a sub-packet of. These should be already defined. Only relevant for backward-chaining packets.

:domains — Same as **super-packets**, except that the containing packet should be defined as a domain.

2.3.2 Rule packet definition examples

```
(defrule-packet fore (x y)
  (:locals a b)
(:make-instances fore1 fore2))
```

This is an example of a do-1 forward rule packet (since no :type clause was present, the default was used). It takes two arguments **x** and **y**. It has two local variables, **a** and **b**. Two instances of this packet have been made, **fore1** and **fore2**. Each is now a function with two arguments (though the execution of the functions isn't meaningful until some rules are defined for the packet **fore**).

```
(defrule-packet fore-while (x y)
  (:type while-all)
  (:locals a b)
  (:while (neq x b))
  (:rules
   (*
     (:conditions (eq y a))
     (:actions  (setq a "y" b 0)))
))
```

This is a while-all packet, with arguments **x** and **y**, and local variables **a** and **b**. The while clause, which controls the looping through the rules, is the form **(neq x b)**.

Also, a rule (with no name) has been defined for this packet -- if **y** is eq to **a**, the rule will fire and the packet will return T, "y" and 0. Note that no instances were explicitly asked for. Therefore a new packet name is generated, and **fore-while** becomes the name of an instance.

```
(defrule-packet marriage ()
  (:type backward)
  (:lab-data partner)
  (:locals cost deduction)
  (:domains tax-domain)
  (:instantiate-when
        (:ask-if (eq filing-status ':single)))
  (:instantiate-ask-first
        "Could a marriage be considered? ")
  (:instantiate-ask-rest
        "Could any other marriages be considered? "))
```

This is an example of a backward packet. It has a lab-data variable **partner**, two local variables **cost** and **deduction**. It is a sub-packet of the domain **tax-domain** (see 2.4.1). During a **tax-domain** session, this packet will be instantiated if **filing-status** equals **single** (**filing-status** is a lab-data variable of the domain) and if the user answers the instantiate question "yes". The first time an instance might be made, the **:instantiate-ask-first** question is used. Once one has been made, the :instantiate-ask-rest question will be used.

2.4 Defining a domain

A domain is a special type of backward-chaining rule packet. It has its own defining form, as follows:

```
(defdomain <domain-name>
    { <domain-description-clause> }* )
```

where

<domain-name> A name for this domain. Should be a symbol.

<domain-description-clause>
A clause defining properties of this domain. See below.

A domain description clause is one of the following:

(:goals { <var> }*)
The arguments will be goal variables of the domain.

(:locals { <var> }*)
The arguments will be local variables of the domain. See the decsription of local variables of backward-chaining rule packets, above.

(:lab-data { <var> }*)
The arguments will be lab-data variables of the domain. See the description of lab-data variables of backward-chaining rule packets, above.

2.4.1 Domain definition examples

```
(defdomain tax-domain
  (:lab-data filing-status)
  (:goals action-list))

(defrulevar tax-domain filing-status
  prompt "What is your filing status? "
  type symbol
  choices ((single "single") (married "married"))
  list-choices t)

(defrulevar tax-domain action-list
  trans
   ("~2%Action~P to reduce tax liability:~{'%6X~A~}"
     (1- (length action-list)) action-list))
```

This defines a domain called **tax-domain** with one lab-data variable (i.e., a variable which the user will give a value before any inference takes place) **filing-status**, and one goal variable **action-list**. A given **tax-domain** session thus is an effort to use rules to infer a value for **action-list**.

The variable **filing-status** then has some properties set, in particular the prompt for asking the user for a value, the type of the value and the legal values for this variable (since the type was **symbol**). Since **list-choices** is t, the set of legal values will be printed with the prompt when the user is asked. The set of legal values is expressed as a list of two-element lists. Each sub-list corresponds to one legal value. The first element of a sub-list is the value returned if the second element is typed in (typically, but not always, these two are the same).

The goal variable **action-list** has the **trans** property set. This property expresses how the value of **action-list** will be printed at the end of a session. The list is passed as arguments to **format** to do the printing.

```
(defdomain soybeans
  (:goals diagnosis)
  (:locals
        time-of-occurence
        precipitation
        canker-lesion-color))
```

Another simple example of a domain.

3. Rules

3.1 Types of rules

There are three types of rules: basic, antecedent, and one-shot. The behavior of a rule and its invocation are determined by the type.

3.1.1 Basic rules

A basic rule (the default type) is invoked by the packet of which it is a member. It fires if its condition part is satisfied. For a rule which does not use uncertainty, satisfaction simply means that all of the rule's condition forms evaluate to non-nil values. Rules that use uncertainty have more complicated satisfaction requirements (see 3.3 for an explanation).

3.1.2 Antecedent rules

Antecedent rules are a special sort of rule that should only be members of backward-chaining rule packets (see 2.2). An antecedent rule has the same behavior as a basic rule. However, it is not directly invoked by the owning rule packet. Instead it is invoked anytime some other (non-antecedent) rule of the owning packet (or of some sub-packet of the owning packet) fires. Similarly, if a packet variable of the owning packet (or of some sub-packet of the owning packet) is set through interaction with a user, the antecedent rules are invoked.

3.1.3 One-shot rules

One-shot rules are invoked by the owning packet, as with basic rules. Once it fires, however, it will not fire again (in fact, the conditions will not even be looked at) until it has been reset.

3.2 Defining a rule

A rule definition looks like this:

```
(defrule <rule-name> <rule-packet-name>
    { <description-clause> }*
    { <condition-list> }*
    (:actions { <action-form> }*)  )
```

where

<rule-name> A name to use for this rule (or * to have the system generate a name). Should be a symbol.

<rule-packet-name>
The name of the packet that this rule is to be a member of. Should be a symbol.

<description-clause>
A symbol or a Lisp form which states some property of this rule (see 3.2.1 for details).

<condition-list> A set of conditions which must be satisfied for this rule to fire. A condition-list takes the following form:

```
(<condition-coefficient> { <condition-form> }*)
```

where

<condition-coefficient>
One of: **:conditions**, **:sufficient-conditions**, **:confirming-conditions**, or **:condition-coefficient**. In the last case the first condition-form is the coefficient and should be a number between 0 and 1. See 3.3 for explanations of these values.

<condition-form> is a Lisp expression. See 3.3.2 for some special expressions that might appear here in the case that uncertainty is being used for this rule.

<action-form> A Lisp expression which is evaluated if the rule fires. See 3.3.2 for details on the sorts of forms which should appear here in the case that uncertainty is being used for this rule.

3.2.1 Rule description clauses

Rules have various properties which may be set by providing rule description clauses in the rule definition. Each property has an associated keyword, and may also take arguments. The general form of a rule description clause is a list whose car is the relevant keyword and whose cdr is a list of arguments. In the case that a property takes no arguments, the description clause is a list just containing the relevant

keyword; in this case the keyword itself may be used as the clause (i.e., the symbol instead of a list containing the symbol).

The defined keywords and their arguments are as follows:

:type — One argument, a symbol, which is the name of the type of this rule. The possible values here are **antecedent**, **one-shot**, or **basic**. If no type clause appears in a rule definition, the rule is assumed to be **basic**. See 3.1 for descriptions of the different types of rules.

:use-uncertainty — One argument which states whether or not this rule should use uncertainty (see 3.3). If this clause is not provided, the rule will not use uncertainty.

:strength — One argument, a number, which is the strength of the action part of the rule. This is only relevant for rules which use uncertainty. See 3.3.

:documentation — One argument, a string, which is stored as descriptive information for the rule.

:updated-vars — Arguments are the variables updated by this rule, i.e., any variable which would be modified if this rule fires. This is only relevant for rules which are members of backward-chaining rule packets (see 2.2).

3.2.2 Rule definition examples

```
(defrule fore-rule-x fore
    (eq x a) -> (setq b "x"))
```

This is a rule of the **fore** packet (see 2.3.2). This rule does not use uncertainty. Therefore it will fire if **x**, an argument of the packet, is eq to **a**, a local variable of the packet.

```
(defrule D1 soybeans
  (:updated-vars diagnosis)
  (:use-uncertainty T)
  (:sufficient-conditions
    #[time-of-occurence = {august .. september}]
    #[precipitation >= normal])
  (:confirming-conditions
    #[canker-lesion-color = brown])
  (:actions
    #[diagnosis <- "Diaporthe stem canker"]))
```

This is a rule of the **soybeans** domain. If it fires, it will modify the value of **diagnosis** (a goal variable of the domain -- see 2.4.1). It is a rule which uses uncertainty. There are three conditions, two of which have a high valued coefficient (that is, they are "**sufficient**"), and one of which has a lower valued coefficient (it is "confirmatory", but

not sufficient). The form of the condition clauses is the **selector**. See 3.3.2. Note that { and } are not meta-symbols but are literally part of the rule.

3.3 Uncertainty

FLEX has the ability to deal with uncertain values and inferences. The mechanism which is used is derived from <reference to Michalski here>. In brief, it works as follows.

Every packet variable which has a value also has a measure of the certainty of that value. There are two ways in which this certainty measure can be set, depending on which of the two ways were used to get the value. If the user was asked for the value, then s/he also would be asked to provide the certainty. If the value was inferred, then the certainty is derived from a function of three arguments: the certainty of the assignment (3.3.2 explains how this is expressed); the certainty of the condition part of the rule whose action part is doing the assignment; and the strength of the rule (set by the :**strength** property of the rule).

The certainty of the condition part of a rule is derived from a function of the certainty of each clause modified by the condition coefficient associated with the clause (this coefficient is set in the rule definition through the condition-list construct -- :**conditions** have the highest possible coefficient (i.e., 1.0), :sufficient-conditions have a relatively high coefficient, :**confirming-conditions** have a significantly lower coefficient; other coefficients can be specified with the :condition-coefficient form). The certainty of a clause, in turn, is determined in part by the certainty of any variables in the clause, and in part by the function which the evaluation of the clause invokes. See 3.3.2 for details about the construction of uncertain condition clauses.

The certainty of the condition part of a rule also constrains the firing of the rule. Rather than firing if all condition clauses are true, as a rule which does not use uncertainty would, a rule which uses uncertainty fires if the certainty of the condition part is greater than some threshold.

3.3.1 Uncertain values and sets

If a value for a variable is not certain, then the variable may in fact have several distinct possible values, each with its own certainty measure. Such a variable is said to be **multi-valued**. Anytime such a multi-valued variable is mentioned in a selector (see 3.3.2), all possible values are tried (i.e., an implicit **or** of the possible values). If the variable is mentioned in some context other than a selector, the most certain value will be used (this is referred to as the **principal value** of the variable). When a variable is uncertainly assigned a value, this value and its certainty are added to the list of values so far; if the value is already a possible value, the two certainty measures are combined.

If a variable can simultaneously have more than one value (as distinct from several possible but mutually exclusive values, as described above) then the variable is said to be **set-valued**. That is, it has one value which is a set of value/certainty pairs. When such a variable is used in a selector, all of its values must be satisfied (i.e., an implicit **and** of the set of values). When a set-valued variable is uncertainly assigned a value, this value and its certainty are added to the set; if the value is already a member of the set, the two certainties are combined. Set-valued variables can't meaningfully be used in contexts other than selectors. They can be used in domains which don't use uncertainty.

Any variable whose type is a set-type is set-valued; all other variables are multi-valued.

3.3.2 Expressions involving uncertainty

The basic form for building expressions which use uncertainty is the **selector**. A selector is a function which applies a relational operator to a variable and each of the elements of some subset of the domain of the variable. The most general form of a selector is:

```
#[ <variable> <relation> <sub-domain> ]
```

In this and other examples in this section, the symbols [,], { and } are literals of the

rule (not meta-symbols as they are elsewhere in this document). The return value of a selector is (essentially) a function of the certainty of the value of <variable> and the weight (i.e., a relative significance measure) of the value of <variable>.

There are two classes of selector, depending on whether the <sub-domain> argument is nil or not. If it is not nil, then the arguments should be interpreted as follows:

<variable> A symbol.

<relation> A predicate of two arguments (typically something like =, >, <= etc.).

<sub-domain> A construct which specifies a set of possible values for <variable> and perhaps measures of significance (called **weights**) for each of those possibilities. The selector will work by applying <relation> to <variable> and each of the possible values specified by <sub-domain>, returning the maximum weight.

The form of this argument is one of the following:

- a simple value or a range of values; these are called **unweighted values**. A range is written:

  ```
  {<lower-bound> .. <upper-bound>}
  ```

- a list of of unweighted values.

- an unweighted value with a weight; these are called **weighted values**. A weighted value is written:

  ```
  <unweighted-value>@<weight>
  ```

- a list of weighted values.

Some examples:

```
Unweighted values:

#[temperature >= normal]

#[time-of-occurence = {may .. august}]
```

The first of these applies >= to the value of **temperature** and **normal**. The second applies = to **time-of-occurence** and the range May through August.

```
List of weighted values:

#[precipitation = {normal@0.7
                   above-normal@1.0
                   below-normal@0.3}]
```

In this example, the value of the selector will be a function of the certainty of

precipitation and the weight as selected by the value of **precipitation**, i.e., 0.7 if the value = **normal**, 1.0 if the value = **above-normal**, and 0.3 if the value = **below-normal**.

If the <sub-domain> argument is nil, the the <relation> arguments is interpreted differently. It is considered to be a **weight function**, i.e. a function of one argument (the variable from the selector) which returns a weight.

```
(defun soybean-YR1 (years)
  (cond ((>= years 3)
         1.0)
        ((eq years 2)
         0.8)
        (t
         0.2)))

#[years-crop-repeated soybean-YR1]
```

The selector will be evaluated by calling **soybean-YR1** with **years-crop-repeated** as an argument. The value of the selector will be a function of the weight returned by **soybean-YR1** and the certainty of the current value of **years-crop-repeated**. This selector may also be written as:

```
#[soybean-YR1(years-crop-repeated)]
```

which emphasizes its functional nature.

A selector may also be used to uncertainly assign a value as follows:

```
#[ <variable> <- <value> ]
```

where <value> can be weighted. The meaning of this kind of assignment depends on whether the variable is multi-valued or set-valued. See 3.3.1 for details.

Uncertain forms can be built up recursively with the following functions:

(| &rest disjuncts)
: where disjuncts are each uncertain forms. This is an uncertain version of the **or** function.

(& &rest conjuncts)
: where conjuncts are each uncertain forms.This is an uncertain version of the **and** function.

(=> if-part the-part)
: where if-part and then-part are uncertain forms. This is an uncertain implication function.

4. History and tracing

FLEX provides an ability to trace packet instances and rules as they run, as well as an ability to examine the history of packet instance and rule invocation after a run. This chapter describes these two facilities. It should be noted that, in the current implementation, forward-chaining packet instances which have been expanded can no longer be traced and will no longer appear in history trees.

4.1 Tracing

If a rule is traced, then the following information will be printed (usually to **standard-output**) anytime that rule is invoked:

- When the rule is entered, its name will be printed.
- As each condition clause is evaluated, the clause and its value are printed (as well as the certainty factor).
- Finally, if the rule fires, each action clause is printed and the ultimate value of the rule is printed. If the rule doesn't fire, this fact is printed. Certainty information is printed as well.

If a packet instance is traced, then all of its rules are automatically traced. Also, the following is printed anytime the packet instance is invoked:

- When the packet instance is entered, its name, its arguments and their values, and its local variables and their values are printed. Values are printed with certainty measures.
- When the packet instance is exited, the resulting value is printed.

If a packet is traced, then all instances of this packet (hence all rules of the packet) are automatically traced.

The functions which turn tracing on and off are:

(rule-packet-trace &rest packets)
: This macro will cause all of the named packets or packet instances to be traced. If no packets are specified, then all defined packets are traced.

(rule-packet-untrace &rest packets)
: This macro will turn off tracing for all of the named packets. If none are given, then all defined packets will have tracing turned off.

(rule-trace &rest rules)
This macro will turn on tracing for all of the named rules, or all defined rules if none are specified.

(rule-untrace &rest rules)
This macro will turn off tracing for all of the named rules, or all defined rules if none are specified.

(tracing-packet form &optional path)
This macro assumes that **form** is an invocation of a packet instance. It will turn on tracing of that packet instance, evaluate the invocation, then turn the tracing off. If **path** is supplied, the trace output will go to the file which it names rather than to standard-output.

(with-trace-file path form &rest packets)
This macro will trace the packets specified by **packets**, evaluate **form**, then untrace the packets. The trace output will go to the file named by **path**.

The variable **$rule-output-stream** can be used to send trace information somewhere other than **standard-output**, simply by setting it to some other stream. With-trace-file and **tracing-packet** use this functionality.

4.2 History

The same general information which is printed via tracing is also stored for later examination. The function **history** does this examination.

```
(history &rest packets)
```

The latest history for each of the given packet instances is printed. If no packets are given, then the history of the most recently invoked packet instance is printed.