PSRL: An SRL-Based Production Rule System

Reference Manual for PSRL Version 1.2

Michael D. Rychener

CMU-RI-TR-85-7

Intelligent Systems Laboratory
The Robotics Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

December 1984

This research has been supported by Digital
Equipment Corporation.

Table of Contents

List of Figures

Abstract

PSRL is a production-rule interpreter based on the primitives and knowledge representation of the SRL/1.5 language. It adds a procedural capability to SRL's declarative features, and may become the basis for a complete control environment for SRL users. The current version includes a facility for converting OPS5 rules into PSRL schemata. They can then be run with an SRL database context as the working memory. Many OPS5 debugging and tracing functions have been implemented in PSRL. Being based on SRL gives PSRL advantages over OPS5 in allowing relations among data elements, user-defined inheritance of values, default values, automatic type-restriction testing on values, and the availability of meta-information on schemata, slots and values. In addition, the user has SRL demons, a large database capability and SRL contexts for considering alternative models. PSRL encourages hierarchical organization of small modular sets of rules, each with its own control strategy. SRL demons can be used to control the execution of rule sets. PSRL is implemented so that its facilities can be reconfigured in the service of a variety of problem-solving architectures.

Acknowledgments

Mark Fox, Pat Langley, and Mark Wright were co-authors of the Preliminary Design of PSRL, and this paper borrows from some sections of that document. Digital Equipment Corporation is the primary supporter of this research. Michael Greenberg wrote the first version of the PSRL matcher code. Elaine Kant provided useful comments on the introductory overview of PSRL. The users whose projects are mentioned here have provided valuable feedback on the system, well before the manual was finished; in particular, Mary Lou Maher, Joe Mattis and Arvind Sathi have helped to make PSRL more useful. The last three have also supplied written descriptions that were used to construct the chapter on applications. I am grateful to Mary Lou Maher, Arvind Sathi, Duvvuru Sriram and Stephen Smith for many helpful comments on the manual. Jill Fain, Duvvuru Sriram, Mark Fox, and Arvind Sathi provided crucial assistance in the comparisons of PSRL to ROSIE and LOOPS.

1. Introduction and Summary

PSRL combines the production system approach of data-directed control [13, 14, 5, 15] with the features of SRL, the Schema Representation Language of Fox, et al [8, 16]. SRL is a Lisp-based declarative formalism for representing complex, structured objects and their interrelations. Rules in PSRL are expressed in an extension of Forgy's OPS5 [6] language syntax, $PSRL_{OPS8}$, and many of the OPS5 user commands are available, so that the running environment of PSRL is similar to that of OPS5.[1] Since rules are translated into schemata, and operate with respect to schemata, the PSRL environment also includes the full set of SRL commands and utilities. PSRL is much less efficient than OPS5 (it is currently run interpretively, with an optimizing compiler still being planned), so that it is impractical to deal with large rule sets. Instead of executing rules in large sets, they are partitioned into small packets that are stored in schemata, organized using SRL relations, and triggered by SRL demons.

Given the variety of ways of operating within SRL (including a version of OPS5 with an interface for transferring data to and from SRL schemata), PSRL seems most appropriate for monitoring and manipulations that involve bringing together and combining information in an associative, pattern-directed way, from a number of schemata. PSRL is less appropriate for single-schema operations and operations where specific schemata (as opposed to classes or sets with similar properties) are used. It is currently being used in a high-rise building design system, in a project management system with heuristic rules, and in a knowledge-based graphics system (these are described in more detail below).

1.1. Data Memory

PSRL rules access and manipulate schemata within the scope of entire SRL database contexts, inheriting considerable representational power from SRL. An SRL user can define new relations among objects, allowing various kinds of information to be inherited automatically and according to user-defined constraints. SRL contains facilities for automatically checking the validity of values, and for executing procedures (demons) as a result of schema operations. The SRL database system provides automatic swapping of schemata to disk files, so that very large collections of them are manageable. Thus PSRL rules can deal with a much larger and more permanent database than is possible in OPS5. Databases are usually built up by other means, with PSRL applied to the resulting mass of data to perform bookkeeping, maintenance, and communication operations.

1.2. Production Memory

PSRL interprets rules whose syntax closely resembles that of OPS5. A translator takes the OPS5-like rules and builds schemata to represent them internally. Rules are grouped into sets, and stored in "production-system" schemata, whose names can be assigned to global variables or stored inside a user's schemata and demons. Features have been added to extend the syntax beyond OPS5: allowing specific, named schemata to be matched; allowing schemata to specify direct relationships to other schemata (OPS5 only allows indirect pointers); and allowing restrictions to be placed on class names and on element variables. Element variables (the ones in braces surrounding condition elements) can be used freely inside elements, for matching and modification purposes. The user can specify that inheritance of values is to be allowed in matching and that matching is to be done with respect to particular contexts within an SRL database.

---

[1] OPS5 is well known; OPS6 existed as a design only; OPS7 is currently only an experimental implementation in Pascal, for the PERQ.

A direct link to Lisp functions is provided in the 'eval' action function, which takes a Lisp S-expression, substitutes values for OPS variables, and then evaluates it. The OPS5 Lisp interface is not implemented, since SRL functions provide equivalent power, in combination with the eval action function. Demons can be attached to classes of schemata to provide another sort of procedural escape. Using either the eval function or demons, a user can readily have rule-sets executed recursively from within the actions of rules.

1.3. Control

Rule sets can be invoked by demons attached to schemata, or by Lisp function calls. Each rule set has associated with it a control function that determines what style of conflict resolution is to be used during its execution. A number of options are supplied, and the user can code others, as needed. Some examples of existing control functions are: cycling repeatedly through a list, going back to the beginning after each successful rule firing; sequencing through a list, executing all possible matches of each rule, then going on to the next; stopping execution after one rule has fired; and cycling through the list of productions in reverse order.

1.4. The Interpreter

Most of the tracing and debugging facilities available in OPS5 are also provided to PSRL users. In addition, SRL-proficient users can attach special-purpose demon functions for monitoring and tracing purposes, as needed.

1.5. Extensions

Many problem-solving architectures other than the basic production system recognize-act cycle are possible, using the basic modules of PSRL. PSRL also compares favorably with other similar systems such as LOOPS [1] and ROSIE [4], especially in regard to making full use of the power of SRL.

1.6. Prerequisites

Some familiarity with the OPS5 language [6] will be helpful, but not required, here. The basic prerequisite is familiarity with AI programming, pattern matching and production systems [2, 3, 15], and with the SRL/1.5 language and concepts [16]. Examples, especially the extended one in Chapter 3, are intended to be helpful to production-system novices in reviewing the central concepts.

1.7. Manual Organization

Chapter 2 discusses PSRL from the standpoint of its representation as SRL schemata. Most system aspects are described declaratively, and the schemata precisely delineate the full power of expressions in PSRL. Dealing exclusively with PSRL at the SRL level, however, seems cumbersome in comparison to the $PSRL_{OPS8}$ notational level, as described in the later chapters. Chapter 3 introduces the $PSRL_{OPS8}$ notation, first presenting an extended example of how an OPS8 rule is represented as SRL schemata, and then proceeding with full details on various declarations and on the options available for expressing rules in OPS8. Chapter 4 gives details on the user commands available in PSRL. The first few sections fill in details on loading and running the system, and on setting up the control of rule set execution. Then come details on tracing, debugging and user interface.

Chapter 5 briefly sketches several developing systems that are using PSRL, in order to illustrate some of the possibilities. It also compares PSRL to several other rule-based representation systems. Chapter 6 indicates some directions for further development of PSRL.

Chapter 7 contains several illustrative runs of PSRL, showing examples of execution of most of the important PSRL commands. These appendices are intended to present selected features of the system in a natural user-oriented ordering, in contrast to the main chapters whose organization is along logical, structural, or taxonomic lines.

2. Overview of PSRL Representation

The main concepts involved in production systems are:

- Production system: a set of rules that are considered together as a procedure for performing a data-directed computation;
- Production, or Rule: a condition-action (or IF-THEN) pair, specifying a data condition under which the action of the rule is performed;
- Condition, or left-hand-side (LHS): a pattern or abstraction specifying a data state or a class of possible data states;
- Action, or right-hand-side (RHS): a sequence of data manipulations to be performed;
- Architecture: the combination of a pattern matcher for conditions, procedures for executing actions, a conflict resolution or other control procedure, and various parameters that affect the operation of the interpreter; and
- Environment: the combination of architecture, production system, working memory, and other database parameters that are involved with the execution of a production system.

In PSRL, SRL schemata are used to specify all of the above. The current implementation does not make use of the last two schemata while it is running, since only one value is available for each slot. Later versions, however, are expected to use them for a variety of special-purpose configurations.

This chapter will sketch the concepts of PSRL at the SRL level. The progression is generally from high-level schemata, such as production system architecture, to low-level ones, such as templates. As mentioned above, users of PSRL will probably prefer to do most manipulation at the $PSRL_{OPS8}$ level (see Chapter 3), due to its greater convenience - many schemata can be involved in the SRL definition of even a single rule, making it cumbersome to work at that level. So the purpose of this chapter is to provide some insight as to how rules and systems can be represented internally, as a basis for understanding how the $PSRL_{OPS8}$ notation is interpreted. Some readers may prefer to skim this and refer back to it at later points for clarification.

2.1. Production System Architecture

The slots of a **ps-architecture** schema, Figure 2-1,[2] specify the functions that the PSRL interpreter should use to interpret rules according to that architecture. The LHS-MATCHER (left-hand-side-matcher) is applied to the 'if' or 'condition' parts of rules to determine which conditions are satisfied by the current state of the database. The RHS-EXECUTER is a function to perform the right-hand-side actions of rules. The CONFLICT-RESOLUTION slot has a function for choosing among the set of matched rules, for a subset of rules whose actions are to be performed. The repeated application of the functions specified in the **ps-architecture** as just described is the traditional recognize-act cycle of production systems.

---

[2]The fonts used here are as in other SRL reports: **bold** for schema names, SMALL CAPITALS for slot names, and *italics* for facet names.

```
{{ ps-architecture
      LHS-MATCHER:
      RHS-EXECUTER:
      CONFLICT-RESOLUTION: }}
```

Figure 2-1: ps-architecture Schema

As mentioned above, the current $PSRL_{OPS8}$ interpreter does not make use of the **ps-architecture** schema, but simply operates according to **psrl-ops8**, shown in Figure 2-2. The above description of the recognize-act cycle is only an approximation, as regards the current implementation.

```
{{ psrl-ops8
      IS-A: "ps-architecture"
      LHS-MATCHER: fire-m
      RHS-EXECUTER: fire-x
      CONFLICT-RESOLUTION: update-conflicts}}
```

Figure 2-2: psrl-ops8 Schema

Each set of PSRL rules (each 'method' or 'PS module') is organized by a schema that is an instance of the one shown in Figure 2-3.

```
{{ production-system
      RULE-SET:
          restriction: (set (or (type "instance" "rule")
                                (type "instance" "production-system")))
      NUMBERS:
      CONTROL: }}
```

Figure 2-3: production-system Schema

The value of the RULE-SET slot names the rules that are contained in the specific system; the set of elements is taken to be ordered. If an element of a RULE-SET is a production system rather than a rule, then that production system is interpreted (its entire RULE-SET is processed according to its CONTROL slot) whenever a rule would be, in that position in the set. The NUMBERS slot specifies numerical parameters controlling the interpretation of the rules in the **production-system**, as in the Langley's PRISM system [9]. It is unused at present. The CONTROL slot specifies a function for the recognize-act cycle for the given **production-system**. A number of different possibilities are currently available, described in Section 4.6. A detailed example of a production system is given in Section 3.1.

2.2. Multiple environments for production system execution

PSRL will eventually allow multiple environments in which rules are evaluated (Figure 2-4). An environment would include all the 'dynamic' aspects of PS execution, in contrast to production-system, rule, etc., which define relatively 'static' aspects. In fact, the environment is the only place that such dynamic information is placed, in order to cleanly separate it from rules and other static entities. Think of rule sets as functions, while environments specify the data to which the functions are applied - there could obviously be more than one such application going on at the same time in a computation. Thus, separate activations of the same set of production rules would be specified by different environments. Each environment would maintain its own rule set, in its **production-system** schema, and its own conflict set. The user would define environments, activate and deactivate them, and delete them. (A later version may allow parallel execution.) In the current version of $PSRL_{OPS8}$, only one environment is active, and it is specified internally rather than using a schema.

{{ **environment**
PS-ARCHITECTURE:
PRODUCTION-SYSTEM:
CONTEXT:
AGENDA:
STATUS: }}

Figure 2-4: environment Schema

The slots in an environment would indicate how one would start up and run a PS. One slot would contain an instance of the **ps-architecture** schema, described above. Another would contain the name of a production-system, which includes a set of rules. The CONTEXT slot would indicate the SRL database context in which interpretation of the rules would take place.[3] The AGENDA and STATUS slots would hold other information regarding the dynamic execution within an environment, namely currently uncompleted actions within the rule-set (used, for example, when a rule-set's execution is pre-empted or suspended by that of another rule-set) and the current execution status (e.g., active, suspended, or completed).

2.3. Rule Structure

A rule is specified as an instance of the **rule** schema, Figure 2-5. The PSRL-LHS slot contains a set of elements that are interpreted as conditions, being instances of **lhs-element**. The PSRL-RHS slot contains a set of elements, instances of **rhs-element**, that are interpreted as actions to be executed - an "<action>" is one of a set of predefined schemata (including user-defined ones). Details on these appear in the next sections. A third slot is added by the current interpreter, PSRL-VARIABLES, whose value is a list of the match variables in the rule, alphabetized.

---

[3] In version 1.3, it will be the name of the working memory context, as described in the $PSRL-CONTEXT global variable, described below; rules will be stored in contexts according to the variable $PSRL-RULE-CONTEXT.

```
{{ rule
    PSRL-LHS:
        restriction: (set (type "instance" "lhs-element"))
    PSRL-RHS:
        restriction: (set (type "psrl-action" "<action>")) }}
```

Figure 2-5: rule Schema

2.4. Left-hand-side element schemata

The PSRL-LHS slot of a **rule** is filled by an ordered set of instances of the **lhs-element** schema (Figure 2-6).

```
{{ lhs-element
    TEMPLATE:
    NEGATION:
        restriction: (or t nil)
        default: nil
    SCHEMA-VARIABLE:
    INDEX:
        restriction: (or t nil)
        default: t
    MATCH-SET: }}
```

Figure 2-6: lhs-element Schema

The TEMPLATE slot contains a schema that is to be matched against all schemata in the current PSRL context. Slots and values in a template act as a pattern, successfully matching all schemata that have similar slots and values, along with others not mentioned in the template. The NEGATION slot can reverse the logical sense of the match, if filled with a 't' value: the lhs-element will succeed in matching whenever the template does NOT have any matches, and vice versa. When a match succeeds, the schemata that matched are bound to the value of the schema-variable slot, if it is an instance of **match-variable** (described in Section 2.5). That binding can then be used in later templates within the same left-hand-side. If the variable was bound previously, its binding, a schema name, is checked to ensure that it matches the present template. This allows a schema to be obtained in a relational slot in one schema and then matched in detail in a later template, e.g., to force it to have some further properties. Other match features exist, and will be described in Section 3.4. (The example in Section 3.1 may also be helpful.)

The INDEX slot, unused at present, may be used in a future PSRL version to improve matching efficiency, by enabling the recording of possible matches. The list of matching schemata would be stored in the MATCH-SET slot.

A template is an image of a schema, with variables allowed in some or all of its slot values, and

with different restrictions, defaults, etc. specifiable on the slots. A sketch of a template is given in 2-7. In real applications, the "<xxx>" schema names would be replaced by names of actual corresponding types of schemata. Templates, for instance, have unique names generated by the $PSRL_{OPS8}$ translator.

```
{{ <template>
    TEMPLATE+INV: "<lhs-element-containing-this>"
    PSRL-INSTANCE: "<element-class-to-be-matched>"
    <SLOT1>: "<match-variable>"
        range: <range-spec>
    <SLOT2>: <value>
    etc.}}
```

Figure 2-7: <template> Schema

A relation must be included that links a template with the class of schemata that it can match; the default relation to serve this purpose is **psrl-instance**, shown in 2-8. (The user can change the relation to be used, by setting the variable $psrl-instance, as described in Section 4.5.) The **psrl-instance** relation has a limited transitivity and is used internally in PSRL in such a way as to limit the search paths associated with it, in the hope of making **psrl-instance** efficient enough to be used in its role as a syntactic relation. The <element-class-to-be-matched> in a template can be an existing SRL schema, or it can be specified with the $PSRL_{OPS8}$ literalize declaration.

```
{{ psrl-instance
    IS-A: "relation"
    INCLUSION: "is-a-inclusion-spec"
    FUNCTION: is-a-fn
    TRANSITIVITY: (step "psrl-instance" all t)
    INVERSE: "psrl-instance+inv"}}
```

Figure 2-8: psrl-instance Schema

Two important features that templates may have are: a <range-spec> that restricts the values to be matched in the slot; and a <match-variable> to be bound to the matched value, or whose binding (if it is already bound) serves to restrict any further bindings. A slot with empty value specifies to the matcher that the slot must be defined, but may not yet have a value. The presence of a <match-variable> or other expression or <range-spec> implies that the slot must have a value in order to match.

A schema matches a template if the schema has all of the relations specified in the template and if values of other slots in the template match the corresponding ones in the schema. Values match if they are identical or if a variable from the template is either unbound (it will be bound after matching) or bound to the identical value. If there is a *range* attached to any of the slots, those must also be satisfied by the value in the schema (which is the value bound to the match variable, if any). There is a special value, '{}', that allows a slot in a template to match any value at all in the schema, including the case where a slot is defined but has no value.

The process of matching and the ways of specifying rule elements will be clarified in later chapters, by way of examples (see, e.g., Sections 3.1 and 3.4).

A *range-spec* is an expression specified by:[4]

```
range-spec          ::= (AND range-spec*)
                    ::= (OR range-spec+)
                    ::= (NOT range-spec+)
                    ::= (QUOTE atomic-value)
                    ::= (= atomic-value)
                    ::= (<> atomic-value)
                    ::= (<=> atomic-value)
                    ::= (< atomic-value)
                    ::= (> atomic-value)
                    ::= (<= atomic-value)
                    ::= (>= atomic-value)
                    ::= (FUNCTION Lisp-function-name)
                    ::= (FUNCTION Lisp-lambda-expression)
                    ::= (TYPE schema-name schema-name)
                    ::= atomic-value

atomic-value        ::= constant-symbolic-atom
                    ::= variable
                    ::= number

schema-name         ::= SRL/1.5-double-quoted-schema-name

constant-symbolic-atom  ::= SRL/1.5-schema-name-without-double-quotes
```

These terms are not defined formally here: *Lisp-function-name*, *Lisp-lambda-expression*, *number*, *variable*, *SRL/1.5-double-quoted-schema-name* and *SRL/1.5-schema-name-without-double-quotes*. A *number* is a Franz Lisp or Common Lisp number. A *variable* is an atom beginning with '<' and ending with '>', as in OPS5.

Range-spec expressions serve the same purpose as corresponding elements of the OPS5 language, testing matched values with various logical (AND, OR, NOT) predicates, as in SRL restrictions, and arithmetic (=, <>, <=>, <, >, <=, >=) predicates as in OPS5, which are described more fully in Section 3.4.2. When match-variables occur in the expressions, their bound values are substituted before the predicates are evaluated, except inside a QUOTE list. The FUNCTION option allows a one-argument function to be evaluated, given as its argument the current value being matched. If the function returns a non-nil value, the match succeeds and continues. (There are user-callable functions for accessing values of match-variables, as described in Section 3.4.2, so that predicates can obtain other arguments from the matching context.) The TYPE option uses the SRL function **r-test**.

---

[4] These grammar conventions (cf. OPS5 manual) are used here: an *italics typefont* is used for non-terminals in the grammar; **this typefont** is used for literal symbols; superscript asterisk (*) means 0 or more of the item; superscript plus (+) means one or more.

2.5. Match variable schema

Schemata that are to be considered variables during pattern matching must be instances of the **match-variable** schema, Figure 2-9. (There is a function to declare such schemata, described in Section 4.11, but $PSRL_{OPS8}$ uses its grammar to declare most variables implicitly, so the user doesn't usually need to be concerned about declaring them.) The VALUE slot is used to store a variable's binding. The TYPE slot is unused, but may be used later for specifying other semantics of variable matching. More on the process of matching and the use of variables is given in Section 3.1. See also Section 3.4.

{{ **match-variable**
VALUE:
TYPE: }}

Figure 2-9: match-variable Schema

2.6. Right-hand-side element schemata

The PSRL-RHS slot of a **rule** is filled by an ordered set of instances of the **rhs-element** schema (Figure 2-10). The PSRL-ACTION slot has the name of the action to be done, e.g., "make" or "modify". There is a predefined set of such 'generic' actions that can fill the PSRL-ACTION slot, and the user can define others by creating corresponding schemata. Many actions include a PSRL-INSTANCE slot for various purposes, usually to name a schema class. The PSRL-INTERP-FN slot names a function used to interpret the action - interp-action interprets an arbitrary action (by dispatching to other interpretation functions).

{{ **rhs-element**
PSRL-ACTION:
PSRL-INSTANCE:
PSRL-INTERP-FN: interp-action}}

Figure 2-10: rhs-element Schema

An example of a generic action schema is in Figure 2-11. It has two of the slots described for the **rhs-element** schema, and one new one, PSRL-MAKE-NAME, which indicates the name of the schema to be made (an option of the $PSRL_{OPS8}$ syntax, to be described in Section 3.5.1).

{{ **make**
PSRL-INSTANCE:
PSRL-MAKE-NAME:
PSRL-INTERP-FN: interp-make}}

Figure 2-11: make Schema

Notice that conditions and actions in PSRL are not parallel in their definitions. Conditions are two-level structures (lhs-element and template), while actions are a single level. Actions however must fit into a set of pre-defined classes, while conditions express patterns on arbitrary schemata.

The full set of actions is given below, in Section 3.5.1.

3. Representing Rules in PSRLOPS8 Notation

The first section of this chapter introduces the syntax of PSRL rules by an extended example that covers the main features. The second and succeeding sections will define formally and semantically the $PSRL_{OPS8}$ notation. Some aspects of the ways that the language can be used, namely those that are parameterized by Lisp global variables, are postponed until Chapter 4, where user commands are discussed. What is focussed on here is the external notation and its default translation and interpretation.

3.1. An example: Mapping from OPS8 to schemata

This section illustrates the structure of PSRL rules and the correspondence to the $PSRL_{OPS8}$ by going through a translation of a $PSRL_{OPS8}$ rule. Along the way, some features of the language syntax are introduced. The practical aspects of how to execute rules are postponed until Chapter 4.

The rule to be used, named **beg** (short for begin), is taken from a simple auto factory simulation written in OPS5. The rule, as it appears in this section, has been augmented slightly with a couple of other features for illustrative purposes. Appendix 7.4 includes a run with this rule and others executing in PSRL. The **production-system** containing the rule is shown in Figure 3-1. This schema is usually created automatically when a rule file is loaded (see Section 4.5), and its slots are filled as rules are defined using 'p' forms and control declarations. These declarations and commands are described in Section 4.5, but this schema can be taken as given for the purposes of this section.

```
{{ autosel
    INSTANCE: "production-system"
    RULE-SET: "buy2" "beg" "clu" "eng" "end"
    CONTROL: fire-sequence}}
```

Figure 3-1: autosel Schema

The **beg** rule, in its OPS8 format, is the following:

```
[p beg
  { (clock ↑mod going ↑timer <t>) <c>}
  - (automac ↑mod busy ↑machine engine)
    (steel ↑what scrap ↑amount > 4)
-->
    (bind <n>)
    (make auto ↑mod order ↑serial <n>)
    (modify <c> ↑timer (compute <t> + 1)) ]
```

This rule recognizes that the clock is running (first condition element) and that there is no currently busy machine making an engine (second condition element). Under these conditions, if there is enough scrap steel (third element), the rule fires, creating an order to manufacture an auto. It also updates the clock's time by adding one (this is an extra action not contained in the original **beg** rule). The rule's schema is in Figure 3-2.

```
{{ beg
    INSTANCE: "rule"
    PSRL-LHS: "begc1" "begc2" "begc3"
    PSRL-RHS: "bega1" "bega2" "bega3"
    PSRL-VARIABLES: "<c>" "<n>" "<t>"}}
```

Figure 3-2: beg Schema

The first condition in the lhs is

{ (clock ↑mod going ↑timer <t>) <c>}

whose translation is shown in Figures 3-3 and 3-4. The first schema (**begc1**) records the element variable, <c>, in the SCHEMA-VARIABLE slot, and points to the second schema (**begc1t**), using the TEMPLATE slot. The second schema has a slot for each of the OPS5 attributes (preceded by '↑'). There is also a PSRL-INSTANCE slot[5] for the class name, 'clock', and an inverse pointer from the template back to the lhs-element schema.

In the process of executing a production system containing the **beg** rule, this condition element will match all schemata in the current SRL database context that are related to the **clock** schema by the **psrl-instance** relation, and that have the atom 'going' in their MOD slot and any value at all in their TIMER slot. After matching, the match-variable <t> will be bound (i.e., will have as the value of its VALUE slot) to the value of the TIMER slot of each matched schema. Also the variable <c> will be bound to the name of the schema matched. If more than one schema matches this first condition element, then the other condition elements in **beg** will be matched in turn using each distinct assignment of values to match-variables.

```
{{ begc1
    INSTANCE: "lhs-element"
    TEMPLATE: "begc1t"
    SCHEMA-VARIABLE: "<c>"}}
```

Figure 3-3: begc1 Schema

[5] As mentioned above, this relation for templates can be changed by the user.

```
{{ begc1t
      TEMPLATE+INV: "begc1"
      PSRL-INSTANCE: "clock"
      MOD: going
      TIMER: "<t>"}}
```

Figure 3-4: begc1t Schema

The second element in the lhs is,

```
- (automac ↑mod busy ↑machine engine)
```

whose translation is in Figures 3-5 and 3-6. The new feature in this element is the negation, indicating that the match will succeed if this element's match fails, i.e., if there are no database schemata matching it. This is indicated by the NEGATION slot with value 't'.

```
{{ begc2
      INSTANCE: "lhs-element"
      TEMPLATE: "begc2t"
      NEGATION: t}}
```

Figure 3-5: begc2 Schema

```
{{ begc2t
      TEMPLATE+INV: "begc2"
      PSRL-INSTANCE: "automac"
      MOD: busy
      MACHINE: engine}}
```

Figure 3-6: begc2t Schema

The third element in the lhs,

```
(steel ↑what scrap ↑amount > 4)
```

has another new feature, the comparison of numerical values. The condition states that the instance of the steel schema to be matched must have an AMOUNT slot value greater than 4, and this expression is stored on the *range* facet of that slot in **begc3t**, Figure 3-8. Notice that the slot AMOUNT has as its value the atom '{}', which in OPS5 and $PSRL_{OPS8}$ will match to any value. In some cases, a user will want to bind the value matched here (e.g., something greater than 4) to a variable, so that the variable name would take the place of the '{}', and the pattern would be expressed as '↑amount {<x> > 4}'.

```
{{ begc3
      INSTANCE: "lhs-element"
      TEMPLATE: "begc3t"}}
```

Figure 3-7: begc3 Schema

```
{{ begc3t
      TEMPLATE+INV: "begc3"
      PSRL-INSTANCE: "steel"
      WHAT: scrap
      AMOUNT: {}
          range: (> 4) }}
```

Figure 3-8: begc3t Schema

The right-hand-side or action part of the **beg** rule starts out,

```
--> (bind <n>)
```

which is an action that binds the variable <n> to a new 'random' schema. (It can bind other values, if they are specified as expressions following the variable name, but in this case a default expression analogous to (gensym) is assumed, since nothing is given.) The schema for this action is in Figure 3-9, and has the action name "bind" as the value of the PSRL-ACTION slot, "<n>" as the value of the VARIABLE slot, and no value for the RHS-TERMS slot. These slot names are specific to the bind action - each $PSRL_{OPS8}$ action has its own particular set of slots, though some share the same ones, and all have the PSRL-ACTION slot naming the particular action involved.

```
{{ bega1
      PSRL-ACTION: "bind"
      VARIABLE: "<n>"
      RHS-TERMS: }}
```

Figure 3-9: bega1 Schema

The second action of **beg** is

```
(make auto ↑mod order ↑serial <n>)
```

whose translation is in Figure 3-10. The make action creates a schema with a gensym'd name that is PSRL-INSTANCE of the value of the PSRL-INSTANCE slot in the action schema (**auto** in this example). Other slots in the action schema specify slots and values to be created in the new schema. When the values are match variables (**<n>** in this case), their bound values from the match or from bind actions are filled in.

```
{{ bega2
    PSRL-ACTION: "make"
    PSRL-INSTANCE: "auto"
    MOD: order
    SERIAL: "<n>"}}
```

Figure 3-10: bega2 Schema

The third action,

```
(modify <c> ↑timer (compute <t> + 1))
```

is another type, **modify**, and is shown in Figure 3-11. The modify action works on an existing schema, specified by the value in the PSRL-INSTANCE slot; in this case it would be the value bound to <c>, i.e., the schema that matched the first lhs-element. The other slots of the modify action schema indicate the slots in the modified schema that are to be changed, and their values specify new values in the changed schema. In this case, there is a compute expression to be evaluated, whose result will become the TIMER slot in the changed schema.

```
{{ bega3
    PSRL-ACTION: "modify"
    PSRL-INSTANCE: "<c>"
    TIMER: (compute "<t>" + 1) }}
```

Figure 3-11: bega3 Schema

3.2. OPS8 language definition overview

As mentioned several times in the above discussion, $PSRL_{OPS8}$ is based directly on OPS5. The grammar presentation below, in particular, is closely modelled on that of OPS5. Thus, for those readers who are familiar with OPS5, there are accompanying summaries of the differences of detail. PSRL is intended to be a compatible extension of OPS5, with the exception of some actions and matching features that are not appropriate in the SRL environment. Thus most of the differences here are new features that have been added.

All aspects of the $PSRL_{OPS8}$ notation are precisely described in a BNF grammar. The following grammar conventions (cf. OPS5 manual) are used here: an *italics typefont* is used for non-terminals in the grammar; **this typefont** is used for literal symbols; superscript asterisk (*) means 0 or more of the item, separated by spaces; superscript plus (+) means one or more.

The following is the top level of $PSRL_{OPS8}$, listing the types of things that can occur in a program.[6]

[6] Some OPS5 constructs are recognized and ignored: strategy external literal

```
OPS8-entity      ::= literalize
                 ::= vector-attribute
                 ::= production-rule
                 ::= production-system-name
                 ::= control
                 ::= ( setq parameter Lisp-expression )
                 ::= ( debugging-command )
                 ::= ( action-command )
                 ::= Lisp-expression
                 ::= comment

comment          ::= ; any-string-terminated-by-newline.
```

All of the above *OPS8-entity*'s except *Lisp-expression* are defined in this chapter and the next. *Lisp-expression* is not defined formally in this manual, and is included to emphasize that $PSRL_{OPS8}$ files are interpreted using the standard Lisp read conventions.

3.3. Working Memory and Production Memory declarations

Working Memory in PSRL is loosely defined: rules access and manipulate an entire SRL database context, and thus the context would correspond to conventional Working Memory (WM). But for debugging purposes, a more restricted definition of WM is used, namely those schemata that have been created by the "make" action (either at the top level or inside rules). During matching, the SRL function **r-find** is used to access schemata, and recall that **r-find** depends on the presence in schemata of SRL relations.

Several features in PSRL make WM more powerful representationally than is the case in OPS5:

- SRL schemata are fundamentally more powerful than OPS5 attribute-value lists; in particular, a wider variety of types is available as values of slots, including lists of values, attachment of meta-information and restrictions, default values, and inherited values; in most cases, the $PSRL_{OPS8}$ notation does not yet exploit the full power of SRL; in particular, the matching of more than the first element of a list value is not yet supported (and its semantics is not understood, due to SRL's own lack of semantics for list values); in some other cases, e.g., inheritance, the power is not obtained notationally but through global variables.

- Relations can directly connect schemata, with a schema name appearing as a value in a slot, whereas in OPS5, WM elements are not named and can point to each other only indirectly through a shared symbol.

- Several WMs can be defined within a single SRL database by using the context facility; rules would match one or the other depending on a global variable to be described below.

- Very large databases are possible, since SRL has automatic swapping of schemata to disk when they exceed available main memory; while efficiency would prohibit PSRL access to huge databases, judicious subdivision into contexts would make handling a large database feasible.

Classes of WM schemata (for use in templates and in actions) are declared, as in OPS5, by a

literalize declaration,

literalize ::= (**literalize** *class-name slot-name**)

Such declarations should be executed before rules and WM-initializing commands are loaded, to be described in more detail in Chapter 4. In this, *class-name* and *slot-name* are Lisp symbolic atoms or numbers. The effect of a literalize is to define a general schema and its slots to represent the class of elements. For instance,

```
(literalize clock mod timer maker)
```

defines the schema in Figure 3-12. Notice that a $PSRL_{OPS8}$ user can modify such schemata after defining them[7] (e.g., adding slots or attaching demons that could be inherited by instances), which is not possible in OPS5. There are some internal side-effects of a literalize, on the class-name's atom (not the schema).

```
{{ clock
    PSRL-INSTANCE: "literalize"
    MOD:
    TIMER:
    MAKER: }}
```

Figure 3-12: clock Schema

Also as in OPS5, a slot whose value is to be treated as a list must be declared with a vector-attribute declaration, as follows:

vector-attribute ::= (**vector-attribute** *slot-name*$^+$)

For example,

```
(vector-attribute items)
 . . .
(literalize checker name balance items)
```

declares the value of the 'items' slot to be considered a list of values. Such a declaration has internal Lisp effects, and no schemata are created, but schemata for the attributes declared are modified with a vector-attribute slot. The order of appearance of vector-attribute and literalize declarations is not significant; the declaration applies to all other declarations, making any use of 'items' (for example) as an attribute in a literalize have the special interpretation. The number of vector-attributes in an element is not restricted to one, as in OPS5.

General note: it is not necessary or correct for a user of $PSRL_{OPS8}$ to put double-quotes (") around names in rules and declarations that are known to be schema names, EXCEPT in the case of ordinary values of slots. Variables are always automatically translated to schemata, as are atoms that appear in other locations inside elements and in declarations. Values of relations are automatically translated to schema names.

Chapter 4 will discuss ways to change the interpretation of WM, along with display and debugging commands.

---

[7] Or they can be set up previously in a user's database, since the literalize declaration will not disturb an existing schema.

The production memory (PM) of $PSRL_{OPS8}$ is structured into a number of sets of rules, each associated with a RULE-SET slot of a **production-system** schema (described above). The name of the production-system schema with which rules are associated is determined by a production-system declaration, defined in the syntax description below.[8]

production-system-name ::= (**production-system** *constant-symbolic-atom*)

control ::= (**control** *constant-symbolic-atom*)

production-rule ::= [**p** *rule-name lhs* --> *rhs*]
::= (**p** *rule-name lhs* --> *rhs*)

lhs ::= *lhs-element*$^+$

rhs ::= *rhs-element*$^+$

The type of control function to be used in interpreting a rule set is established with a control declaration, defined above, which puts a slot in the current production-system's schema (it must come after the production-system-name declaration). Rules themselves are defined with the 'p' form in the above grammar, as in OPS5 (the square-bracket form is an addition to the official OPS5 grammar, being equivalent but slightly more readable). Rules are also associated with the current production-system schema. When multiple production systems are being defined, a new production-system-name supersedes the preceding one, for purposes of storing CONTROL and RULE-SET information. This would be the case, for instance, for a system composed of several rule sets, each in a different file, and with each file containing a production-system-name at its head.

The definitions of *lhs* and *rhs* in the above grammar are also slightly different from OPS5's definitions, in requiring each to contain at least one element.[9] The undefined elements of the above grammar are defined in the next sections. Other aspects of PM, e.g., those dealing with rule-set interpretation and demon-based invocation, will be discussed in the next chapter (see Section 4.6).

3.4. Left-hand-side elements in OPS8

3.4.1. Notations for element forms

A left-hand-side (lhs) is a sequence of lhs-elements, each of which translates to two schemata, one an instance of **lhs-element** and the other a template, as described in Section 2.4. The following syntax defines these elements.

---

[8] A default production-system name is used when rules are loaded using the tlps function, described in Section 4.5.

[9] Empty lhs and rhs might work, but have not been tested, and no commitment is currently made as to whether they work or not; it seems desirable that they would, and future versions will ensure that they do (if they don't already); note that modules of PSRL are accessible in Lisp for performing the equivalent of only conditions or actions.

lhs-element	::=	*template-form*
	::=	- *template-form*
	::=	{ *restriction template-form* }
	::=	{ *template-form restriction* }
	::=	{ *restriction* }
template-form	::=	(*lhs-term**)
	::=	(*lhs-value lhs-term**)
lhs-term	::=	↑ *constant-symbolic-atom lhs-value*
lhs-value	::=	{ *restriction** }
	::=	*restriction*

The template schema is represented notationally by a parenthesized list containing a class name or other relation at the head, followed by slot-value pairs, where slots are preceded by the '↑' character. Entities outside the parentheses of the template have an effect on the lhs-element instance and not on the template. These include negation, denoted by '-', and restriction, delimited by braces '{ ... }'. The most frequent restriction is one containing only a variable, which is then placed in the SCHEMA-VARIABLE slot of the lhs-element instance, and is bound during matching to the schema that matches the template. A schema-p test is done on values of schema-variables, if they are already bound, so that this is an implicit restriction. If a predicate expression does occur inside the braces, it is placed as a *range* facet of the SCHEMA-VARIABLE slot, and is taken to apply to the value bound to the schema variable. That is, the restriction expresses relationships among schemata, the most useful of which are equal (=) and not-equal (<>), forcing matches to find distinct schemata, for instance. Most of this is similar to the usage in OPS5, but allowing the restriction on schema variables is a feature only in $PSRL_{OPS8}$.

The following is a list of the differences of the above syntax from OPS5's syntax:

- *form* has been renamed to *template-form*;
- *ce*, *positive-ce* and *negative-ce* have been eliminated, simplifying the grammar and removing the restriction that there be at least one positive *lhs-element*;
- *restriction* is allowed inside the {}s surrounding *lhs-element*; notice that this eliminates the *element-variable* as a distinct entity; also, inside braces, the *template-form* can be omitted;
- no numeric tabs (the constants following the '↑' character) are allowed, or at least, if they are used, they must refer to explicit slots, rather than behaving as in OPS5;
- only one *lhs-value* per tab is allowed - no vector attributes are matchable at this level;
- each *lhs-term* must have an explicit tab;
- *lhs-value* is explicitly allowed (due to preceding) at the head of *template-form*;

Some examples are given below, after the remainder of the syntax of lhs-elements is presented.

Within a template, whose $PSRL_{OPS8}$ syntax is indicated by parentheses, there is an optional class name (a value without a slot name preceding it) followed by a sequence of terms. The class name, if present, becomes the value of a PSRL-INSTANCE[10] relation in the template. Each term has a constant name, delimited by a prefixed '↑' (up-arrow or carat), and a value expression. The name becomes a slot in the template schema, and the value expression becomes the slot's value along with any range restrictions on that slot value. The usual slot value is a variable, a constant atom or schema name, all of which are simple forms of *restriction* (defined fully below). In the variable case, the template will match another schema whose corresponding slot has an arbitrary value, when the variable hasn't been bound already by a previous value match, or with a value that is the same as the variable's binding. In the cases of constant atoms or schema names, the template matches only schemata with precisely those values.

3.4.2. Details on restriction forms

When it is desirable to place a further restriction on a value in a matched schema, it is done by one of several notations. These are defined first formally, as follows.

```
restriction          ::= predicate atomic-value
                     ::= << atomic-value* >>
                     ::= atomic-value
                     ::= schema-name

predicate            ::= =
                     ::= <>
                     ::= <
                     ::= <=
                     ::= >
                     ::= >=
                     ::= <=>
                     ::= FUNCTION
                     ::= !
                     ::= TYPE schema-name

atomic-value         ::= constant-symbolic-atom
                     ::= variable
                     ::= number

schema-name          ::= SRL/1.5-double-quoted-schema-name

constant-symbolic-atom  ::= SRL/1.5-schema-name-without-double-quotes
```

These terms are not defined formally: *number*, *variable*, *SRL/1.5-double-quoted-schema-name*, and *SRL/1.5-schema-name-without-double-quotes*. A *number* is a Franz Lisp or Common Lisp number. A *variable* is an atom beginning with '<' and ending with '>', as in OPS5.

The simplest notation covers the case where there is no main slot value (i.e., no match-variable), but only a single-predicate restriction, in which case it suffices to have the predicate, as described below. When the restriction is more complex, including a main value (usually a variable to be bound to the value matched) and one or more predicates, then the restriction expression is

[10] This relation name can be changed by the user, as described in Section 4.5.

enclosed in braces '{ ... }' - this represents an 'AND' conjunction of the predicates. Another case is when one of a set of values is to be matched, in which case the '<< set-of-values >>' notation is used, converting internally into an 'OR' expression, with each value quoted so that variable bindings are not substituted. Note that these options do not allow a complete expression of all the possibilities available for PSRL restrictions as described in Section 2.4 (e.g., NOT and evaluated OR are not available).

Each predicate in a restriction has a predicate operator followed by a value to which the value in the matched schema is compared using the predicate operator. The current set of operators is: = (the equality predicate), <> (not-equal), < (strictly less than, numerically), <= (less than or equal to), > (strictly greater than), >= (greater than or equal to), <=> (of the same type, e.g., both numeric), FUNCTION or its synonym ! (which is followed by the name of a one-argument Lisp function to be evaluated - the argument passed to it is the value in the schema being matched to the template), and TYPE (which is followed by two schema names, one a relation and the other the desired value of the relation). Currently, the implementation of PSRL places a constraint on the use of variables within predicate expressions: they must have been previously bound to values in the ordinary left-to-right matching of the lhs-elements, or an error will occur and the match will fail. This constraint can be easily met by judicious ordering of lhs-elements.

One feature that may cause surprising results in matching is that a slot without a value will appear to have the value nil (which is also true, and surprising, in OPS5). The way to prevent this nil from being bound to a variable as if it were a real value is to use a restriction such as '<> nil' or '{<x> <> nil}'.

The following lists the changes in the above syntax from the corresponding OPS5 syntax.

- FUNCTION, !, and TYPE have been added to the list of possible predicates;
- *constant-symbolic-atom* is further defined;
- There is no \\ quoting here;
- *var-or-constant* has been eliminated (cf. *atomic-value*);
- Schema names can be matched here;
- The distinction between *element-variable* and *variable* has been dropped.

Here are some fragments of rules with unusual syntactic constructions. Elements from the same rule are grouped together, with different rules separated by blank lines. The entire rule system (constructed as a syntax test) is shown in Appendix 7.5, including a test run.

```
{ (class1 ↑tab1 {<a> > 0} ↑tab2 val1 ↑tab3 nil) <g>}
{ (class1 ↑tab1 > 0 ↑tab2 TYPE "psrl-instance" "class2" ↑tab3 {})
      <g> ! schema-p}

{<s> <> <g> (class2 ↑tab1 val1 ↑tab2 ! schema-p)}
{ (class2) <> <s>}

  ({} ↑is-a relation ↑transitivity <> nil)[11]
{ (class2 ↑tab1 <> nil) <> "myname" }

{ (class2 ↑tab1 <a>) = "myname" }
{ (class1 ↑tab2 <s>) }
{ <s> }
```

The FUNCTION feature of $PSRL_{OPS8}$ allows much more predicate-testing power than is available in OPS5. For instance, functions of values of match variables can be computed and compared with a value being matched, as in the following example:

```
(class1 ↑slot1 <a> ↑slot2 <b>)
(class2 ↑slot3 ! exfn1)
    . . .
where
(def exfn1 (lambda (v) (lessp (interp-compute '("<a>" + "<b>")) v)))
```

In this example, the value being matched in the 'class2' schema is compared to the sum of the values of the variables <a> and <b> matched previously by the 'class1' schema. The match would succeed if the sum were less than the value being matched. Interp-compute is an internal PSRL function that takes one (evaluated) argument, an expression to be computed, as in the compute rhs function (whose full grammar is described in Section 3.5.2). An alternative to using interp-compute is for the user to write the needed Lisp code; within such a user function, the form

```
(valueg1 "<var>" "value" nil nil)
```

can be used to get a match variable's value.

3.4.3. Summary of OPS5-OPS8 differences

In order to summarize some of the most powerful features of $PSRL_{OPS8}$, it is useful to bring out the most important ways in which the left-hand-sides of OPS5 differ from those of $PSRL_{OPS8}$:

- direct relations among schemata (including those matching to templates) are expressible;
- restrictions can be placed on variables bound to schemata matched by templates, including '= "name"';
- the template may be omitted, leaving just a restricion on a schema-variable (called element-variable in OPS5) - if only a variable appears in braces, a schema-p test is done on it;
- inheritance is available, along with SRL contexts;
- the restriction operators FUNCTION and TYPE are available;

[11] {} first in a template means that "psrl-instance" must be defined as a slot; it is legal generally to not have any "psrl-instance" entity, by starting the template with '↑'.

- it is legal for a template to start with '↑', as long as some SRL relation is specified within the template;
- {} means that a slot or relation is defined (nil requires that nil be the value);
- an element (schema) can have more than one vector attribute slot.

3.4.4. Vector attributes

A template or schema can contain more than one vector attribute. This is implemented simply by having a list value of a slot. But in matching, only the first value will be matched. This reflects a philosophy that SRL slots should be used for representation of complex entities, rather than lists; lists are seen as simply a convenience for processing sets or lists of objects, one at a time. Action functions to manipulate such lists are psrl-cdr and psrl-copy (see below); no substr function is provided.

Relations in SRL can be a type of vector attribute, since matching a relation can get a list of possibilities bound to a variable. When a relation matches to a bound variable that is a list of schemata, a match occurs (by definition) if some relation in the list succeeds an r-test.

3.4.5. Relations in templates

There must be at least one relation in a template, to allow matching, since r-find is used to access possible candidate schemata for matching. One exception is a schema-variable already bound in the normal left-to-right matching process. If user-defined relations are used, their definitions must be completed before they are used in rules; the user should be sure that r-find and r-test work on such relations.

3.4.6. Efficiency hints for Left-hand-sides

It is best to put the most specific schema templates first in the lhs, since this will narrow down the possible matches for the rest of the lhs, and reduce searching. One helpful construct is to use a relation in a template to bind a variable that is used later in the lhs as a schema-variable.

3.5. Right-hand-side elements in OPS8

A right-hand-side (rhs) is a sequence of rhs-elements, each of which translates to a schema that is an instance of **rhs-element**. In the process of executing an rhs, a Lisp function is evaluated to perform the action specified by each rhs-element. These actions are done in a left-to-right order. An rhs-element is represented notationally by a parenthesized list containing an action name followed by some arguments, and in some cases by slot-value pairs, where slots are preceded by the '↑' character. A grammar describing this precisely is given below, as an overview; each type of rhs-element is considered in more detail later in this section.

```
rhs-element     ::= ( make atomic-value rhs-term* )
                ::= ( make schema-name atomic-value rhs-term* )
                ::= ( make nil rhs-term* )
                ::= ( make schema-name nil rhs-term* )
                ::= ( make rhs-term* )
                ::= ( make schema-name rhs-term* )
                ::= ( Remove rhs-value* )
                ::= ( modify atomic-value rhs-term* )
                ::= ( halt )
                ::= ( bind variable )
                ::= ( bind variable rhs-value )
                ::= ( cbind variable )
                ::= ( write rhs-value* )

rhs-term        ::= ↑ slot-name rhs-value

slot-name       ::= constant-symbolic-atom
                ::= number

rhs-value       ::= atomic-value
                ::= function

function        ::= ( genatom )
                ::= ( accept )
                ::= ( acceptline atomic-value* )
                ::= ( compute expression )
                ::= ( eval Lisp-S-expression )
                ::= ( psrl-cdr atomic-value slot-name )
                ::= ( psrl-copy atomic-value slot-name )
                ::= ( crlf )
                ::= ( rjust atomic-value )
                ::= ( tabto atomic-value )

expression      ::= number
                ::= variable
                ::= expression operator expression
                ::= ( expression )

operator        ::= +
                ::= -
                ::= *
                ::= //
                ::= \\
```

Here are the differences between the above grammar and that of OPS5:

- *action* has been renamed to *rhs-element*;

- make has two new formats, one with an explicit class-name first, standing for an optional schema-name for the created schema, the other with nil as the class name, in which case no class name is given to the created schema (it won't be **psrl-instance** of anything);

- Remove has to be capitalized (as in ISLOPS), to distinguish it from the Lisp function of

the same name;

- Remove's arguments are more general, not just element designator (but numerical element designators cannot be used here as in OPS5);
- modify's first argument are more general (but numerical element designators cannot be used here as in OPS5);
- bind's form is more restricted, but it corresponds with the usual interpretation;[12]
- cbind's argument is more general;
- write's arguments are more restricted, excluding the use of the '↑' notation;
- rhs-term must have an explicit tab, and that tab can't be a variable;
- slot-name is a new category, used in place of atomic-value in some cases;
- some ops5 action (rhs-element) and function forms are not allowed: call, openfile, closefile, default, build, litval, substr, user-defined-function (use eval instead);
- accept and acceptline don't take file symbol arguments;
- eval, psrl-cdr, and psrl-copy are new.

3.5.1. Details on actions

$PSRL_{OPS8}$ contains many actions and functions corresponding to those of OPS5, but PSRL's actions do not have quite all of the features available in OPS5, as follows. It is not meaningful to refer to positions within SRL schemata numerically, but only by slot name.[13] (The user who desires this capability for lists can do the needed operations in Lisp.) Also, $PSRL_{OPS8}$ will not allow schemata matched by lhs-elements to be referred to in actions by number (as with *element-designator* in the OPS5 grammar), since the SCHEMA-VARIABLE slot exists for this purpose. The actions are described individually in the remainder of this subsection.

---

[12] OPS5 allowed further arguments, but these would only be meaningful in case of the use of the call mechanism, which is not implemented in PSRL.

[13] Slot names can be numbers, but this is not the same meaning as it would be in OPS5.

```
( make atomic-value rhs-term* )
( make schema-name atomic-value rhs-term* )
( make nil rhs-term* )
( make schema-name nil rhs-term* )
( make rhs-term* )
( make schema-name rhs-term* )
e.g.,
(make cat ↑type Siamese ↑age 3)
(make "my-favorite" cat ↑color grey ↑fur long ↑type domestic)
(make nil ↑instance my-favorite ↑patches white)
(make "GrowlTiger" nil ↑instance my-favorite ↑stripes gold)
(make ↑instance my-favorite ↑patches white)
(make "Smokey" ↑instance my-favorite ↑eyes blue)
```

Make creates and associates with Working Memory a new schema, given a template in one of six forms. In all forms, match variables' bindings are substituted and functions are evaluated. The first form is the usual one (corresponding to a make form in OPS5), where an initial atomic value is interpreted as a class name, and is placed in a PSRL-INSTANCE[14] slot in the schema. The rest of the arguments are simply slot-value pairs. The Lisp function gensym is used to create a new schema name. The second form has an additional argument, the name of the schema to be created. The last four forms allow the user to create a schema without the PSRL-INSTANCE slot in it - only the slots and values are put into it. The fourth and sixth forms correspond in action to the SRL mk-schema function. When a schema name is given, it is taken to be the name of the schema to be created, unless the schema already exists and is the name of a literalized class; that is, the semantics can be used to resolve ambiguities in arguments. Note that schema names must have the double-quotes, while class names can omit them.

The schema form that the make element instantiates upon translation is shown in Figure 3-13. The class name goes into the PSRL-INSTANCE slot, and the name, if any is given, goes in the PSRL-MAKE-NAME slot.

{{ **make**
PSRL-INSTANCE:
PSRL-MAKE-NAME:
PSRL-INTERP-FN: interp-make}}

Figure 3-13: make Schema

```
( Remove rhs-value* )
e.g.,
(Remove <g> "other-random" (eval (some-fn some-arg)))
```

Remove destroys the schemata designated by its arguments. (Note that upper case is required for the first letter - cf. ISLOps, ops5 running in ISLisp.) The example above destroys three schemata, each specified in a different way. Schemata are destroyed by using the SRL function schemad. A

[14] The user can change this, as described later.

restriction is applied, though:[15] the schema must have been placed in PSRL's working memory (e.g., must have been created with a make). Otherwise, Remove is a no-op. Figure 3-14 displays the form of which Remove actions are instantiations. Arguments to Remove are placed as a list in the RHS-TERMS slot.

```
{{ Remove
    RHS-TERMS:
    PSRL-INTERP-FN: interp-remove}}
```

Figure 3-14: Remove Schema

(**modify** *atomic-value rhs-term**)
e.g.,
(modify <x> ↑fur short ↑color black&white)

Modify makes changes to the schema designated by its first argument, according to the slot-value pairs that form the rest of its argument list. For each slot-value pair, the value of the named slot in the schema to be modified is replaced by the given value. (This is not like the OPS5 modify operator, which destroys the entire element and makes a new copy with the requested changes.) Figure 3-15 shows the general schema of which each modify action is an instance.

```
{{ modify
    PSRL-INSTANCE:
    PSRL-INTERP-FN: interp-modify}}
```

Figure 3-15: modify Schema

(**halt**)

The halt action, as in OPS5, causes the current rule set to halt execution. It does so with a Lisp throw to the containing run function (described in Section 4.5). Thus actions within the rhs occurring after the halt will not be performed. If several runs are nested, only the immediate one is affected. Figure 3-16 has the schema form.

```
{{ halt
    PSRL-INTERP-FN: interp-halt}}
```

Figure 3-16: halt Schema

[15] The user can easily create an action without this restriction, as described later.

```
( bind variable )
( bind variable rhs-value )
e.g.,
(bind <x>)
(bind <z> (compute <w1> + <w2>))
```

The bind action assigns a value to a match variable. The value is the Lisp value returned by evaluating its second argument. When no second argument is given, the value assigned is a schema made from a gensym atom, i.e., from a new, unique name. Later uses of the same variable in expressions in the rhs will result in using the computed value assigned to the variable. The variable value is reset to have no binding before the next rule match is done. Figure 3-17 shows the schema form for bind.

```
{{ bind
    VARIABLE:
    RHS-TERMS:
    PSRL-INTERP-FN: interp-bind}}
```

Figure 3-17: bind Schema

```
( cbind variable )
e.g.,
(cbind <s>)
```

Cbind assigns the name of the most recently created or modified schema to the given match variable. Figure 3-18 shows its form.

```
{{ cbind
    VARIABLE:
    PSRL-INTERP-FN: interp-cbind}}
```

Figure 3-18: cbind Schema

```
( write rhs-value* )
e.g.,
(write (crlf) Be advised that (compute <a> + <b>) is the sum of
    <a> and <b> \. (crlf))
```

The write action causes a message to be printed on the standard output. The crlf, tabto, and rjust functions, described below, can be used within a write action to achieve formatting of the output. Figure 3-19 gives the schema form of the action.

```
{{ write
    RHS-TERMS:
    PSRL-INTERP-FN: interp-write}}
```

Figure 3-19: write Schema

3.5.2. Functions

$PSRL_{OPS8}$ has a number of functions that are used to compute values that are used to fill slots or as bindings of variables within an rhs. In the case of several functions for use within write, there is only a side-effect, and the value returned is not important. Examples of uses of these functions within rules are given in the appendices of this manual.

(genatom)

The genatom function returns a new schema, generating its name from the Lisp function gensym.

(accept)

The accept function reads a Lisp expression (either a single atom or a list structure in parentheses) from the standard input (e.g., typed by the user on the terminal), and returns what was read. It uses the Lisp function read.

(acceptline *atomic-value** **)**
e.g.,
(acceptline nothing input)

The acceptline function reads a line of text from the standard input (e.g., typed by the user on the terminal) and puts it into a list, returning what was read. It uses the Lisp function lineread. If an empty line is entered, the list of atomic values that are the arguments to acceptline are returned, instead of nil.

(compute *expression* **)**
e.g.,
(compute <a> + <b> / (<c> - <d> * <e>))

The compute function evaluates an arithmetic expression (including substituting values for match variables), returning the final value. Parenthesized subexpressions are allowed. As in OPS5, evaluation of the expression goes from right to left, through the use of recursion (except that subexpressions may be computed before others). The operators, +, -, *, //, and \\, are plus, minus, times, quotient, and remainder (numbers are subjected to Lisp fix, making them integers, before dividing). The computation is done by the Lisp function interp-compute, which has been described above in connection with evaluating expressions in lhs predicates.

(eval *Lisp-S-expression* **)**
e.g.,
(eval (random-in-range <n1> <n2>))

The eval function evaluates a Lisp expression, and puts the returned value into the corresponding schema slot. Values of bound match variables are substituted for the variables, except when they are quoted (i.e., inside (quote ...)). (To get the value of the variable quoted, as when passing it to an expr, use (list 'quote <var>).)

(psrl-cdr *atomic-value slot-name* **)**

The psrl-cdr function takes the Lisp cdr of a slot value (which makes sense only for those with list values, i.e., those declared to be vector attributes). For instance, to replace the value of the items slot of a schema (assumed to be bound to the variable "<s>" previously in the containing rule), the following would work,

```
(modify <s> ↑items (psrl-cdr <s> items))
```

(**psrl-copy** *atomic-value slot-name*)

The psrl-copy function makes a copy of a list value in a slot, allowing it to be placed in another slot. For instance,

```
(make backup-class ↑backup-items (psrl-copy <s> items))
```

(**crlf**)

The crlf function, used inside the write action, causes a newline character (or character sequence) to be printed before the elements that follow it are printed.

(**rjust** *atomic-value*)[16]

```
e.g.,
(rjust 10)
```

The rjust function, used inside the write action, causes the element that follows it to be printed right-justified in a field whose width is the value of the argument to rjust.

(**tabto** *atomic-value*)[17]

```
e.g.,
(tabto 5)
```

The tabto function causes spaces and newlines to be printed such that the next element in a write action will start in the column specified by the value of its argument. It uses the Lisp tab function.

In summary, the following major differences exist between OPS5 and the right-hand-sides of $PSRL_{OPS8}$:

- it is not possible here to refer to matched elements (schemata) by their position in the lhs, but only by their being bound to a schema-variable;
- the eval action function makes it more direct for actions to incorporate function values into schemata;[18] the OPS5 user-defined function primitives are not implemented here, given that eval, SRL primitives and PSRL user-defined actions can be used to accomplish the same thing;
- the functions psrl-cdr and psrl-copy are provided for vector-attribute value manipulations,

---

[16] Not fully implemented at press time.

[17] Not fully implemented at press time.

[18] This is done in OPS5 only indirectly, via the $value primitive, and not by using the value returned from a function call.

replacing substr (which has no meaning in a schema environment);

- the user can define new rhs-element forms, as described in Section 4.10.

4. User Commands for PSRLOPS8

The previous chapters have defined the elements of the PSRL language. What remains is to describe the various facilities for organizing and controlling systems written in PSRL. This chapter gives detailed instructions for starting to run the PSRL system, for making use of control functions for rule sets, and for interactively running, debugging and expanding PSRL-based systems. Many of the topics here are illustrated with the extended examples in the appendices of this manual (Chapter 7). The appendices are intended to provide motivation for why the various features are considered useful, and they give complete debugging runs, indicating the order in which the various commands are typically given. Also, the appendices show examples of command outputs, where only the input form of the command is given here.

4.1. Loading the PSRL system

The PSRL system runs within ISLisp5 and ISLisp6,[19] the current versions of Intelligent Systems Laboratory Lisp, which is based on Franz Lisp as distributed by Berkeley, for their Unix™ operating system. It is available as a collection of files that are read into ISLisp using a single load-command file, either

/usr/mdr/psrl/bin/psrl.init

for ISLisp5, or,

/usr/mdr/psrl/bin6/psrl.init

for ISLisp6.[20] The init file also connects the user to a modification of the standard SRL basic database, and performs an initialization function. See Appendix 7.1 for a full listing of /usr/mdr/psrl/bin/psrl.init.

There are a number of subdirectories of /usr/mdr/psrl that the user may want to access for on-line examples (including rule sets and transcripts of runs), information, internal documentation and code:

- bin = the object files that are loaded to obtain the PSRL system for ISLisp5, as described above, along with a PSRL database creation file (psrldb.upd) and a database file (basicps.db);

- doc = various help files, including a history of the changes that PSRL has gone through; there are also .schout files, showing various schemata in the system, and mss files for generating the manual;

- test = test programs used for debugging and verification, along with scripts of the most recent test runs (extension .scr*); subdirectories contain various bug-repair tests;

- binsrc = source files corresponding to the .o files in bin (i.e., backups);

- binold = files from a past version of PSRL, combined from a former bin & binsrc;

---

[19] Some parts of the ISLisp6 version of PSRL are not fully debugged, and the ISLisp6 version will be referred to as version 1.3, when completed.

[20] These pointers apply to the ISL1 and ISL2 Vaxes at CMU; on most systems, the /usr/islisp/... (?) directory will be more appropriate; contact the author for further information regarding other computers.

- src = Lisp source files (extension .l), some possibly in experimental states, along with test programs, scripts, and compiler output listings (.liszt); the source files contain a detailed history and comments for each function;

- bin6 = files superseding those in bin, for the ISLisp6 version of PSRL;

- src6 = files superseding those in src, for ISLisp6.

4.2. Style of Use

After the user has loaded the PSRL system according to the above procedure, various top-level commands are available to load and translate rules, to assert and manipulate elements in Working Memory (the current database context), and to run production systems. All such operations allow the user to express schemata in the $PSRL_{OPS8}$ notation (the basic SRL notation is also available, but it tends to be slightly more cumbersome to use). There are several debugging modes that a user might adopt:

- load and translate rules, then edit them using the Lisp editor, saving changed rules with the chkpt function; chkpt files can later be merged with the original rules using a text editor;

- load and translate rules, then save them on a database; edit rules using a text editor (after suspending the Lisp job), then go back to the job, and load and translate only the changed rules (which are kept in a separate file); for later runs, connect to the most recent version of the database and load in only the rules that have changed since that database was updated; (periodically, the changed rules might be merged with the original text file;)

- load and translate rules once, then edit only using sedit, saving the rules in their database form, and not maintaining the OPS8 text form; (SRL databases may not be very reliable, but there are ways of saving them in a text form so that a fresh start can be made;)

- create rules interactively as schemata, without using the OPS8 notation; this tends to be cumbersome and slow; retaining the database would be done as above.

As modes of use evolve, the PSRL system will have more facilities added to support the most common ones.

PSRL is meant to be run within a modified Lisp top-level, called UIN8, that is obtained by typing '↑' (carat or uparrow) to ISLisp. That top-level is also exited with '↑'. The following command examples adopt the UIN8 conventions; UIN8 itself is described in Section 4.9. It is very similar to the standard CMU-Lisp top level, with added features for use with OPS5 and $PSRL_{OPS8}$.

Notation. In the following function descriptions, arguments are assumed unevaluated unless they are preceded by a single-quote ('). That is, most functions are Franz nlambdas. If schema names are involved, for the unevaluated case, it is usually legal to drop the double-quotes around the names, but not for the other case - in either case, double-quotes are acceptable and their use removes the need for the user to know whether a function is an nlambda or not. Square brackets surround optional arguments.

4.3. Complete grammar for commands

This section summarizes the available commands. Details are given in the sections that follow, each of which repeats and discusses a small portion of this grammar.

debugging-command ::= *help-command*
::= **init-psrl**
::= **init-ops8**
::= **tlps** *ps-name [rule-file-extension] [literal-file-extension]*
::= **lps** *ps-db-name*$^{+}$
::= **run** *['ps-name] ['control-fn] [number]*
::= **production-system** *ps-name*
::= **control** *fn*
::= **wm** *class-name**
::= **matches** *rule-name*
::= **matches** *lhs-element*$^{+}$
::= **pm** *rule-name*$^{+}$
::= **fire** '*rule-name*
::= **backtrace**
::= **watch** *[number]*
::= **psrl-version**
::= **↑**
::= **ops8-read**
::= **ed** *atomic-value*
::= **newp** *rule-name lhs* **-->** *rhs*
::= **newp** *rule-name* **like** *rule-name*
::= **incstats**
::= **pp** *atom*
::= **changes**
::= **chkpt**
::= **prop** *atom*$^{+}$

help-command ::= **?**
::= **?b**
::= **?e**
::= **?i**
::= **?p**
::= **?u**
::= **?5**
::= **?8**
::= **?8p**
::= **?8v**

```
parameter-setq      ::= parameter Lisp-expression
                    ::= setq parameter Lisp-expression
                    ::= ( setq parameter Lisp-expression )

parameter           ::= $ops8-ps
                    ::= $psrl-instance
                    ::= $psrl-inherit-slots
                    ::= $match-inherit
                    ::= $psrl-context
                    ::= $psrl-control-step
                    ::= $demon-run
                    ::= $trace-match
                    ::= $debug-fire
                    ::= $psrl-silence
                    ::= $debug-match

action-command      ::= make atomic-value rhs-term*
                    ::= make schema-name atomic-value rhs-term*
                    ::= make nil rhs-term*
                    ::= make rhs-term*
                    ::= modify atomic-value rhs-term*
                    ::= Remove rhs-value*
                    ::= bind variable
                    ::= bind variable rhs-value
                    ::= cbind variable
```

Lisp-expression and *atom* are not defined formally here.

4.4. On-Line help

The function named '?' prints a message describing the collection of on-line help functions for PSRL and its user interface. Some on-line help is also included for OPS5, since the $PSRL_{OPS8}$ interface is a superset of the UIN8 interface for OPS5. The following does the same, listing the functions alphabetically:

- ?b - hints on reporting PSRL bugs, i.e., what things to do in order to give the maintainer enough information to solve the problem;

- ?e - a summary of an easy subset of commands available in the Lisp editor, which is used in PSRL to edit the OPS8 notation of rules (full documentation is in the Franz Lisp manual);

- ?i - the INITF utilities (described with the user interface, below);

- ?p - UIN8 Parameters (Lisp global variables that can be set by the user);

- ?u - the UIN8 User Interface;

- ?5 - OPS5 (its actions, declarations, functions and commands); ?8 - $PSRL_{OPS8}$ functions;

- ?8p - $PSRL_{OPS8}$ parameters (globals);
- ?8v - a display of current values of SRL and PSRL global parameters.

4.5. Loading, translating and running rules

Executing rules in PSRL requires a preliminary step of defining the rules themselves along with creating a number of other auxiliary schemata and slot values that are used by the interpreter. This section is concerned with that step, while later sections will give details of controlling execution and other operations that users perform in applying PSRL.

A user can construct .init files for loading and initializing production systems (using the standard Lisp function, load - see also Appendix 7.1), or the following commands can be used, for most systems. When a file with rules is loaded, the declarations are executed as functions. The p function has some built-in error recovery features that are described below in connection with the newp function.

```
init-psrl
init-ops8
```

These two equivalent functions perform the PSRL system initialization. This is done automatically in the psrl.init file mentioned above for loading the system, and also by the following.

tlps *ps-name [rule-file-extension] [literal-file-extension]*
e.g.,

```
tlps autofac
tlps autofac op5 lit
tlps test3 nil nil
```

Tlps translates $PSRL_{OPS8}$ rules into schemata, after doing initialization and database (basicps) connection, if it hasn't been done already. It should be used only interactively (typed by the user) and not within files, since it prompts the user for information. A file with literalize declarations is loaded first, if either no third argument is given and a default value exists (as described presently), or if something non-nil is given; the third argument is taken to be an extension of the *ps-name*, e.g., autofac.lit in the case of the second example above. Then the rule file is loaded, named by extending the *ps-name* with the second argument. Default file extensions, used in case nothing is given after the *ps-name* (first example above), are determined by the setting of the global variable $ops8-exts. The value of $ops8-exts is a list of two elements,

```
(rule-file-ext literalize-file-ext-or-nil)
```

whose initial value is '(op5 lit). If the second list element is nil, no literalize file is read; a rule file is always read - if the user gives nil for the rule file extension argument to the command, the first element of $ops8-exts is used. Tlps allows the user to place schemata for translated rules in a new database, according to interactive prompts. It automatically does a production-system declaration, using *ps-name*, whose effects are described below.

lps *ps-db-name*$^{+}$
e.g.,

```
lps autofac
```

The lps function connects to database(s) for production systems that have either been set up by

SRL schema-building commands or by previous tlps's. Like tlps, it will do initialization and basicps database connection, if that hasn't been done yet. It also does a production-system declaration, implicitly, since the production-system schema is usually in the database. (This function is rarely used, given tlps.)

$ops8-ps

The global Lisp variable $ops8-ps is bound to the name of the current production-system schema. The user should not set this variable directly - it is maintained by the production-system function (a declaration) and by run (which binds it and saves the old value). Its stack property has old values, stored first-in-first-out, so that the user can examine old values by using the prop function (part of the OPS8 user interface, described below).

run *['ps-name] ['control-fn] [number]*
e.g.,
run
run "autofac"
run "autofac" 'fire-cycle 13
run 'autofac 'fire-cycle 13
run 13
run 13 "autofac"

Run executes the production system *ps-name*, if it is given, or uses the value of the global $ops8-ps. If a second non-numeric argument appears, it is used as a control function (see control declaration, below); otherwise, it uses the control slot of the chosen production-system, or it goes into an interactive mode, where it prompts the user for the next rule to attempt to fire (this mode is also entered if $ops8-ps is unbound and no *ps-name* argument is given). The numeric argument, if given, determines the maximum number of rule firings to be done; if not given, -1 is used (which means no maximum). The Lisp special variable $fire-cycles is set by run, to convey a numeric argument to the control function. When run is called automatically by UIN8 (detailed below), the numeric argument is the value of $ops8-run-n, which starts out at 20.

production-system *ps-name*
e.g.,
production-system autofac

The production-system function declares *ps-name* to be the current one. This is an interactive way to do the declaration, and would override any declarations appearing in files (as described in Section 3.3). Rules entered interactively (see newp, below) would be placed into the named production system. This could also be used to set $ops8-ps to a previously-defined system.

control *fn*
e.g.,
control fire-sequence

The control function changes the control slot of the current value of $ops8-ps to *fn*, which should be a user-defined control function or one of the standard ones: fire-cycle[-reverse], fire-cycle-one, fire-one[-reverse], fire-sequence. These are described in more detail below, in a separate section on the recognize-act cycle (4.6).

Other declarations

Note: It is also legal to do p, literalize and vector-attribute declarations as commands, but these are usually done in files; see also **newp** below.

```
$psrl-instance
e.g.,
$psrl-instance "instance"
(setq $psrl-instance "instance")
```

The Lisp global $psrl-instance names the relation between a WM element and its class (as the term is used in OPS5); the default value is "psrl-instance", which is defined in the PSRL database to be similar to "instance" (its definition is in Section 2.4).[21] It is a separate relation to allow it to have different inheritance, if a need for that were to arise. Recall that in PSRL, each schema, in order to be matched by lhs-elements, must have a relation. The $psrl-instance relation is automatically defined for schemata as they are translated from $PSRL_{OPS8}$ notation. (This can be over-ridden by putting a '↑' as the first element of an element, but the user is then responsible for including some other SRL relation.) The inverse relation of $psrl-instance is the value of the global $psrl-instance + inv, which should be set at the same time as $psrl-instance is; it defaults to "psrl-instance + inv". This global should be set before any rules are loaded or translated.

```
$psrl-inherit-slots
e.g.,
$psrl-inherit-slots t
$psrl-inherit-slots '(step "is-a" all t)
```

The value of the Lisp global $psrl-inherit-slots is used as the *path* argument to various SRL functions that create values in schemata, e.g., valuec1. When non-nil, this forces slots to be inherited, rather than being created with slotc.[22] When it is nil, the PSRL code does the slotc explicitly as appropriate. Ordinarily, slots would be inherited from the schema that results from a literalize declaration (described in Section 3.3), or from a similar schema previously defined by the user. Thus a non-nil setting provides a check of rule syntax, since an error would occur for slots not properly declared. The default value is nil. See also $match-inherit.

Nested rule sets

PSRL does not provide any functions to support the feature described in Section 2.1, namely that a "rule-set" slot can contain names of other production systems, which are executed in turn when a rule in the same location would have been matched. I.e., the user must arrange any hierarchies of rule sets manually.[23]

---

[21] The second example form indicates the right form when including this parameter in a file; the first is useable only inside UIN8.

[22] Due to its nature, slots can not be inherited within the modify action.

[23] It would be easy to construct a function that could be placed in a file among rules to put a ps-name into the rule-set slot in the corresponding place.

4.6. Controlling execution: the recognize-act cycle

The basic cycle of rule execution consists of four steps in PSRL:

1. selection of rules for matching,

2. matching,

3. conflict resolution, and

4. action execution.

Unlike many other PSs (e.g., OPS5), PSRL does not find all possible rule matches before doing conflict resolution, but rather uses the user-specified control strategy (see the control declaration, described above), within the context of a particular rule set, to select rules to be matched. Usually conflict resolution and action execution are done as soon as a rule is found that matches successfully - conflict resolution is done to restrict execution to a subset of the instantiations of the successful rules that are executed.

Selection of rules for matching

There are a number of options for control, as established by the control declaration described above. These are atomic names of Lisp functions, and the user can imitate the code for these to construct other variants - contact the author for details and pointers.

- fire-cycle goes through a rule-set in order, executing the first rule that matches (it may match in several different ways, but only one is executed, as described in conflict resolution below): when a successful execution is finished, control returns to the beginning of the list, and repeats from there; when no successful match is found, or when the number of cycles specified in the run command have been done (see $fire-cycles, Section 4.5), or when a halt action is done, running of the rule-set is stopped; a number of matches to a single rule may be found, with the choice of which one to execute left up to the conflict resolution step;

- fire-cycle-reverse performs like fire-cycle, except that the rule-set is processed from last element to first, rather than starting from the first;

- fire-cycle-one performs like fire-cycle, except that the matching stops as soon as one match is found, making the match step faster (though providing less information) and conflict resolution trivial;

- fire-one goes through a rule-set in order, executing the first rule that matches, then stopping; like fire-cycle, it may find more than one match, with conflict resolution making the choice of which one to execute;

- fire-one-reverse is like fire-one, except that the rule-set is examined from last element to first;

- fire-sequence processes a rule-set in order, executing each rule that matches in turn, without returning to the beginning of the list; all possible matches of a rule are executed before going on to the next rule to match; when all rules have been matched once, or when a halt is executed, the run stops.

Matching

Matching is controlled by the following Lisp global variables.

```
$match-inherit
e.g.,
$match-inherit t
```

The Lisp variable $match-inherit is used as the <path> argument to valueg1 and other value-accessing functions, within the match procedure of PSRL. Setting it to t, for instance, specifies that inheritance along any path will be allowed. The initial value of $match-inherit is nil, which is the opposite convention from most SRL functions, for purposes of efficiency.

```
$psrl-context
e.g.,
$psrl-context '$branch-context-3
```

The value of the Lisp variable $psrl-context is the context relative to which all matching of lhs-elements is done. It applies to 'working memory', not to rule templates[24] It is initially set to nil, signifying the currently asserted context.

Conflict resolution

Most control functions (described above) have the same conflict resolution: to choose the first element of the conflict set. The order of the conflict set depends on the order that the corresponding matches are found by the matcher, and is thus somewhat arbitrary (though consistent) - currently each new match found is placed at the beginning of the set. An exception is fire-sequence, under which all instantiations of a rule are executed (without matching in between the executions). The conflict set is emptied before each cycle.

An OPS5-like conflict resolution procedure cannot be defined because SRL schemata don't ordinarily have anything corresponding to OPS5's time tags.[25] These are central to the main conflict resolution principle, which uses memory recency, allowing the processing to focus on the most recently-modified working memory elements. There are also efficiency considerations in avoiding an OPS5-like approach. The current approach also has the advantage of giving the user much closer control over processing. It has been considered undesirable in other systems to follow this because it makes some aspects of control implicit in the rule-set organization, rather than having each rule state its control assumptions and control elements explicitly as part of working memory.

---

[24] The plan is to have in Version 1.3 of PSRL a separate context variable for rules, named $psrl-rule-context; if nil, the default context would be used; if t, a unique context would be set up for each rule-set; otherwise, the value would be a context into which all rules would be placed.

[25] It's actually not impossible but difficult; there is a cost associated with maintaining the SRL meta-information that would be necessary here, and most users would not want to pay the extra cost, unless PSRL itself were to be sped up considerably. Another issue for large existing databases is whether they have maintained that meta-information properly.

Action execution

Action execution is controlled by the following Lisp global variables.

```
$psrl-control-step
e.g.,
$psrl-control-step t
```

Setting $psrl-control-step to non-nil causes a pause to occur after each rule firing. The pause uses ops8-read (part of UIN8, described below), which allows arbitrary user interaction to take place before proceeding; the run continues when the user enters 'ok'. The pause takes place after the rule's actions are completed, and match variables retain their bindings from the rule. It is not a good idea to start a debugging run using $psrl-control-step from within a file that is loaded at the beginning, since when the end of the file is read by ops8-read, the run will terminate, leaving the user at the Lisp top level.

```
$demon-run
e.g.,
$demon-run t
```

Setting $demon-run to a non-nil value enables having active demons monitoring schema changes. The user is responsible for including a test on this switch inside all demon functions; PSRL only changes the value internally in places where it would not be appropriate to execute demons, particularly inside the rule translator. Its default value is t, but it is set to nil (as a special prog variable) by the p function.

PSRL does not have any specific constructs to support the use of SRL demons to control rule-set activation, though the user is free to set up demons to invoke rule sets when particular values in particular schemata are established. One can, for instance, set up a demon to execute the run function on a particular production-system schema, when some value is changed, e.g., by a modify action. Due to indeterminacies surrounding just when an action will perform a particular slot setting (which arise from implementation details within SRL's ordering of slots), it is best to place such demon-activating changes last in a rule, after the other actions. Wherever such a demon activation is placed, the execution of that rule is suspended and the demon's rule-set will be processed. This can have side effects, the most important of which is that match variables' bindings may be changed.[26]

4.7. Working Memory and Production Memory Commands

Working Memory in PSRL is used primarily for bookkeeping and display purposes; it is not used to restrict the set of legal matches of templates to the SRL database. (The $psrl-context variable, described above, might be used in that role.)

```
$wm
e.g.,
insert "something" $wm nil nil
```

[26] The knowledgeable user can of course program around this, given that all the variables in the current rule are stored in the "psrl-variables" slot of the rule schema; those variables' "value" slots could be saved before a nested run, and then restored after the run.

The value of the Lisp variable $wm is a sorted list of schemata that have been created by make actions. The user can add other elements as shown. It is used for display (see the following), and for error checking in the Remove action.

```
wm class-name*
e.g.,
wm
wm goal
wm "goal"
wm machine clock
```

The wm function displays all of $wm or selected classes of schemata (where class is determined by the $psrl-instance relation).

```
matches rule-name
matches lhs-element+
e.g.,
matches beg
matches "beg"
matches (clock ↑mod going)
```

The matches function can be used to display selected parts of working memory, by its detailed trace of the matching process. If given a rule name, the lhs of the rule is matched, with full debugging traces turned on; if given lhs-elements, they are translated into templates and then matched as if an lhs. The conflict set is emptied before starting to match.

PSRL has fewer facilities for displaying production-rules than for working memory.[27] The user can find out the current rule-set by displaying the value of $ops8-ps; that variable also records other PSs that may be active (as described above). Individual rules can be displayed with the Lisp pp function (which displays the external notation as read within the p declaration that defined the rule), or with the following.[28]

```
pm rule-name+
e.g.,
pm beg eng clu
pm "beg"
```

The pm function displays an ops5-like format for rules encoded as schemata. It generates the display from the schemata.

4.8. Debugging and tracing

A number of functions from OPS5 can be useful in diagnosing program behavior. One has been described above, as a working memory display: matches, which can be used to help determine why a rule doesn't match. As mentioned above, the appendices provide some illustrations about how

---

[27] Future versions of PSRL may include ways to match to rule contents in the same way as the current version does to working memory.

[28] A future version of pm might do some kind of $ops8-ps display, if given no arguments.

these commands combine in debugging.

```
fire 'rule-name
e.g.,
fire 'beg
fire "beg"
```

The fire function matches the lhs of a rule and executes its rhs if the match succeeds. Its argument is evaluated in order to make it more convenient to use within programs, e.g., within demon functions. The conflict set is emptied before starting to match.

```
backtrace
```

The backtrace function prints out the last few rule firings; it gets them from the Lisp variable $psrl-backtrace, whose length can be adjusted by the user by cons'ing on more elements or by setting it to its cdr. It shows the rule name and the bindings to its variables when it was fired.

```
watch [number]
e.g.,
watch 2
```

The watch function sets a level of tracing, where higher numbers indicate more detail. *Number* ranges from -1 through 4; if no argument is given, the value returned is the current setting. Setting the watch level to 4 automatically sets $trace-match, $debug-match, and $debug-fire to t; setting it to less than 4 changes them to nil.

```
$trace-match
e.g.,
$trace-match t
$trace-match '("beg" "clu")
$trace-match '(dummy)
watch 4
```

The Lisp variable $trace-match determines whether certain trace information is printed out during matching. If set to t (or any non-nil atom), the trace is printed for every match. Setting it to a list restricts the trace to the rules named in the list (names must have double-quotes). If the list contains something that is not a rule, then only conflict resolution is traced (it is already included in the other traces). The watch function automatically turns on 't' tracing at level 4, and turns it off when the level is changed to less than 4.

```
$debug-fire
e.g.,
$debug-fire t
```

The Lisp variable $debug-fire controls tracing of action execution: if set to t, tracing is enabled.

```
$psrl-silence
e.g.,
$psrl-silence t
```

Setting $psrl-silence to t suppresses status-reporting 'politeness' messages, especially in the initialization of PSRL. Equivalent: 'watch -1'.

```
$debug-match
$debug-match t
watch 4
$debug-match nil
watch 2
```

Setting the Lisp variable $debug-match to t turns on a detailed trace of the matching process. It is set as a side-effect of watch 4, and is reset to nil for lower watches. The following will provide yet more trace information about the match, using the Lisp trace facility:

```
trace match-slot match-elem value-in-range-p match-rels-rel ,
    match-rels-elem
```

psrl-version

This function prints the current PSRL-OPS8 version number. It returns nil.

The ?8v function, described in Section 4.4, can be used to display all at once a number of important system parameters. Often strange program behavior can be explained by noting an unexpected parameter binding. The ?b function, also described there, prints out suggestions on what things to look at in reporting apparent PSRL bugs, and these can be used to advantage by the user in looking for bugs in rule sets.

Another useful capability is to be able to interrupt execution to execute debugging commands. One way to do this from inside a user Lisp function is the ops8-read function, described in Section 4.9. A user can call ops8-read by using the form,

```
(catch (ops8-read) ok)
```

Using this convention, a return is done when the user types 'ok'; other actions such as returning to top-level with '↑' are also available (some care is necessary).

An important aspect of debugging is to be able to alter the contents of Working Memory, and in general to perform rule actions interactively. The following commands are oriented towards this capability.

make *atomic-value rhs-term**
make *schema-name atomic-value rhs-term**
make nil *rhs-term**
make *rhs-term**
e.g.,
```
make auto ↑mod order ↑serial xx84
buy ↑dollars 20 & steel ↑what scrap ↑amount 3
make "extra-clock" clock ↑mod going ↑timer 25
make ↑is-a machine ↑capacity 7
```

The make function can be used at the top level to create schemata, in the same way that it is used within rules. The second example above shows the abbreviated form that is possible with the UIN8 top-level; that command line becomes two make commands, and run is automatically called, which makes the top-level appear as if typed commands are communicated directly to the current production-system. The third example creates a schema named "extra-clock".

```
modify atomic-value rhs-term*
e.g.,
modify g00010 ↑mod stopping
```

The modify function can be used to modify existing schemata. Those in working memory (if created by make without an explicit name) have gensym'd names like the one shown.

```
Remove rhs-value*
e.g.,
Remove g00010 extra-clock
```

The Remove function can be used to destroy (with schemad) schemata in working memory.

```
bind variable
bind variable rhs-value
e.g.,
bind <x>
bind <y> (compute <z> + 256)
```

The bind function can be used at the top level to assign a value to a variable. The action in the first example assigns a schema name made from a gensym'd atom.

```
cbind variable
e.g.,
cbind <g>
```

The cbind function can be used at the top level to assign to a variable the name of the schema most recently touched by make or modify.

```
Note: interp-action
```

The user can execute other PSRL actions as commands by creating the corresponding schemata, and then executing interp-action, a function whose single (evaluated) argument is the schema.

4.9. User interface

UIN8 (user interface for OPS5 and OPS8) has several aspects:

- A top level that extends the standard Lisp one, allowing various shortcuts in typing commands;
- A command history facility;
- Automatic connection to the OPS make and run functions, along with production system run statistics;
- An interactive command interpreter for use within user rules and functions; and
- Defining and editing facilities for $PSRL_{OPS8}$ rules, along with extensions to the file

package to save the changes.

The on-line help functions, described in Section 4.4, are also an important part of the user interface. Each of the above areas is now described in turn.

↑
ops8-read

The ops8-read function, alias ↑, is a basic command interaction loop: it reads commands from the user terminal, then either it eval's them as Lisp functions or variables, or it treats them as working memory assertions, doing a make followed by a run command. In the former case it acts like the Lisp top-level, a read-eval-print loop, while in the latter, it treats the input as something to be responded to by the production system. Ops8-read doesn't simply return a value, though; it continues to take in commands until one of several kinds of exit command is read. An '↑' causes a Lisp reset to be done, which returns the user to the Lisp top level. Two ↑D's (ctrl-d) will cause an emergency exit, also returning to the Lisp top level, via (reset). (Typing exit will cause an abrupt exit from Lisp entirely.)

Ops8-read uses lineread to read commands, so that no parentheses are needed to enclose a command (function call). There is an implicit setq feature: typing a Lisp variable (an atom that's boundp) followed by a value causes a setq to that variable; typing a variable alone causes its value to be printed. Unlike the Lisp top-level, nothing is done about quoting, so that "'" (apostrophe) has to be used on literal arguments, for functions that take eval'd arguments. To extend a command over several input lines, end each line with a tab, then return, then continue typing. If you're in doubt about formats, it always works to type the complete unabridged Lisp expression to be eval'd. A final feature is that a schema name (in double-quotes) has a ps done on it. A special case of this is typing an OPS variable (e.g., "<x>"), which will get its value displayed (along with the rest of its schema).

A number of editing and history features exist in uin8. '??' typed anywhere but at the beginning of a line will allow you to edit the current input line, using the Lisp editor; the line will be executed as a command on returning from the editor. The usual history commands are available: ed (to edit a previous command and then re-execute it; refer to commands by number), redo (to simply re-execute a previous command), and ?? (to display the current history - give it a number to display a particular past command). The Lisp variable $ops8-history has a record of past commands, in the form of a list of pairs (n . what-was-read). The user can change the amount of history saved by cons'ing t's onto $ops8-history, or by setting it to its cdr.

Ops8-read has several OPS-specific features. When ops8-read is entered, a line of run statistics is printed. This is also done after each command that fires rules. An automatic make is done if an OPS class (something declared with literalize) begins a command line. After such a make, a run command is automatically done, using the variable $ops8-run-n to determine the number of cycles to run; it starts out at 20; -1 will indicate indefinite firings. When that number of cycles is done, the user has an opportunity to continue or stop (the system prompts for 'y' or 'n'). To get a make of an unliteralized class, start the line with '&' or 'make' . '&' can be used to get several make's on one line; '&&' must be used to get a literal '&'. $incstats is a Lisp variable that controls the printing of statistics mentioned above. A nil value will suppress the statistics. It is initially set to '(-1 -1 -1 0), a list of starting values for various statistics.

Examples of uin8 interactions appear in the appendices of this manual.

```
ed atomic-value
newp rule-name lhs --> rhs
newp rule-name like rule-name
e.g.,
ed rule12
ed 3
ed "autosel"
ed my-eval-fn
newp rule13 (dummy ↑slot1 val1) --> (make dummy ↑slot1 val2)
newp rule14 like rule12
```

The ed function is a general-purpose editor command: it works for functions, schemata (via sedit), history lines, and rules (operating on the OPS8-format rule notation). (If all else fails, an editp is done, allowing an edit of an atom's property list.) The newp function creates a new rule, with the same argument format as p. The Lisp editor is entered with the typed-in line, then when the editor exits, the line becomes a 'p' and the rule is compiled. In case of syntax errors, the user has the opportunity to enter the editor again. (The p function could be used, but there would be no edit opportunity.) It is advisable not to type in 'newp' lines that are too long - Lisp's read buffer is not long enough for some rules. The 'newp like' form copies an existing rule and allows the user to edit that copy, to make a new rule similar to the other one. Use the changes function to list what things have been edited during the present run. Use pp to display rules. The chkpt function will save the recently-changed entities on a file named either chkpt.0 or chkpt.1 (it alternates between them, to allow a backup version).

The p and newp functions have a form of error recovery built in: when a serious error is detected while a rule is being translated, the external form of the rule is put on the 'err-production' property of the rule name. To get this saved information into the right form for re-trying the rule definition, an extra 'ed' command, which will enter the editp function to change the property list of the atom, is done, in which the 'err-production' property name should be replaced with 'production'. A second 'ed' on the rule name will enter the editor to allow the rule contents to be changed, after which another attempt to translate the rule is automatically done.

```
incstats
e.g.,
```

The Lisp function incstats prints incremental statistics; this is done automatically within ops8-read, if $incstats is non-nil (see above).

The user interface also includes a package of utilities named INITF (source: initf.l), including the following features:

- prop - (nlambda) pretty-prints the property lists of atoms, including values and functions;

- chkpt - (no arguments) saves changes to rules, functions and values on file chkpt.0 or chkpt.1 (alternatively); this can be much quicker than saving an entire file of functions (as would dskouts, in the Lisp file package), in the middle of a long interactive session;

- addfn, copyfn, and movefn - (lambdas) maintain fns-lists (described in the standard Lisp file package);

- da - (no arguments) prints the date and time of day;
- agenda - (nlambda) adds strings to the variable %agenda, which is saved by chkpt and can be referenced at a later time;
- oblist searching and other Lisp utilities - other odd useful functions.

Details on these can be found in the source file, located in the PSRL source directory.

4.10. User-defined actions and functions

New rhs elements, or actions, can be defined by creating a schema like the one for write, in Section 3.5.1. That is, a user-defined action has slots RHS-TERMS and PSRL-INTERP-FN, as shown in Figure 4-1.

```
{{ user-action
      PSRL-INTERP-FN: my-interp-user-action
      RHS-TERMS: }}
```

Figure 4-1: user-action Schema

The PSRL-INTERFP-FN slot should be filled with the name of a function that takes one argument, the schema representing the actual action to be performed. That instance of the action schema is produced automatically by the $PSRL_{OPS8}$ translator, making an action similar to the one in Figure 4-2. The user's function can use the PSRL function interp-valuelist to convert PSRL values to Lisp values; it takes one argument, the list in the RHS-TERMS slot, and it does a mapcan using the PSRL function interp-value. Use the PSRL right-hand-side function eval to do the evaluation of Lisp expressions within that list. (A later version of PSRL may allow more variety in the ways that user actions are translated, to correspond to the variety in standard actions.)

```
{{ userexa2
      PSRL-ACTION: "user-action"
      RHS-TERMS: <a list of the arguments given in the rule element> }}
```

Figure 4-2: userexa2 Schema

The user can supply functions to be invoked during matching by using the FUNCTION feature of restrictions, described in Section 3.4.2. Functions used within actions can be invoked with the eval function, described in Section 3.5.2.

4.11. Internal implementation details

This section points out a few functions that are useful when the user is not running PSRL as a whole system, but is interested in certain components only. It also includes pointers to functions that the user may want to adapt or imitate, for modifying the way PSRL works. In general, the PSRL source code is commented to a sufficient degree that someone knowledgeable in Lisp should be able

to follow what is going on. All source files are in /usr/mdr/psrl/src; various versions of the source may exist, so see also /usr/mdr/psrl/READ.ME. Each source file contains a list of functions at the beginning, as well as an abstract describing what kinds of functions are in the file. The interested reader can contact the author for detailed pointers.

```
( match-schema schema-name nil nil nil )
e.g.,
(match-schema "g00013" nil nil nil)
where
     {{ g00013
                INSTANCE: "activity"
                STATUS: pending }}
```

Match-schema takes a schema,[29] interprets it as a PSRL template, and returns a list of schemata in the database that match it. The template must contain at least one SRL relation in order to be matchable. It may have match-variables, if they have been declared beforehand, either by use in a $PSRL_{OPS8}$ rule or by explicitly using the declare-match-variable function (see below). Variables can be initialized (reset by making them "unbound") with the init-vars function, and matching in general should be initialized with init-match8 (which is done automatically by init-psrl). A template may also have range facets, for applying predicates to the values matched.

```
( declare-match-variable schema-name* )
```

This declares one or more schemata to be PSRL match variables.

```
( init-vars '( schema-name* ) )
```

Init-vars initializes a list of variables by setting them to "unbound", and discarding their previous bindings if any (bindings are in the "value" slot).

The following functions might be useful to users who want to customize PSRL in some way:

- fire-m, which is used to initiate the matching of a rule, or the execution of a production-system, in case one's name appears in a rule set;
- fire-x, which is used to execute the actions of a rule;
- interp-action, which is used to perform individual actions;
- interp-value and interp-valuelist, which evaluate individual expressions that are arguments to actions;
- comp-xxx, where xxx is one of a variety of $PSRL_{OPS8}$ syntactic entities; one possibility is to modify that syntax.

[29] The other arguments after the first are used by the PSRL matching routines in connection with matching lhs's, but they are not relevant to match-schema's stand-alone use.

5. Applications and Comparisons

5.1. Applications of $PSRL_{OPS8}$

5.1.1. Callisto

Callisto [7] is a system to support the management of large engineering projects, both in design and prototype production phases. The activity manager in Callisto is a PSRL-based system that aids the manager in tracking and controlling the execution of activities. Managerial heuristics for activity management are represented as PSRL rules. Two production systems have been built to illustrate the activity management system. The first of these, **check-consistency**, goes through the activities stored to check for consistency in the knowledge. The other production system, **project-control**, checks for status of activities and impossibilities generated in situations like the rule above. It can also warn the manager against project-groups or suppliers with tardy performance records.

The following sample rule from the second production system is described first in English:
IF

- An engineering activity uses some parts
- Part P1 is one of them
- Corresponding prototype building activity is M
- Part P1 is needed in quantity Q1
- Current stock is less than Q1
- Lead time for Part P1 > time for prototype

THEN

- Write the message, "Supplier will not be able to supply as fast as we want and current stock will fall short"

The $PSRL_{OPS8}$ representation for this rule is:

```
[p prj-con-3
  { (engineering-activity ↑cause <x> ↑scheduled-finish-time <t1>) <en>}
  { (state ↑type "need" ↑sub-state <y>) <x>}
  { (state ↑type "need" ↑part <p> ↑quantity <q1>) <y>}
  { (manufacturing-activity ↑enabled-by <e> ↑scheduled-start-time <t3>) <m>}
  { (state ↑type "possess" ↑sub-state <s> ↑possess-to-need <x>) <e>}
  { (state ↑type "possess" ↑part <p>) <s>}
  { (revised-part ↑stock-in <s1>) <p>}
  { (stocked-part ↑lead-time {<t2> ! prj-con-3-f1} ↑qty-on-hand {<q2> < <q1>})
      <s1>}
--> (write (crlf) Part <p> was used in design in <en>)
    (write (crlf) For its prototype development in <m>)
    (write (crlf) we need <q1> of <p>)
    (write (crlf) Our current stock is <q2>)
    (write (crlf) and the supplier will not be able to supply as early as)
    (write (crlf) the scheduled start time for <m>) ]
```

5.1.2. HI-RISE

HI-RISE is an expert system for preliminary structural design of high-rise buildings [10, 11]. This involves the synthesis and selection of a feasible structural configuration, under a variety of constraints, e.g., spatial, functional, strength, and serviceability. The synthesis of a feasible alternative involves a depth-first search through the static physical hierarchy for the current functional system under consideration. An alternative consists of an element (or elements) from each level of the physical hierarchy. As an element is added to an alternative, the alternative is checked by heuristic elimination rules. Rule sets are used in HI-RISE to control the order of tasks, to synthesize structural systems, to group elimination rules into classes, and to evaluate alternatives. Rules exploit the inheritance capabilities within PSRL, drawing from values in a tangled hierarchy of design components. Also important is the ability of PSRL to deal with schemata by patterns rather than by names.

One sample rule begins a search through a list of types of 3-dimensional lateral load systems to find a feasible structural configuration. It triggers a demon that will run another rule-set, by making a subgoal and modifying its status to be active. The rule-set is run under fire-cycle control.

```
[p 3D-syn::first-type
  { (goal ↑name "3D-syn" ↑grid-item <lateral> ↑type-list nil ↑status active)
        <goal>}
  { (3D-lat-types ↑types <2D>) <3D-types>}
-->
   (make 3D-lat-types ↑types (psrl-cdr <3D-types> types))
   (cbind <3D-lat>)
   (modify <goal> ↑type-list <3D-lat>)
   (make 3D-lateral ↑is-alt <lateral> ↑3D-description <2D>)
   (cbind <3D-sys>)
   (make goal ↑name "3D-elim" ↑grid-item <3D-sys> )
   (cbind <goal2>)
   (modify <goal2> ↑status active) ]
```

A second sample rule illustrates the creation of vector attributes and triggers a demon to evaluate a constraint. This is a common construct used by HI-RISE to switch from a production system to a Lisp function for algorithmic calculations. Its rule-set is run under fire-sequence control.

```
[p place-rframe::nedges
  { (goal ↑name "place-rframe" ↑grid-item <material>) <goal>}
  { (material ↑direction narrow ↑stories <s> ↑narrow-bays <n> ↑wide-bays <w>)
        <material>}
-->
   (make location ↑is-alt <material> ↑number 2
                  ↑class nedges
                  ↑n1 2 2
                  ↑n2 (compute 2 * <n>) (compute 2 * <n>)
                  ↑w1 1 (compute (2 * <w>) + 1)
                  ↑w2 nil nil
                  ↑s1 2 2
                  ↑s2 (compute 2 * <s>) (compute 2 * <s>))
   (cbind <loc>)
   (make constraint ↑uses <loc> ↑name rframe-ok)
   (cbind <constr>)
   (modify <constr> ↑evaluate yes) ]
```

5.1.3. KBGraphics

Autograph [12] is an expert system for doing knowledge-based graphics. Given a set of application data, expressed as SRL schemata, Autograph will draw a picture that effectively presents the data to the user. Autograph does this by mapping high-level, non-graphical schemata into low-

level, graphical schemata. The mappings between different levels are performed by sets of PSRL rules. Alternative pictures of the same data are drawn by selecting alternative rule-sets. Rule-sets are selected based on the current display context, which includes information about the current user's display preferences and task objectives.

A sample rule follows:

```
; Move an activity to the right of all its previous activities.

[p adjust-subactivities-x-p
    ; The current task is to adjust the x-coordinates of an activity's
    ; subactivities:
    (task ↑type "adjust-subactivities-x" ↑object <activity> ↑state active)
    ; This might have to be moved:
  { (activity ↑subactivity-of <activity> ↑start-time <start-time> ↑x <x>)
        <subactivity> }
    ; This is the previous activity:
    (activity ↑next-activity <subactivity> ↑subactivity-of <activity>
        ↑finish-time <= <start-time> ↑x {<pre-x> >= <x>})
-->
    ; Move <subactivity> to the right of the previous activity.
    (modify <subactivity> ↑x (compute <pre-x> + 1)) ]
```

5.2. Comparisons to other systems

Comparisons among rule-based systems embedded in representation languages can involve the following questions:

- pattern power: what primitives are provided, what is the scope of database access allowed, how can the user control that scope, and the logical power allowed in patterns;
- action power: what are the primitives and what is their scope of activity;
- rule organization: how are rules and rule-sets structured;
- Working Memory power: viewed as a declarative knowledge representation facility;
- control options: what varieties of conflict resolution are available; is there some other non-recognize-act control cycle; can fuzzy reasoning or other uncertainty be handled; can backward and forward chaining be done;
- efficiency;
- features of the implementation:
    - base language and its portability,
    - existence of a mini-version (for learning or experimentation),
    - continuing evolution of the system,
    - modularity of implementation (impacts on re-configurability, etc.),
    - internal documentation (adaptability, customization),

- size of code, when expressed as text, and
- hardware requirements;

- ability of rules to manipulate rules themselves, which bears on whether learning is a real possibility;
- extensibility (e.g., PSRL allows user-specified actions, Lisp predicates);
- external notation vs. internal: is the language compiled, and is the notation user-modifiable;
- width of domains where it has been successfully applied (which indicates how far along it is with respect to evolving into a comfortable environment with appropriate programming power);
- user facilities: explanation, summary, debug, trace, edit, undo, knowledge acquisition, abstraction (formation of modules in knowledge base), experimentation, systematic testing;
- software maturity: error recovery, error correction, error recognition;
- software support;
- useable for implementing other AI architectures, e.g., multi-layer planning, blackboard, best-first search, problem space / universal weak method, object-oriented.

Two languages in the same class as PSRL can now be examined. This comparison can consider only some of the above dimensions due to lack of complete information on the other systems. PSRL can be compared to LOOPS [1] on the following dimensions:

- pattern power: PSRL can match by content, where LOOPS is limited to matching units by name; PSRL has a cleaner and more uniform notation (e.g., meta-information is stored in matchable schemata);
- control options: PSRL is similar;
- efficiency: LOOPS is probably faster, stemming from its restricted pattern power;
- declarative Working Memory power: SRL is more powerful;
- rule manipulation: PSRL probably has the edge here, due to its uniform internal representation and simple notation - LOOPS rules are apparently lists, not units.

PSRL can be compared to ROSIE [4] on the following dimensions:

- pattern power: PSRL is similar; ROSIE has a more readable notation, though perhaps not so easily written;
- control options: PSRL is similar;

- efficiency: PSRL is similar, apparently, with one to ten seconds required to select and execute a rule;

- declarative Working Memory power: SRL is more powerful;

- rule manipulation: PSRL probably has the edge here - it's not clear how ROSIE represents rules.

HSRL [16], a PROLOG-like inference system that can be used for database queries, is a facility within SRL that has some overlap of functionality with PSRL. The following points are offered for comparison:

- pattern power: similar patterns can be expressed in both PSRL and HSRL, but notationally, HSRL requires more characters to get across the same idea; HSRL also requires entry into Lisp for some of the match predicates built into PSRL; another disadvantage of HSRL is the treatment of negation: apparently to negate a template (in PSRL terms) requires a separate axiom in HSRL that fails when all its literals match, and whose left-hand side is included in the clause where the negation is to take effect; due to these properties, HSRL will be much less readable, more difficult to edit, and thus less appropriate for expert system applications (unless a translator and a higher-level notation were developed); also with respect to patterns, PSRL has a better means for controlling the SRL context in which matching takes place;

- action power: the basic operations in PSRL are at a higher level than HSRL provides (which are at the level of manipulating values, slots and schemata individually); as in the preceding point, this will make HSRL less concise and more difficult to read and manipulate;

- rule organization: the two are similar in functionality here, except that PSRL allows rule sets to be related hierarchically; PSRL's organization is based on schemata, and thus more flexible, while HSRL's simply uses lists;

- control options: HSRL allows only one axiom set to be active at a given time, where PSRL allows many, and allows them to be nested (statically and dynamically); PSRL allows a variety of control regimes, where HSRL has a single fixed control strategy;

- efficiency: HSRL is faster, by a factor of six;

- user facilities: PSRL has an interface for rule editing, etc., while HSRL's interface includes only querying and control capabilities; HSRL has the advantage of using the DYPAR package for natural-language interaction (in at least one application this has been done, but its full capabilities are not known to the author at present);

- rule manipulation: PSRL is much more flexible here, allowing easier approaches to learning.

Thus, in relation to HSRL, PSRL provides a more powerful environment for many types of knowledge engineering, from the point of view of representation. On the other hand, HSRL will be more appropriate for some kinds of database querying, due to its speed advantage. Another factor is that HSRL, due to its PROLOG basis, is not useful for situations where non-monotonic reasoning is

involved. For instance, forming a total of the values of slots within a given set of schemata will require HSRL to make use of schema side-effects, doing the main work outside of its space of logical assertions. Thus applications using HSRL will necessarily be a mixture of approaches, and the advantage of having a knowledge base in a uniform notation will be lost.

In summary, in relation to LOOPS and ROSIE, PSRL has full pattern-matching power, a clean, uniform notation (though not as readable as English-like formats), and has acceptable efficiency, though improvement is needed. The main advantage of PSRL seems to derive from its underlying representation system, SRL, which has significant facility in the areas of user-defined relations, user-defined inheritance, default values, demons, capabilities for large databases, database contexts, meta-information (useable, e.g., for audit trails and other explanation demands), and HSRL, which could be used within PSRL for logically modifying data accesses. Since rules are represented as schemata, there is a potential for learning operations to be rule-based. PSRL allows the user to define new match predicates and new actions.

6. Conclusions

It has been the research focus of this work to determine whether rule-based pattern-matching data access and control can be workable when embedded in a state-of-the-art representation language. Not only do we want rule-based capabilities, but they must be well-integrated and must give the user access to the full power of the underlying mechanisms. $PSRL_{OPS8}$ has gone a considerable distance towards these goals, and has stimulated, while under development, several important applications. PSRL viewed as a procedural component is a good complement to the declarative SRL language. Work is continuing towards better efficiency and on further exploitation of possible synergies between the two representations. There are two other systems that seem to be aiming at the same level of power as PSRL, which have been discussed above and with which PSRL compares favorably.

In addition to considering PSRL as a complete language environment, it can be viewed as a powerful package of facilities to be used within SRL. It includes basic schema manipulation functions (make, modify, et al.), the psorts package for allowing modular use of the matching capabilities (not yet fully implemented, sketched in Appendix 7.2), functions for associative pattern-based retrieval, and programmable procedures at a higher level than Lisp.

6.1. Future developments

Research possibilities exist in several areas:

- Implementation level: improving efficiency, perhaps through RETE nets as are used in OPS5; making the context facility more accessible, allowing rules to match and manipulate rule-sets; deeper embedding in the SRL implementation, for efficiency; monitoring database changes more directly; and monitoring and limiting computing resources within the match, for use in a multi-processing search environment.

- User facilities: rule library maintenance; friendly user interface; on-line help and training; examples of use on larger problems.

- Further integration with SRL: more precise control of inheritance paths; access to elaborations; demons and translation of monitoring rules into demons; hierarchies of literalizes; semantic or partial matching; access to contexts.

- Knowledge acquisition and management: better interactive specification of rules; help in formulating expertise for expert systems; new features as required for new domains of application; consistency checking.

- Explanation: basic facilities; use of meta-information to form audit trail.

- Exploration of other architectures, by implementing some examples (some of these are very close to PSRL): object-oriented, message-based; constraints; simulation; blackboard model; agenda-based problem-solving.

APPENDICES

7. APPENDICES

7.1. Listing of psrl.init file

```
(*** "psrl.init - for PSRL Version 1.2"
     "How to use this file:"
     "Most users will want to take the contents here and adapt them to their"
     "  own purposes; usually this will involve merging them into their own"
     "  init files for loading ops rules; some rearrangement may be needed,"
     "  due to the (reset) at the end of this file - see note there")

(*** "Recent changes here: %changes line; compl8 dskin; $-switch setting"
     "  more: help8 dskin; %changes nil moved to end")

(msg (N 1) "Loading initf and uin8 ... ")
(print (fasl '/usr/mdr/psrl/bin/initf))
(print (fasl '/usr/mdr/psrl/bin/uin8))

(msg (N 1) "Connecting to basicps = basic + psrl database: ")
(*** "The basicps database is created by loading psrldb.upd - user may want"
     "  to create his/her own merged basic by the same means.")
(print (db-connect '/usr/mdr/psrl/bin/basicps))
(terpri)

(msg (N 1) "Loading psrl functions ... ")
(print (load '/usr/mdr/psrl/bin/ops8)) (drain)
(print (load '/usr/mdr/psrl/bin/help8)) (drain)
(print (load '/usr/mdr/psrl/bin/compl8)) (drain)
(print (load '/usr/mdr/psrl/bin/match8)) (drain)
(print (load '/usr/mdr/psrl/bin/act8)) (drain)

(*** "The following fixes bad effects of using full paths in dskin args")
(setq %changes nil)
(msg (N 1) "Initializing PSRL-OPS8 ... ")
(print (init-ops8))

(*** "Here is where to add settings for SRL switches"
     "PSRL variables can be set here also")
(setq $db-cache-keep (max 2000 $db-cache-keep))
(setq $db-cache-max (max 2400 $db-cache-max))
(setq prinlength (max prinlength 30)) (setq prinlevel (max prinlevel 7))

(*** "The following allows the changes to top level in initf to take effect:"
     "Note: this init file can't be loaded by another file without aborting"
     "  the rest of that file's commands, due to the effects of reset")
(reset)
```

7.2. Proposed PSorts Package

Pattern-based Schema Sorting

M. Rychener

Preliminary Specification

1 Nov. 1983

Revised 7 Nov. (new fn names, argument conventions)
Revised 21 Nov. (agenda, examples, ''s in fn calls indicating eval)
Revised 30 Nov. (more details on arg. defaults, "ps-collect-pattern-n")

This package is to provide a set of manipulation functions for lists of schemata. The user would set up a list (either with these functions or with others outside of PSRL), assign it to a Lisp variable, then proceed to combine, filter, and otherwise operate on it to produce a select list for some further purpose. The basic pattern expression language is that of $PSRL_{OPS8}$, and the package is considered to be an extension of PSRL. The principle is similar to that used in the PSRL function matches, which takes a list of conditions and shows how they match the current database.

Schedule for implementation: after the current PSRL version 1.2 is announced.

FUNCTIONS

```
    Arguments:
        Optional arguments on almost all fns:
            '<sort-pat> '<include-pat> '<exclude-pat>
        <sort-pat> is a list of slot-names (strings), whose order indicates
            their priority in sorting; numeric, atomic and string values
            are sorted by the usual lessp or alphalessp; for lists, there is
            a search in the car direction until (atom (car list)) is true;
            If empty, or in case of ties, schema names are compared.
        <include-pat> is an <ops8-form> or a schema returned by ps-mk-pattern;
            if empty, all are included (unless explicitly excluded).
        <exclude-pat> is an <ops8-form> or a schema returned by ps-mk-pattern;
            if empty, none are excluded.
        If t is given for a pattern, a corresponding pattern is used from the
            schema "ps-collect-pattern", which is set up by ps-collect or
            ps-sort, with slots "sort", "include", and "exclude".
            > actually, "ps-collect-pattern-n", where n is incremented, to
                allow essentially a stack of ps-collect schemata (convenience)
            > the slot "comparefn" will also be set up, to do "sort" pattern
                efficiently - used within Lisp sort and insert
        All functions return a sorted list of schemata, unless otherwise noted,
            and also place the result in the "list" slot of the schema
            "ps-collect-pattern".
    (ps-mk-pattern <ops8-form>)
        - converts a pattern (in PSRL_OPS8 form) into a schema (of PSRL
            type template), returning the name of the schema
        - argument uneval'd, as in p and matches
        - eg, (ps-mk-pattern (goal ↑status active))
        - eg, (ps-mk-pattern (↑color brown ↑size { > 20 < 30} ↑fur long))
    (ps-collect '<sort-pat> '<include-pat> ['<exclude-pat>])
        - <include-pat> must contain at least a relation.
        - eg, (ps-collect '("size" "color") "g00017" '(↑size >= 30)), where
            "g00017" = (ps-mk-pattern (dog ↑color << brown tan >> ↑fur short))
    (ps-sort '<list> ['<sort-pat> '<include-pat> '<exclude-pat>])
        - if null <list>, <include-pat> must contain a relation, and the effect
            is like ps-collect
        - given <include-pat> or <exclude-pat>, serves as a filter, too
            (implying at least one pass over the list to ensure sortedness)
    (ps-insert '<schema> '<list> ['<sort-pat> '<include-pat> '<exclude-pat>])
        - adds a <schema> to the list, maintaining sorted order (assumed),
            unless the include- or exclude- patterns are not satisfied;
    (ps-merge '<list1> '<list2> ['<sort-pat> '<include-pat> '<exclude-pat>])
        - filters and merges two lists together; assumes sorted inputs
    (ps-intersect '<list1> '<list2> ['<sort-pat> '<include-pat> '<exclude-pat>])
        - filters and takes set intersection; assumes sorted inputs
    (ps-union '<list1> '<list2> ['<sort-pat> '<include-pat> '<exclude-pat>])
        - filters and takes set union; assumes sorted inputs
    (ps-difference '<list1> '<list2>
                   ['<sort-pat> '<include-pat> '<exclude-pat>])
        - filters and takes set difference; assumes sorted inputs
```

```
------
AGENDA
------

    post a notice about current revisions
    add notices about sorts to doc/news
    dla has some kind of useful function? (callisto mbox)
```

7.3. Basics

This appendix has a script of some of the basic PSRL display, execution, and translation facilities.

```
Script started on Sun Jan  8 13:20:57 1984
L
L pwd
/usr10/mdr/psrl/test
L
L islisp5 -t
[Initializing, please wait]

ISLisp 5.25, 12-18-83, from Franz Lisp, Opus 38

   Copyright (c) 1983 CMU Robotics Institute
```

The following loads the psrl.init file, which in turn loads the PSRL object files and does initialization.

```
1.load psrl.init

Loading initf and uin8 ... [fasl /usr/mdr/psrl/bin/initf]
t[fasl /usr/mdr/psrl/bin/uin8]
t
Connecting to basicps = basic + psrl database: t

Loading psrl functions ... [fasl /usr/mdr/psrl/bin/ops8.o]
t[fasl /usr/mdr/psrl/bin/help8.o]
t[fasl /usr/mdr/psrl/bin/compl8.o]
t[fasl /usr/mdr/psrl/bin/match8.o]
t[fasl /usr/mdr/psrl/bin/act8.o]
t
Initializing PSRL-OPS8 ...

PSRL Version 1.2 - 7 Jan. 1984

Type ? for help; type ↑ for Ops8 user interface.
Done
```

There are a number of help functions in PSRL, all of which start with the '?' character. The following shows what they print.

```
2.?

Type the following, for help with PSRL-OPS8 and User Interface (UIN8):
  ?e - the Lisp editor
  ?i - INITF utilities
  ?p - UIN8 Parameters (globals)
  ?u - the UIN8 User Interface
  ?5 - OPS5
  ?b - hints on reporting bugs
  ?8 - PSRL-OPS8 functions
  ?8p - PSRL-OPS8 parameters (globals)
  ?8v - PSRL-OPS8 values of SRL and PSRL parameters
t
3.?8

PSRL-OPS8 has the following main functions (in addition to many OPS5 ones):
  tlps <ps-name> [<rule-file-extension>] [<literal-file-extension>]
        translates and loads ops5 rules, after doing init.; see $ops8-exts
  lps <ps-db-name> . . . - connects to databases for PSs, does init.
  fire '<rule> - matches lhs and executes rhs if the match succeeds.
  matches <rulename> - matches lhs of rule and prints trace
  matches <ops8-template> . . . - translates and matches templates as lhs
```

```
  init-ops8 (alias init-psrl) - does initialization.
  run [<ps-name>] [<number>] - runs ps-name or $ops8-ps; uses control slot,
      or prompts; stops after <number> firings, unless absent or -1 specified.
  wm <class> . . . - displays all of $wm or selected classes of items.
  pm <rule> . . . - displays ops5-like format for rules encoded as schemas.
  control <fn> - control slot of $ops8-ps becomes <fn>, user-defined or one of
      fire-cycle[-reverse], fire-cycle-one, fire-one[-reverse], fire-sequence.
  production-system <ps-name> - declares ps-name to be the current one,
      eg, place it at head of file; tlps does it automatically
  backtrace - prints out the last few rule firings; accesses $psrl-backtrace
  declare-match-variable <var-schema> . . . - creates slots for the vars.
  watch - works on levels -1 0 1 2 3 4; no args gets current setting
  make, modify and Remove (ops5) can be used at the top level to do WM actions
  psrl-version - current PSRL-OPS8 version number.
t
4.?8p

PSRL-OPS8 has the following global parameters:
  $psrl-instance - relation between WM element and class; def. 'psrl-instance'
  $psrl-instance+inv - inverse of $psrl-instance; default 'psrl-instance+inv'
  $ops8-ps - current production-system schema; set by production-system
    (a declaration) or in run (recursively); its stack property has old values
  $ops8-exts - default extensions for arguments to tlps - a list
    (ops5-rule-file-ext literalize-file-ext-(can be nil)); default '(op5 lit)
  $trace-match - if t, prints out trace information during matching.
    - if set to list, traces only rules in the list
    - if set to list of non-rule, only traces conflict resolution.
  $debug-fire - if t, enables detailed trace of top level execution.
  $debug-match - if t, enables detailed trace of matching.
  $psrl-silence - t suppresses most status-reporting 'politeness' messages.
  $psrl-context - the SRL context used for matching (WM, not rules).
  $match-inherit - a non-nil <path> turns on inheritance, for matched schemata.
  $psrl-inherit-slots - non-nil <path> forces slots to be inherited, no slotc.
  $psrl-control-step - if t, a pause is forced after each rule firing.
  $demon-run - non-nil means OK to run demons; user demon fns must test it;
      default value is t, but is prog-set to nil by the p function.
  $fire-cycles - is set by run, to convey num. argument to 'fire' functions.
t
5.?8v

PSRL-OPS8 and SRL global parameters have these values:
  $db-cache-max = 2400    $db-cache-keep = 2000    $queuelength = 1
  $default-context = $root-context  db-get-all-db = (/usr/mdr/psrl/bin/basicps)
  $demon = nil    $default = nil    $restrict = nil
  $meta-schema = schema    $meta-slot = relation    $meta-value = value
  $psrl-instance = psrl-instance    $psrl-instance+inv = psrl-instance+inv
  $ops8-ps = nil    $ops8-exts = (op5 lit)    $psrl-watch = 1
  $psrl-silence = nil  $trace-match = nil  $debug-fire = nil  $debug-match = nil
  $psrl-context = nil    $match-inherit = nil    $psrl-inherit-slots = nil
  $psrl-control-step = nil    $demon-run = t
t
6.?b

Please include the following in bug reports, to ensure a prompt reply:
  The name of your script file (include complete directory and machine)
  A segment from your script file, including the bug and the following:
    If inside a Lisp debug loop (prompts ':'), do the commands: pp, bkev
    pm and pp of rules involved
    ps of rule template(s) involved
    ps or wm of schemas involved (from wm or elsewhere in database)
    ?8v, to show global parameters' values
  Send bug reports to mdr@d
t
7.?

Type the following, for help with PSRL-OPS8 and User Interface (UIN8):
```

```
  ?e - the Lisp editor
  ?i - INITF utilities
  ?p - UIN8 Parameters (globals)
  ?u - the UIN8 User Interface
  ?5 - OPS5
  ?b - hints on reporting bugs
  ?8 - PSRL-OPS8 functions
  ?8p - PSRL-OPS8 parameters (globals)
  ?8v - PSRL-OPS8 values of SRL and PSRL parameters
t
8.?u

UIN8 (User Interface OPS5 / OPS8) has the following main functions:
  ↑ (alias: ops8-read) - the main read-eval-make-run-print loop
    - similar to Lisp toplevel, with addition of doing make if OPS class first
    - after make, a run command is automatically done; see $ops8-run-n
    - lineread is used; NO auto quoting; extend input line by ending with tab
    - to get make of unliteralized class, prefix with '&' or 'make'
    - & can be used to get several make's on one line; && to get literal &
    - ?? anywhere but 1st, to edit before doing whatever the line specifies
    - ↑ to exit to Lisp top-level
    - Lisp-variable, then value implies setq; ()s on fn-call optional
    - schema name in double-quotes implies a ps on it
    - ed, redo, ?? are the usual history commands; see $ops8-history
  incstats - prints incremental statistics
  ed - a general editor: works for functions, schemas, and rules
  newp - creates new rule, same argument format as p
  newp <name> like <rule-name> - make & edit new rule similar to existing rule
  changes & pp will display edited fns, rules; chkpt will save them on a file
t
9.?p

UIN8 has the following global parameters:
  $incstats = non-nil causes print-out of incremental statistics
    - init. is (-1 -1 -1 0), a list of starting values of statistics
  $ops8-history = record of entities read in (n . what-was-read)
    - change length by cons'ing on t's, or by setting it to its cdr
  $ops8-run-n = number of cycles to run, on each make input; default 20
t
10.?i

INITF utilities include these functions (see initf.l for doc):
  #print addfn agenda chkpt copyfn da di dt ed more movefn pp-prina prop
  searchob searchobu ty
t
11.?5

OPS5 has the following top-level functions:
  make Remove openfile closefile default call run ppwm wm cs
  matches strategy watch pbreak exit excise back
OPS5 has the following declarations:
  p literalize vector-attribute literal external
OPS5 has the following actions:
  make Remove modify openfile closefile default write (crlf tabto rjust)
  call halt bind cbind build
OPS5 has the following functions:
  substr genatom compute litval accept acceptline
OPS5 has the following user-defined functions and actions:
  $parameter $parametercount $assert $tab $value $reset $ifile $ofile
  $varbind $litbind $eql $symbol $intern $cvan $cvna
t
12.?e

p or pp - print current expression
n - go to n'th subexpression; if n < 0, count back from end
0 - go up a level & back to beginning of containing list; up - 0 but not back
```

```
nx - go to next expression; bk - go back one expression; ↑ - go to topmost list
undo - take back the previous command (repeats go further back) (try undo undo)
ok - exit the editor
f exp - find first occurrence of exp, searching forward; bf searches backward
e (lisp expr) - evaluates and prints the value of the Lisp expression
(n e1 e2 ...) - replace n'th element with e1 e2 ... (if no e's, deletes n'th)
(-n e1 e2 ...) - insert e1 e2 ... before n'th element
r e1 e2 - replace e1 by e2 throughout the current expression
n e1 ... - add (appeNd) e1 ... to end of current expression
b e1 e2 ... - insert before current; a inserts after current
: e1 e2 ... - replace current expression with e1 e2 ...
mbd (... * ...) - embed current expression (*) in a larger structure (...)
xtr n - replace current expression by (extracting) its n'th element
move n to before m - move n'th element of current expression to before m'th
copy n to before m - make a copy of n'th before m'th; after works too
sw m n - swap the m'th and n'th subexpressions of current
bi n m - put in a pair of ()s, before n'th, after m'th; bo n takes them out
t
13.?

Type the following, for help with PSRL-OPS8 and User Interface (UIN8):
  ?e - the Lisp editor
  ?i - INITF utilities
  ?p - UIN8 Parameters (globals)
  ?u - the UIN8 User Interface
  ?5 - OPS5
  ?b - hints on reporting bugs
  ?8 - PSRL-OPS8 functions
  ?8p - PSRL-OPS8 parameters (globals)
  ?8v - PSRL-OPS8 values of SRL and PSRL parameters
t
```

The following goes through some simple declarations and then uses them in doing some simple WM manipulations.

```
14.literalize dummy att1 att2
1
"dummy"
15.literalize dummy2 att1  att2
1
"dummy2"
16.make dummy ↑att1 val1 ↑att2 val2

"g00008"
```

Make results in a new schema with a dummy (gensym) name. The slots are defined with the literalize statements. The wm function, below, displays the schemata in OPS5-like format, and returns a list of the internal names of the schemata. The segment continues with a number of simple changes to WM elements.

```
17.ps "g00008"
{{ g00008
    psrl-instance: "dummy"
    att1: val1
    att2: val2}}

nil
18.wm

(dummy  ↑att1 val1  ↑att2 val2)
("g00008")
19.modify g00008 ↑att2 val3

"g00008"
```

```
20.wm

(dummy  ↑att1 val1  ↑att2 val3)
("g00008")
21.watch 4

4
22.modify g00008 ↑att1 val4

Modify:
(dummy  ↑psrl-instance+inv psrl-result-element  ↑att1 val1  ↑att2 val3)
(g00008  ↑att1 val4)
(dummy  ↑psrl-instance+inv psrl-result-element  ↑att1 val4  ↑att2 val3)
"g00008"
23.wm

(dummy  ↑att1 val4  ↑att2 val3)
("g00008")
24.make dummy2 ↑att1 val1 ↑att2 val6
 sc:g00009
"g00009"
25.wm

(dummy  ↑att1 val4  ↑att2 val3)
(dummy2  ↑att1 val1  ↑att2 val6)
("g00008" "g00009")
26.Remove g00009

Remove:
(dummy2  ↑att1 val1  ↑att2 val6)
("g00009")
27.wm

(dummy  ↑att1 val4  ↑att2 val3)
("g00008")
28.ps "g00009"

nil
29.make dummy2 ↑att1 val7 ↑att2 val9
 sc:g00010
"g00010"
```

The next segment shows how the PSRL matcher works. The matches function creates a dummy rule, translating the user input into schemata, and then tries to match the rule's condition. Detailed tracing is turned on here, and within matches, a more detailed tracing is done automatically.

```
30.watch

Watch level = 4, $psrl-silence nil, $trace-match t, $debug-fire/match t/t

nil
31.watch 3

3
32.watch

Watch level = 3, $psrl-silence nil, $trace-match nil, $debug-fire/match nil/nil

nil
33.matches (dummy ↑att1 <x>)
PS:"generic-psrl-ps"
{{ g00011
    instance: "rule"
    psrl-lhs: "g00011c1"
    psrl-rhs:}}
```

```
Attempting to match g00011
LHS of g00011 = (g00011c1)

Comp-vars: vars in lhs and rhs: (<x>)

Matching g00011c1t in rule g00011
Matching to template g00011c1t ( psrl-instance dummy )

Candidate schemas to be matched: (g00008)
  Checking slots: (att1)
  Checking schema: g00008
    Binding (<x>) to (val4)

New conflict (g00011 val4)
Update-conflicts adding: (g00011 val4)

Result of PSRL match = $psrl-conflicts:

Conflict set, with dominant entry first:
("g00011" val4)
"g00011"
34.ps "<x>"
{{ <x>
    psrl-instance: "match-variable"
    value: $unbound$}}

nil
35.wm

(dummy  ↑att1 val4  ↑att2 val3)
(dummy2  ↑att1 val7  ↑att2 val9)
("g00008" "g00010")
```

Notice above that after the matching, the internal match-variable "<x>" is reset to be unbound. The user finds out what matched by looking at the conflict set display, which has a rule name followed by bindings for its match variables.

The next segment shows how PSRL translates rules from OPS8 format into schemata. The tlps function is reading in a file, autosel.op5 (see next appendix for a more complete explanation of the meanings of the rules), and the trace output shows its progress in translating each rule in that file in turn. The detailed trace output includes schema creations (marked by "sc:") and beginnings of production systems (marked by "PS:"). First are a bunch of literalize declarations, then the rules themselves.

```
36.watch 4

4
37.tlps autosel

[Storage: List 9449, symbol 4286, string 19, binary 2066, array 13, value 128, hunk nil]

[13h:35:41 Elapsed 0, Process 0, Fired 0 (avg  msec)]

Before loading, there are 0 rules, 10 OPS8 schemata.

Do you want to save these on a database after translation? (y n):y

Connecting to autosel database ... tPS:"autosel" sc:autosel
Literal file: autosel.lit sc:buy
literalize (buy dollars)
{{ buy
```

```
    psrl-instance: "literalize"
    dollars:}}
 sc:help
literalize (help with)
{{ help
    psrl-instance: "literalize"
    with:}}
 sc:steel
literalize (steel what amount)
{{ steel
    psrl-instance: "literalize"
    what:
    amount:}}
 sc:clock
literalize (clock mod timer maker)
{{ clock
    psrl-instance: "literalize"
    mod:
    timer:
    maker:}}
 sc:auto
literalize (auto mod serial clock)
{{ auto
    psrl-instance: "literalize"
    mod:
    serial:
    clock:}}
 sc:automac
literalize (automac mod serial machine clock)
{{ automac
    psrl-instance: "literalize"
    mod:
    serial:
    machine:
    clock:}}

Rule file: autosel.op5 sc:buy2 sc:buy2c1 sc:buy2c1t sc:<a> sc:<g>
sc:buy2c2 sc:buy2c2t sc:<a2> sc:<s> sc:buy2a1 sc:buy2a2 sc:buy2a3
Comp-vars: vars in lhs and rhs: (<a2> <a> <g> <s>)

[p buy2
  { (buy ↑ dollars { <a> > 0 }) <g> }
  { (steel ↑ what scrap ↑ amount <a2>) <s> }
--> (modify <g> ↑ dollars (compute 0 - <a>))
    (modify <s> ↑ amount (compute <a> + <a2>))
    (make clock ↑ mod going ↑ timer 0 ↑ maker 0) ]

lhs:   (buy2c1 buy2c2)
{<g> (buy  ↑dollars {<a> (> 0)})}
{<s> (steel  ↑what scrap  ↑amount <a2>)}
rhs:   (buy2a1 buy2a2 buy2a3)
(modify <g>  ↑dollars (compute 0 - "<a>"))
(modify <s>  ↑amount (compute "<a>" + "<a2>"))
(make clock  ↑mod going  ↑timer 0  ↑maker 0)
```

Immediately above, the system first prints out the external format of the rule as it was read in, then prints a newly-generated version of it, based on the schemata created to represent it. When detailed tracing is in effect, the user has an opportunity to interact after each rule is defined, as shown next. The '[]' line is printed out by PSRL's user interface reading function.

```
Debug p interrupt; type ok to continue:
[13h:36:33 Elapsed 0:52, Process 0:18.7 (GC 0%), Fired 0 (avg  msec)]
↑1 ok

[13h:36:56 Elapsed 0:23, Process 0:00.1 (GC 0%), Fired 0 (avg  msec)]
```

```
 sc:beg sc:begc1 sc:begc1t sc:<t> sc:begc2 sc:begc2t sc:begc3 sc:begc3t
 sc:bega1 sc:<n> sc:bega2
Comp-vars: vars in lhs and rhs: (<n> <t>)

[p beg
    (clock ↑ mod going ↑ timer <t>)
  - (automac ↑ mod busy ↑ machine engine)
    (steel ↑ what scrap ↑ amount > 4)
--> (bind <n>)
    (make auto ↑ mod order ↑ serial <n>) ]

lhs:   (begc1 begc2 begc3)
(clock  ↑mod going  ↑timer <t>)
- (automac  ↑mod busy  ↑machine engine)
(steel  ↑what scrap  ↑amount  (> 4))
rhs:   (bega1 bega2)
(bind   ↑variable <n>  ↑rhs-terms nil)
(make auto  ↑mod order  ↑serial <n>)

Debug p interrupt; type ok to continue:
[13h:37:14 Elapsed 0:18, Process 0:14.2 (GC 0%), Fired 0 (avg  msec)]
↑2 "begc1"
{{ begc1
    instance: "lhs-element"
    template: "begc1t"}}

↑3 "begc1t"
{{ begc1t
    template+inv: "begc1"
    psrl-instance: "clock"
        [instance: "psrl-instance"]
        [domain: "begc1t"]
        [slot: "psrl-instance"]
    mod: going
        [instance: "relation"]
        [domain: "begc1t"]
        [slot: "mod"]
    timer: "<t>"
        [instance: "relation"]
        [domain: "begc1t"]
        [slot: "timer"]}}

↑4 "begc2"
{{ begc2
    instance: "lhs-element"
    template: "begc2t"
    negation: t}}

↑5 "begc3t"
{{ begc3t
    template+inv: "begc3"
    psrl-instance: "steel"
        [instance: "psrl-instance"]
        [domain: "begc3t"]
        [slot: "psrl-instance"]
    what: scrap
        [instance: "relation"]
        [domain: "begc3t"]
        [slot: "what"]
    amount: {}
        [instance: "relation"]
        [domain: "begc3t"]
        [slot: "amount"]
        [range: (> 4)]}}

↑6 "bega1"
```

```
{{ bega1
    psrl-action: "bind"
    variable: "<n>"
        [instance: "relation"]
        [domain: "bega1"]
        [slot: "variable"]
    rhs-terms:
        [instance: "relation"]
        [domain: "bega1"]
        [slot: "rhs-terms"]}}

↑7 ok

[13h:38:34 Elapsed 1:20, Process 0:01.0 (GC 0%), Fired 0 (avg  msec)]
 sc:clu sc:cluc1 sc:cluc1t sc:cluc2 sc:cluc2t sc:<o> sc:clua1
Comp-vars: vars in lhs and rhs: (<o>)

[p clu
    (clock ↑ mod stopping)
    (auto ↑ mod order ↑ serial <o>)
--> (write Unfinished <o> (crlf)) ]

lhs:   (cluc1 cluc2)
(clock  ↑mod stopping)
(auto  ↑mod order  ↑serial <o>)
rhs:   (clua1)
(write   ↑rhs-terms (Unfinished "<o>" (crlf)))

Debug p interrupt; type ok to continue:
[13h:38:44 Elapsed 0:10, Process 0:07.2 (GC 0%), Fired 0 (avg  msec)]
↑8 ok

[13h:38:54 Elapsed 0:10, Process 0:00.0 (GC 0%), Fired 0 (avg  msec)]
 sc:eng sc:engc1 sc:engc1t sc:engc2 sc:engc2t sc:engc3 sc:engc3t
 sc:engc4 sc:engc4t sc:engc5 sc:engc5t sc:<q> sc:enga1 sc:enga2
Comp-vars: vars in lhs and rhs: (<n> <o> <q> <s> <t>)

[p eng
  { (auto ↑ mod order ↑ serial <n>) <o> }
  - (automac ↑ serial <n> ↑ machine engine)
  - (automac ↑ mod busy ↑ machine engine)
    (clock ↑ mod going ↑ timer <t>)
  { (steel ↑ what scrap ↑ amount { <q> > 2 }) <s> }
--> (make automac ↑ mod busy ↑ serial <n> ↑ machine engine ↑ clock
        (compute <t> + 7))
    (modify <s> ↑ amount (compute <q> - 3)) ]

lhs:   (engc1 engc2 engc3 engc4 engc5)
{<o> (auto  ↑mod order  ↑serial <n>)}
- (automac  ↑serial <n>  ↑machine engine)
- (automac  ↑mod busy  ↑machine engine)
(clock  ↑mod going  ↑timer <t>)
{<s> (steel  ↑what scrap  ↑amount {<q> (> 2)})}
rhs:   (enga1 enga2)
(make automac  ↑mod busy  ↑serial <n>  ↑machine engine  ↑clock (compute "<t>" + 7))
(modify <s>  ↑amount (compute "<q>" - 3))

Debug p interrupt; type ok to continue:
[13h:39:25 Elapsed 0:31, Process 0:24.5 (GC 0%), Fired 0 (avg  msec)]
↑9 ok

[13h:39:38 Elapsed 0:13, Process 0:00.0 (GC 0%), Fired 0 (avg  msec)]
 sc:end sc:endc1 sc:endc1t sc:endc2 sc:endc2t sc:endc3 sc:endc3t sc:enda1 sc:enda2
Comp-vars: vars in lhs and rhs: (<n> <o> <t>)

[p end
```

```
  { (auto ↑ mod order ↑ serial <n>) <o> }
    (automac ↑ mod done ↑ serial <n> ↑ machine assembly)
    (clock ↑ mod going ↑ timer <t>)
--> (modify <o> ↑ mod new ↑ clock <t>)
    (write New auto <n> at time <t>) ]

lhs:   (endc1 endc2 endc3)
{<o> (auto  ↑mod order  ↑serial <n>)}
(automac  ↑mod done  ↑serial <n>  ↑machine assembly)
(clock  ↑mod going  ↑timer <t>)
rhs:   (enda1 enda2)
(modify <o>  ↑mod new  ↑clock <t>)
(write   ↑rhs-terms (New auto "<n>" at time "<t>"))

Debug p interrupt; type ok to continue:
[13h:40:01 Elapsed 0:23, Process 0:16.9 (GC 0%), Fired 0 (avg  msec)]
↑10 ok

[13h:40:04 Elapsed 0:03, Process 0:00.1 (GC 0%), Fired 0 (avg  msec)]

After loading, there are 5 rules, 70 OPS8 schemata.
{{ autosel
    instance: "production-system"
    rule-set: "buy2" "beg" "clu" "eng" "end"}}

[Storage: List 31759, symbol 4444, string 23, binary 2066, array 13, value 128, hunk nil]

[13h:40:05 Elapsed 0:01, Process 0:00.1 (GC 0%), Fired 0 (avg  msec)]

Do you want a db-update of the rule database now? (y n):y
 . . . result of db-update = t

Loaded
```

That concludes the translation of the autosel rules. The schema "autosel" now contains a set of the rule names. The following schout command (an SRL function) creates a disk file with all the schemata displayed. (Users with on-line access can look at the mentioned file.)

```
38.(schout (db-get-schemata 'autosel) "$root-context" 'autosel.schout)

t
39.ps autosel
{{ autosel
    instance: "production-system"
    rule-set: "buy2" "beg" "clu" "eng" "end"}}

nil
40.pm "buy2"

lhs:   (buy2c1 buy2c2)
{<g> (buy  ↑dollars {<a> (> 0)})}
{<s> (steel  ↑what scrap  ↑amount <a2>)}
rhs:   (buy2a1 buy2a2 buy2a3)
(modify <g>  ↑dollars (compute 0 - "<a>"))
(modify <s>  ↑amount (compute "<a>" + "<a2>"))
(make clock  ↑mod going  ↑timer 0  ↑maker 0)

("buy2")
41.wm

(dummy  ↑att1 val4  ↑att2 val3)
(dummy2  ↑att1 val7  ↑att2 val9)
("g00008" "g00010")
```

The following shows the user entering the user interface and typing a command, which the interface translates into a pair of make commands.. After doing the make, a run command is automatically done by the interface. In the present context, the user has not specified a control function (which would be in the control slot of the "autosel" schema), so run prompts the user for the name of a rule to attempt to match.

```
42.↑

[13h:43:02 Elapsed 2:57, Process 0:34.5 (GC 9%), Fired 0 (avg  msec)]
↑11 buy ↑dollars 20 & steel ↑what scrap ↑amount 3

(make buy ↑ dollars 20) sc:g00012
(make steel ↑ what scrap ↑ amount 3) sc:g00013
[13h:44:04 Elapsed 1:02, Process 0:01.4 (GC 0%), Fired 0 (avg  msec)]

{{ autosel
    instance: "production-system"
    rule-set: "buy2" "beg" "clu" "eng" "end"}}

Run: enter rule schema name to fire, or nil:"buy2"

Attempting to match buy2
LHS of buy2 = (buy2c1 buy2c2)

Matching buy2c1t in rule buy2
Matching to template buy2c1t ( psrl-instance buy )

Candidate schemas to be matched: (g00012)
  Checking slots: (dollars)
  Checking schema: g00012
    Binding (<g> <a>) to (g00012 20)

Matching buy2c2t in rule buy2
Matching to template buy2c2t ( psrl-instance steel )

Candidate schemas to be matched: (g00013 engc5t begc3t)

Candidates narrowed down to: (g00013)
  Checking slots: (what amount)
  Checking schema: g00013
    Binding (<s> <a2>) to (g00013 3)

New conflict (buy2 3 20 g00012 g00013)
Update-conflicts adding: (buy2 3 20 g00012 g00013)

Result of PSRL match = $psrl-conflicts:

Conflict set, with dominant entry first:
("buy2" 3 20 "g00012" "g00013")
```

Since the match has succeeded, run proceeds to execute the actions of the rule.

```
1.  ("buy2" 3 20 "g00012" "g00013")
--> modify
Modify:
(buy  ↑dollars 20)
(modify <g>  ↑dollars (compute 0 - "<a>"))
(buy  ↑dollars -20) g00012(buy) modify
Modify:
(steel  ↑what scrap  ↑amount 3)
(modify <s>  ↑amount (compute "<a>" + "<a2>"))
(steel  ↑what scrap  ↑amount 23) g00013(steel) make sc:g00014
Make:
(clock  ↑mod going  ↑timer 0  ↑maker 0) g00014(clock)
```

```
Run: enter rule schema name to fire, or nil:nil

[13h:45:02 Elapsed 0:58, Process 0:25.6 (GC 26%), Fired 1 (avg 25583 msec)]
↑12 wm

(dummy  ↑att1 val4  ↑att2 val3)
(dummy2  ↑att1 val7  ↑att2 val9)
(buy  ↑dollars -20)
(steel  ↑what scrap  ↑amount 23)
(clock  ↑mod going  ↑timer 0  ↑maker 0)
("g00008" "g00010" "g00012" "g00013" "g00014")
```

The above WM display shows the new schemata created by the rule. Next, we find another rule to execute.

```
↑13 pm "beg"

lhs:   (begc1 begc2 begc3)
(clock  ↑mod going  ↑timer <t>)
- (automac  ↑mod busy  ↑machine engine)
(steel  ↑what scrap  ↑amount  (> 4))
rhs:   (bega1 bega2)
(bind   ↑variable <n>  ↑rhs-terms nil)
(make auto  ↑mod order  ↑serial <n>)

("beg")
↑14 fire "beg"

Attempting to match beg
LHS of beg = (begc1 begc2 begc3)

Matching begc1t in rule beg
Matching to template begc1t ( psrl-instance clock )

Candidate schemas to be matched: (g00014 endc3t engc4t cluc1t buy2a3)

Candidates narrowed down to: (g00014)
  Checking slots: (mod timer)
  Checking schema: g00014
    Binding (<t>) to (0)

Matching begc2t in rule beg
Matching to template begc2t ( psrl-instance automac )

Candidate schemas to be matched: (endc2t enga1 engc3t engc2t)
    No schemas, negated, continuing.

Matching begc3t in rule beg
Matching to template begc3t ( psrl-instance steel )

Candidate schemas to be matched: (g00013 engc5t buy2c2t)

Candidates narrowed down to: (g00013)
  Checking slots: (what amount)
  Checking schema: g00013    Succeeded.

New conflict (beg $unbound$ 0)
Update-conflicts adding: (beg $unbound$ 0)

Result of PSRL match = $psrl-conflicts:

Conflict set, with dominant entry first:
("beg" $unbound$ 0)
2.  ("beg" $unbound$ 0)
--> bind make sc:g00016
```

```
Make:
(auto  ↑mod order  ↑serial g00015) g00016(auto)
"beg"
[13h:45:56 Elapsed 0:54, Process 0:23.0 (GC 0%), Fired 1 (avg 23016 msec)]
↑15 wm

(dummy  ↑att1 val4  ↑att2 val3)
(dummy2  ↑att1 val7  ↑att2 val9)
(buy  ↑dollars -20)
(steel  ↑what scrap  ↑amount 23)
(clock  ↑mod going  ↑timer 0  ↑maker 0)
(auto  ↑mod order  ↑serial "g00015")
("g00008" "g00010" "g00012" "g00013" "g00014" "g00016")
↑16 pp beg

[p beg
    (clock ↑ mod going ↑ timer <t>)
  - (automac ↑ mod busy ↑ machine engine)
    (steel ↑ what scrap ↑ amount > 4)
--> (bind <n>)
    (make auto ↑ mod order ↑ serial <n>) ]

t
```

The next segment shows how a rule can be created similar to an existing rule, using the internal Lisp editor to modify a copy of the existing rule. The first part of the segment is an error - the newp command requires another argument.

```
↑17 newp like beg
edit
#pp


(beg)

#p

(beg)
#ok
 sc:like sc:likec1 sc:likec1t

*** SRL ERROR REPORT ***
*** TYPE:           no-slot
*** SOURCE:         valuec1
*** IN SCHEMA:      likec1tNo catch for this tag   error

[p like
    beg ]

lhs:   (likec1)
((production ((clock ↑ mod going ↑ timer <t>) - (automac ↑ mod busy ↑ machine
engine) (steel ↑ what scrap ↑ amount > 4) --> (bind <n>) (make auto ↑ mod
order ↑ serial <n>))))
rhs:   nil

Debug p interrupt; type ok to continue:
[13h:47:06 Elapsed 1:10, Process 0:03.0 (GC 0%), Fired 0 (avg  msec)]
↑18 newp beg2 like beg
edit
#pp


((clock ↑ mod going ↑ timer <t>)
 (- (automac ↑ mod busy ↑ machine engine))
 (steel ↑ what scrap ↑ amount > 4)
```

```
 -->
 (bind <n>)
 (make auto ↑ mod order ↑ serial <n>))

#sw 2 3

((clock ↑ mod going ↑ timer <t>) (steel ↑ what scrap ↑ amount > 4) (- & ) -->
(bind <n>) (make auto ↑ mod order ↑ serial <n>))
#ok
 sc:beg2 sc:beg2c1 sc:beg2c1t sc:beg2c2 sc:beg2c2t sc:beg2c3 sc:beg2c3t
 sc:beg2a1 sc:beg2a2
Comp-vars: vars in lhs and rhs: (<n> <t>)

[p beg2
    (clock ↑ mod going ↑ timer <t>)
    (steel ↑ what scrap ↑ amount > 4)
  - (automac ↑ mod busy ↑ machine engine)
--> (bind <n>)
    (make auto ↑ mod order ↑ serial <n>) ]

lhs:   (beg2c1 beg2c2 beg2c3)
(clock  ↑mod going  ↑timer <t>)
(steel  ↑what scrap  ↑amount  (> 4))
- (automac  ↑mod busy  ↑machine engine)
rhs:   (beg2a1 beg2a2)
(bind   ↑variable <n>  ↑rhs-terms nil)
(make auto  ↑mod order  ↑serial <n>)

Debug p interrupt; type ok to continue:
[13h:48:17 Elapsed 1:11, Process 0:20.0 (GC 18%), Fired 0 (avg  msec)]
↑19 ok

[13h:48:20 Elapsed 0:03, Process 0:00.0 (GC 0%), Fired 0 (avg  msec)]

beg2
```

The next thing to do is to save the new rule on a file, for later potential use. (There will be a little bit of junk saved, due to the newp mistake; it would be easily edited out later.)

```
↑20 changes

<no-file>      (beg2 like)
nil
↑21 prop like

like
production
          (beg)

t
↑22 chkpt

(beg2 like)
chkpt.0
↑23 nil
↑24 ("Emergency exit")
[13h:49:17 Elapsed 0:57, Process 0:00.4 (GC 0%), Fired 0 (avg  msec)]

43.
Do you really want to exit? (y/n) Bye
L
script done on Sun Jan  8 13:49:20 1984
```

Contents of chkpt.0, which is the file written out by the chkpt function above:

```
[p beg2
    (clock ↑ mod going ↑ timer <t>)
    (steel ↑ what scrap ↑ amount > 4)
  - (automac ↑ mod busy ↑ machine engine)
--> (bind <n>)
    (make auto ↑ mod order ↑ serial <n>) ]

[p like
    beg ]
```

7.4. Selected Auto Factory Rules

This appendix goes through a sample run of a simple rule set that is part of an OPS5 automobile factory simulation. The OPS5 program is itself a simple version, for illustrative purposes. The rules selected here show only part of the basic execution mechanism of the system. First, there is a display of the file of rules.

```
;                       SELECTIONS FROM

;               Automobile Factory Mini-Simulation

;                           June 1983

;                       FOR DEBUGGING OPS8

[p buy2 {<g> (buy ↑dollars {<a> > 0})}
    {<s> (steel ↑what scrap ↑amount <a2>)}
--> (modify <g> ↑dollars (compute 0 - <a>))
    (modify <s> ↑amount (compute <a> + <a2>))
    (make clock ↑mod going ↑timer 0 ↑maker 0) ]

[p beg (clock ↑mod going ↑timer <t>)
    - (automac ↑mod busy ↑machine engine)
    (steel ↑what scrap ↑amount > 4)
--> (bind <n>) (make auto ↑mod order ↑serial <n>) ]

[p clu (clock ↑mod stopping) (auto ↑mod order ↑serial <o>)
--> (write Unfinished <o> (crlf)) ]

[p eng
    {<o> (auto ↑mod order ↑serial <n>)}
    - (automac ↑serial <n> ↑machine engine)
    - (automac ↑mod busy ↑machine engine)
    (clock ↑mod going ↑timer <t>)
    {<s> (steel ↑what scrap ↑amount {<q> > 2})}
--> (make automac ↑mod busy ↑serial <n> ↑machine engine
        ↑clock (compute <t> + 7))
    (modify <s> ↑amount (compute <q> - 3)) ]

[p end {<o> (auto ↑mod order ↑serial <n>)}
    (automac ↑mod done ↑serial <n> ↑machine assembly)
    (clock ↑mod going ↑timer <t>)
--> (modify <o> ↑mod new ↑clock <t>)
    (write New auto <n> at time <t>) ]
```

First, the PSRL system is started up and tlps is used to load and translate the rules (see the preceding appendix for a more complete explanation of this segment). Within the translation, PSRL prints an 'l' when a literalize statement is read, and a 'p' when a rule is translated. The meanings of the rules will be explained as they are executed.

```
Script started on Sun Jan  8 14:14:51 1984
L
L pwd
/usr10/mdr/psrl/test
L
L islisp5 -t
[Initializing, please wait]

ISLisp 5.25, 12-18-83, from Franz Lisp, Opus 38

   Copyright (c) 1983 CMU Robotics Institute

1.load psrl.init
```

```
Loading initf and uin8 ... [fasl /usr/mdr/psrl/bin/initf]
t[fasl /usr/mdr/psrl/bin/uin8]
t
Connecting to basicps = basic + psrl database: t

Loading psrl functions ... [fasl /usr/mdr/psrl/bin/ops8.o]
t[fasl /usr/mdr/psrl/bin/help8.o]
t[fasl /usr/mdr/psrl/bin/compl8.o]
t[fasl /usr/mdr/psrl/bin/match8.o]
t[fasl /usr/mdr/psrl/bin/act8.o]
t
Initializing PSRL-OPS8 ...

PSRL Version 1.2 - 7 Jan. 1984

Type ? for help; type ↑ for Ops8 user interface.
Done
2.↑

[14h:16:04 Elapsed 0, Process 0, Fired 0 (avg  msec)]
↑1 tlps autosel

[Storage: List 10556, symbol 4263, string 5, binary 2066, array 13, value 128, hunk nil]

[14h:16:10 Elapsed 0:06, Process 0:00.4 (GC 0%), Fired 0 (avg  msec)]

Before loading, there are 0 rules, 0 OPS8 schemata.

Do you want to save these on a database after translation? (y n):n
PS:"autosel"
Literal file: autosel.litlllll
Rule file: autosel.op5ppppp
After loading, there are 5 rules, 60 OPS8 schemata.
{{ autosel
    instance: "production-system"
    rule-set: "buy2" "beg" "clu" "eng" "end"}}

[Storage: List 21368, symbol 4353, string 9, binary 2066, array 13, value 128, hunk nil]

[14h:17:30 Elapsed 1:20, Process 0:26.5 (GC 10%), Fired 0 (avg  msec)]

Loaded
↑2 watch 3

3
```

The automobile simulation rules are divided into several categories: initialization, clock management (the clock drives the simulation), and execution of machines in the factory. The selected rules in this section are representative of each category. There is one initialization rule, buy2, which takes an element representing a user wish to buy more materials and starts the clock running. There are two clock rules, beg and clu, which, respectively, start up automobile orders and recognize unfinished autos when materials are exhausted. Finally, there are two machine rules, one for making an engine (named eng) and the other (named end) for announcing the completion of an order. The simulation segment presented here starts with the initialization rule buy2. Its first condition, a buy element, comes from the user. The second condition, a steel element, also comes from the user in this script, but in the full simulation it is automatically initialized by another rule. The buy2 rule has three actions, one to change the buy element to show it has been processed, the second to increase the amount of steel, and the third to make a clock element that will drive a segment of simulated factory operation.

The display of the buy2 rule below is one that is generated from schemata, as opposed to using the external text format. Thus it includes some schema names for conditiona and action elements, and has an internal format within some expressions.

```
↑3 pm buy2

lhs:   (buy2c1 buy2c2)
{<g> (buy  ↑dollars {<a> (> 0)})}
{<s> (steel  ↑what scrap  ↑amount <a2>)}
rhs:   (buy2a1 buy2a2 buy2a3)
(modify <g>  ↑dollars (compute 0 - "<a>"))
(modify <s>  ↑amount (compute "<a>" + "<a2>"))
(make clock  ↑mod going  ↑timer 0  ↑maker 0)

(buy2)
```

The following user input is converted into two make statements by the user interface, UIN8. The make statements are echoed, and then a line of statistics information is printed, before PSRL starts running - the run is also started automatically by UIN8 when it reads a line that translates into a make.

```
↑4 buy ↑dollars 20 & steel ↑what scrap ↑amount 3

(make buy ↑ dollars 20)
(make steel ↑ what scrap ↑ amount 3)
[14h:20:05 Elapsed 2:35, Process 0:20.9 (GC 27%), Fired 0 (avg  msec)]

{{ autosel
    instance: "production-system"
    rule-set: "buy2" "beg" "clu" "eng" "end"}}
```

The "autosel" production system has no control slot, so PSRL prompts the user for a rule to match and possibly execute. The matching process succeeds in this case, indicated by the 'New conflict' line. The rule execution is shown starting with the '1.' line, which has the name of the rule and the values for its match variables (e.g., <a> is bound to 3, <a2> to 20, etc.) The actions performed are traced after the '-->', e.g., first a modify is done on g00008, whose element class is buy. Then the user is again prompted for a rule name. The user command 'wm' shows the new working memory - three elements, followed by the list that the wm function returns, a list of schema names of the elements just displayed.

```
Run: enter rule schema name to fire, or nil:buy2

Attempting to match buy2
New conflict (buy2 3 20 g00008 g00009)
1.  ("buy2" 3 20 "g00008" "g00009")
--> modify g00008(buy) modify g00009(steel) make g00010(clock)
Run: enter rule schema name to fire, or nil:nil

[14h:20:40 Elapsed 0:35, Process 0:06.5 (GC 0%), Fired 1 (avg 6499 msec)]
↑5 wm

(buy  ↑dollars -20)
(steel  ↑what scrap  ↑amount 23)
(clock  ↑mod going  ↑timer 0  ↑maker 0)
("g00008" "g00009" "g00010")
```

At this point, a more automatic mode of rule execution is set up, using the control function, whose effect on the "autosel" schema is then shown. The rule named 'beg' is going to be next to execute, which is already able to match. (The *** function is used to insert a comment in the script.)

Further simulation is set up by the 'auto' input, which in combination with 'beg' results in two orders being processed at once in the factory. Several rules execute before the next stopping point.

```
↑6 control fire-sequence

fire-sequence
↑7 "autosel"
{{ autosel
    instance: "production-system"
    rule-set: "buy2" "beg" "clu" "eng" "end"
    control: fire-sequence}}

↑8 pm "clu"

lhs:   (cluc1 cluc2)
(clock  ↑mod stopping)
(auto  ↑mod order  ↑serial <o>)
rhs:   (clua1)
(write   ↑rhs-terms (Unfinished "<o>" (crlf)))

("clu")
↑9 pm "beg"

lhs:   (begc1 begc2 begc3)
(clock  ↑mod going  ↑timer <t>)
- (automac  ↑mod busy  ↑machine engine)
(steel  ↑what scrap  ↑amount  (> 4))
rhs:   (bega1 bega2)
(bind   ↑variable <n>  ↑rhs-terms nil)
(make auto  ↑mod order  ↑serial <n>)

("beg")
↑10 *** "the clu display above was a mistake ..."

nil
↑11 auto ↑mod order ↑serial xx84

(make auto ↑ mod order ↑ serial xx84)
[14h:23:15 Elapsed 2:35, Process 0:19.1 (GC 16%), Fired 0 (avg  msec)]

Running autosel under control of fire-sequence, for at most 20 cycles.

Attempting to match buy2
Attempting to match beg
New conflict (beg $unbound$ 0)
2.  ("beg" $unbound$ 0)
--> bind make g00013(auto)
Attempting to match clu
Attempting to match eng
New conflict (eng g00012 g00013 23 g00009 0)
New conflict (eng xx84 g00011 23 g00009 0)
3.  ("eng" xx84 "g00011" 23 "g00009" 0)
--> make g00014(automac) modify g00009(steel)
4.  ("eng" "g00012" "g00013" 23 "g00009" 0)
--> make g00015(automac) modify g00009(steel)
Attempting to match end
Finished autosel under control of fire-sequence, after firing 3 rules
[14h:24:29 Elapsed 1:14, Process 1:02.9 (GC 5%), Fired 3 (avg 20949 msec)]
↑12 wm

(buy  ↑dollars -20)
(steel  ↑what scrap  ↑amount -20)
(clock  ↑mod going  ↑timer 0  ↑maker 0)
(auto  ↑mod order  ↑serial xx84)
(auto  ↑mod order  ↑serial "g00012")
(automac  ↑mod busy  ↑serial xx84  ↑machine engine  ↑clock 7)
```

```
(automac  ↑mod busy  ↑serial "g00012"  ↑machine engine  ↑clock 7)
("g00008" "g00009" "g00010" "g00011" "g00013" "g00014" "g00015")
```

Next, the 'clu' rule is set up, by changing the clock's status (the 'mod' field) to stopping. The system is run again, and 'clu' is the rule selected to be fired. Since there are two unfinished auto orders, the rule fires twice.

```
↑13 pm "clu"

lhs:   (cluc1 cluc2)
(clock  ↑mod stopping)
(auto  ↑mod order  ↑serial <o>)
rhs:   (clua1)
(write   ↑rhs-terms (Unfinished "<o>" (crlf)))

("clu")
↑14 wm "clock"

(clock  ↑mod going  ↑timer 0  ↑maker 0)
("g00010")
↑15 modify g00010 ↑mod stopping

"g00010"
↑16 redo 14

(clock  ↑mod stopping  ↑timer 0  ↑maker 0)
("g00010")
↑17 run

Running autosel under control of fire-sequence, for at most -1 cycles.
        (-1 for unlimited number)

Attempting to match buy2
Attempting to match beg
Attempting to match clu
New conflict (clu g00012)
New conflict (clu xx84)
5.  ("clu" xx84)
--> writeUnfinished xx84

6.  ("clu" "g00012")
--> writeUnfinished g00012

Attempting to match eng
Attempting to match end
Finished autosel under control of fire-sequence, after firing 2 rules
t
[14h:26:57 Elapsed 2:28, Process 0:48.8 (GC 6%), Fired 2 (avg 24383 msec)]
↑18 wm

(buy  ↑dollars -20)
(steel  ↑what scrap  ↑amount 20)
(clock  ↑mod stopping  ↑timer 0  ↑maker 0)
(auto  ↑mod order  ↑serial xx84)
(auto  ↑mod order  ↑serial "g00012")
(automac  ↑mod busy  ↑serial xx84  ↑machine engine  ↑clock 7)
(automac  ↑mod busy  ↑serial "g00012"  ↑machine engine  ↑clock 7)
("g00008" "g00009" "g00010" "g00011" "g00013" "g00014" "g00015")
```

Now the same thing is done again, to illustrate how the system looks with no rule tracing.

```
↑19 watch 0

0
```

```
↑20 run
Unfinished xx84
Unfinished g00012

t
[14h:29:04 Elapsed 2:07, Process 0:43.0 (GC 8%), Fired 2 (avg 21474 msec)]
↑21 ?8v

PSRL-OPS8 and SRL global parameters have these values:
  $db-cache-max = 2400    $db-cache-keep = 2000    $queuelength = 205
  $default-context = $root-context  db-get-all-db = (/usr/mdr/psrl/bin/basicps)
  $demon = nil    $default = nil    $restrict = nil
  $meta-schema = schema    $meta-slot = relation    $meta-value = value
  $psrl-instance = psrl-instance    $psrl-instance+inv = psrl-instance+inv
  $ops8-ps = autosel    $ops8-exts = (op5 lit)    $psrl-watch = 0
  $psrl-silence = nil  $trace-match = nil  $debug-fire = nil  $debug-match = nil
  $psrl-context = nil    $match-inherit = nil    $psrl-inherit-slots = nil
  $psrl-control-step = nil    $demon-run = t
t
↑22 nil
↑23 ("Emergency exit")
[14h:30:30 Elapsed 1:26, Process 0:00.1 (GC 0%), Fired 0 (avg  msec)]

3.
Do you really want to exit? (y/n) Bye
L
script done on Sun Jan  8 14:30:34 1984
```

7.5. Syntax Tests

```
;               Testers of syntax and other odd features (no lit file)

(literalize test)
(literalize class1 tab1 tab2 tab3 tab4)
(literalize class2 tab1 tab2 tab3 tab4)

(control fire-sequence)
(setq $psrl-control-step t)
(watch 4)

[p st0 (test)
--> (make class1 ↑tab1 5 ↑tab2 val1)
    (make class2 ↑tab1 val3 ↑tab2 (eval (quote <a>))) ]

[p st1
  { (class1 ↑tab1 {<a> > 0} ↑tab2 val1 ↑tab3 nil) <g>}
  {<s> (class2 ↑tab1 << val1 val2 val3 >> ↑tab2 << <a> <g> >>)}
--> (make class1 ↑tab1 val3 ↑tab2 17)
    (modify <g> ↑tab2 <s> ↑tab3 92)
    (modify <s> ↑tab1 val1 ↑tab2 <g>)
    (make class2 ↑is-a relation ↑transitivity t ↑tab3 arf) ]

;       st1 tests << >>, schema vars in both places; null value;

[p st2
  { (class1 ↑tab1 > 0 ↑tab2 TYPE "psrl-instance" "class2" ↑tab3 {})
        <g> ! schema-p}
  {<s> <> <g> (class2 ↑tab1 val1 ↑tab2 ! schema-p)}
  { (class2) <> <s>}
--> (make class1 ↑tab1 105 ↑tab2 val7) ]

;       st2 tests some odd range restrictions; {} slot value

[p st3
    ({} ↑is-a relation ↑transitivity <> nil)
  { (class2 ↑tab1 <> nil) <> "myname" }
--> (make "myname" class2 ↑tab1 57) ]

;       st3 tests {} class, new make, restr on schema var without var binding

(setq $psrl-inherit-slots t)

[p st5 (class1 ↑tab1 <x> ↑tab9 <z>)
--> (!write this is impossible) ]

(setq $psrl-inherit-slots nil)

;       st5 tests undeclared slot, also illegal action

[p st99
  { (class2 ↑tab1 <a>) = "myname" }
  { (class1 ↑tab2 <s>) }
  { <s> }
--> (write <a>) (halt) ]

;       st99 tests another restr on schema var; null template;
;       bare schvar test (internally, only schema-p on it); also halt

(msg (N 1) "Start syntax tests running by making 'test'" (N 1))


Script started on Mon Jan  2 14:19:25 1984
L
```

```
L : "fix to match-schema for null template; msg in editr; st99 fixed"
L
L islisp5 -t
[Initializing, please wait]

ISLisp 5.25, 12-18-83, from Franz Lisp, Opus 38

   Copyright (c) 1983 CMU Robotics Institute

1.load exppsrl.init

PSRL-OPS8 Experimental Version 1.2

Loading initf and uin8 ... [fasl /usr/mdr/psrl/src/initf]
[fasl /usr/mdr/psrl/src/uin8]

Connecting to basicps = basic + psrl database: t

Doing (dskin ops8 help8 compl8 match8 act8) ...(/usr/mdr/psrl/src/ops8)
(/usr/mdr/psrl/src/help8)(/usr/mdr/psrl/src/compl8)(/usr/mdr/psrl/src/match8)
(/usr/mdr/psrl/src/act8)
Initializing ops8 ...

PSRL Version 1.2 - 31 Dec. 1983

Type ? for help; type ↑ for Ops8 user interface.
Done
2.↑

[14h:21:14 Elapsed 0, Process 0, Fired 0 (avg  msec)]
↑1 changes

nil
↑2 load 'syntax.op8
111PS:"generic-psrl-ps" sc:st0 sc:st0c1 sc:st0c1t sc:st0a1 sc:st0a2 sc:<a>
Comp-vars: vars in lhs and rhs: nil

[p st0
    (test)
--> (make class1 ↑ tab1 5 ↑ tab2 val1)
    (make class2 ↑ tab1 val3 ↑ tab2 (eval '<a>)) ]

lhs:   (st0c1)
(test)
rhs:   (st0a1 st0a2)
(make class1  ↑tab1 5  ↑tab2 val1)
(make class2  ↑tab1 val3  ↑tab2 (eval '"<a>"))

Debug p interrupt; type ok to continue:
[14h:21:54 Elapsed 0:40, Process 0:11.1 (GC 29%), Fired 0 (avg  msec)]
↑3 ok

[14h:23:04 Elapsed 1:10, Process 0:00.1 (GC 0%), Fired 0 (avg  msec)]
 sc:st1 sc:st1c1 sc:st1c1t sc:<g> sc:st1c2 sc:st1c2t sc:<s> sc:st1a1 sc:st1a2 sc:st1a3 sc:st1a4
Comp-vars: vars in lhs and rhs: (<a> <g> <s>)

[p st1
  { (class1 ↑ tab1 { <a> > 0 } ↑ tab2 val1 ↑ tab3 nil) <g> }
  { (class2 ↑ tab1 << val1 val2 val3 >> ↑ tab2 << <a> <g> >>) <s> }
--> (make class1 ↑ tab1 val3 ↑ tab2 17)
    (modify <g> ↑ tab2 <s> ↑ tab3 92)
    (modify <s> ↑ tab1 val1 ↑ tab2 <g>)
    (make class2 ↑ is-a relation ↑ transitivity t ↑ tab3 arf) ]

lhs:   (st1c1 st1c2)
{<g> (class1  ↑tab1 {<a> (> 0)}  ↑tab2 val1  ↑tab3 nil)}
```

```
{<s> (class2  ↑tab1  (OR val1 val2 val3)  ↑tab2  (OR (QUOTE <a>) (QUOTE <g>)))}
rhs:   (st1a1 st1a2 st1a3 st1a4)
(make class1  ↑tab1 val3  ↑tab2 17)
(modify <g>  ↑tab2 <s>  ↑tab3 92)
(modify <s>  ↑tab1 val1  ↑tab2 <g>)
(make class2  ↑is-a relation  ↑transitivity t  ↑tab3 arf)

Debug p interrupt; type ok to continue:
[14h:23:24 Elapsed 0:20, Process 0:13.1 (GC 0%), Fired 0 (avg  msec)]
↑4 ok

[14h:23:29 Elapsed 0:05, Process 0:00.1 (GC 0%), Fired 0 (avg  msec)]
 sc:st2 sc:st2c1 sc:st2c1t sc:st2c2 sc:st2c2t sc:st2c3 sc:st2c3t sc:st2a1
Comp-vars: vars in lhs and rhs: (<g> <s>)

[p st2
  { (class1 ↑ tab1 > 0 ↑ tab2 TYPE "psrl-instance" "class2" ↑ tab3 { }) <g> !
    schema-p }
  { (class2 ↑ tab1 val1 ↑ tab2 ! schema-p) <s> <> <g> }
  { (class2) <> <s> }
--> (make class1 ↑ tab1 105 ↑ tab2 val7) ]

lhs:   (st2c1 st2c2 st2c3)
{<g> (FUNCTION schema-p) (class1  ↑tab1  (> 0)  ↑tab2  (TYPE psrl-instance class2)  ↑tab3 {})}
{<s> (<> <g>) (class2  ↑tab1 val1  ↑tab2  (FUNCTION schema-p))}
{{} (<> <s>) (class2)}
rhs:   (st2a1)
(make class1  ↑tab1 105  ↑tab2 val7)

Debug p interrupt; type ok to continue:
[14h:23:42 Elapsed 0:13, Process 0:09.2 (GC 0%), Fired 0 (avg  msec)]
↑5 ok

[14h:23:57 Elapsed 0:15, Process 0:00.0 (GC 0%), Fired 0 (avg  msec)]
 sc:st3 sc:st3c1 sc:st3c1t sc:st3c2 sc:st3c2t sc:st3a1
Comp-vars: vars in lhs and rhs: nil

[p st3
    ({ } ↑ is-a relation ↑ transitivity <> nil)
  { (class2 ↑ tab1 <> nil) <> "myname" }
--> (make "myname" class2 ↑ tab1 57) ]

lhs:   (st3c1 st3c2)
({}  ↑is-a relation  ↑transitivity  (<> nil))
{{} (<> myname) (class2  ↑tab1  (<> nil))}
rhs:   (st3a1)
(make class2  ↑psrl-make-name myname  ↑tab1 57)

Debug p interrupt; type ok to continue:
[14h:24:07 Elapsed 0:10, Process 0:06.5 (GC 0%), Fired 0 (avg  msec)]
↑6 ok

[14h:24:10 Elapsed 0:03, Process 0:00.1 (GC 0%), Fired 0 (avg  msec)]
 sc:st5 sc:st5c1 sc:st5c1t sc:<x> sc:<z>

*** SRL ERROR REPORT ***
*** TYPE:           no-slot
*** SOURCE:         valuec1
*** IN SCHEMA:      st5c1t
*** IN SLOT:        tab9
*** WITH VALUE:     ({"<z>" nil})No catch for this tag   error

[p st5
    (class1 ↑ tab1 <x> ↑ tab9 <z>)
--> (!write this is impossible) ]
```

```
lhs:   (st5c1)
(class1  ↑tab1 <x>)
rhs:   nil

Debug p interrupt; type ok to continue:
[14h:24:17 Elapsed 0:07, Process 0:03.9 (GC 0%), Fired 0 (avg  msec)]
↑7 ok

[14h:24:52 Elapsed 0:35, Process 0:00.0 (GC 0%), Fired 0 (avg  msec)]
 sc:st99 sc:st99c1 sc:st99c1t sc:st99c2 sc:st99c2t sc:st99c3 sc:st99a1 sc:st99a2
Comp-vars: vars in lhs and rhs: (<a> <s>)

[p st99
  { (class2 ↑ tab1 <a>) = "myname" }
  { (class1 ↑ tab2 <s>)
  { nil <s> }
--> (write <a>)
    (halt) ]

lhs:   (st99c1 st99c2 st99c3)
{{} (= myname) (class2  ↑tab1 <a>)}
{{} (class1  ↑tab2 <s>)}
{<s> ()}
rhs:   (st99a1 st99a2)
(write   ↑rhs-terms ("<a>"))
(halt )

Debug p interrupt; type ok to continue:
[14h:25:04 Elapsed 0:12, Process 0:08.4 (GC 0%), Fired 0 (avg  msec)]
↑8 ok

[14h:25:29 Elapsed 0:25, Process 0:00.0 (GC 0%), Fired 0 (avg  msec)]

Start syntax tests running by making 'test'

t
↑9 changes

nil
↑10 ed st5

st5
edit
#p

(err-production (&  --> & ))
#(1 production)

(production (&  --> & ))
#ok

(st5)
↑11 ed st5

st5

(editr) edit
#p

((class1 ↑ tab1 <x> ↑ tab9 <z>) --> (!write this is impossible))
#r tab9 tab3

((class1 ↑ tab1 <x> ↑ tab3 <z>) --> (!write this is impossible))
#ok
(excising st5)
{{ st5
```

```
    instance: "rule"
    psrl-lhs: "st5c1"
    psrl-rhs:}}

Destroyed old schema (above) st5
 sc:st5
{{ st5c1
    instance: "lhs-element"
    template: "st5c1t"}}

Destroyed old schema (above) st5c1
 sc:st5c1
{{ st5c1t
    template+inv:
    psrl-instance: "class1"
        [instance: "psrl-instance"]
        [domain: "st5c1t"]
        [slot: "psrl-instance"]
    tab1: "<x>"
        [instance: "relation"]
        [included-from: "54+118780492114020084"]
        [domain: "st5c1t"]
        [slot: "tab1"]}}

Destroyed old schema (above) st5c1t
 sc:st5c1t sc:st5a1
Warning: undefined action !write in st5a1

Comp-vars: vars in lhs and rhs: (<x> <z>)

[p st5
    (class1 ↑ tab1 <x> ↑ tab3 <z>)
--> (!write this is impossible) ]

lhs:   (st5c1)
(class1  ↑tab1 <x>  ↑tab3 <z>)
rhs:   (st5a1)
(!write   ↑rhs-terms (this is impossible))

Debug p interrupt; type ok to continue:
[14h:26:55 Elapsed 1:26, Process 0:14.5 (GC 59%), Fired 0 (avg  msec)]
↑12 ok

[14h:28:49 Elapsed 1:54, Process 0:00.0 (GC 0%), Fired 0 (avg  msec)]

st5 now in %changes
(st5)
↑13 changes

<no-file>      (st5)
nil
↑14 ed st99

st99

(editr) edit
#pp


(({ (class2 ↑ tab1 <a>) = "myname" })
 ({ (class1 ↑ tab2 <s>) })
 ({ <s> })
 -->
 (write <a>)
 (halt))
```

```
#sw -1 -2

(({ &  = "myname" }) ({ &  }) ({ <s> }) --> (halt) (write <a>))
#ok
(excising st99)
{{ st99
    instance: "rule"
    psrl-lhs: "st99c1" "st99c2" "st99c3"
    psrl-rhs: "st99a1" "st99a2"
    psrl-variables: "<a>" "<s>"}}

Destroyed old schema (above) st99
 sc:st99
{{ st99c1
    instance: "lhs-element"
    template: "st99c1t"
    schema-variable: {}
        [instance: "relation"]
        [domain: "st99c1"]
        [slot: "schema-variable"]
        [range: (= "myname")]}}

Destroyed old schema (above) st99c1
 sc:st99c1
{{ st99c1t
    template+inv:
    psrl-instance: "class2"
        [instance: "psrl-instance"]
        [domain: "st99c1t"]
        [slot: "psrl-instance"]
    tab1: "<a>"
        [instance: "relation"]
        [domain: "st99c1t"]
        [slot: "tab1"]}}

Destroyed old schema (above) st99c1t
 sc:st99c1t
{{ st99c2
    instance: "lhs-element"
    template: "st99c2t"
    schema-variable: {}
        [instance: "relation"]
        [domain: "st99c2"]
        [slot: "schema-variable"]}}

Destroyed old schema (above) st99c2
 sc:st99c2
{{ st99c2t
    template+inv:
    psrl-instance: "class1"
        [instance: "psrl-instance"]
        [domain: "st99c2t"]
        [slot: "psrl-instance"]
    tab2: "<s>"
        [instance: "relation"]
        [domain: "st99c2t"]
        [slot: "tab2"]}}

Destroyed old schema (above) st99c2t
 sc:st99c2t
{{ st99c3
    instance: "lhs-element"
    template:
    schema-variable: "<s>"
        [instance: "relation"]
        [domain: "st99c3"]
```

```
        [slot: "schema-variable"]}}

Destroyed old schema (above) st99c3
 sc:st99c3
{{ st99a1
    psrl-action: "write"
    rhs-terms: ("<a>")
        [instance: "relation"]
        [domain: "st99a1"]
        [slot: "rhs-terms"]}}

Destroyed old schema (above) st99a1
 sc:st99a1
{{ st99a2
    psrl-action: "halt"}}

Destroyed old schema (above) st99a2
 sc:st99a2
Comp-vars: vars in lhs and rhs: (<a> <s>)

[p st99
  { (class2 ↑ tab1 <a>) = "myname" }
  { (class1 ↑ tab2 <s>)
  { nil <s> }
--> (halt)
    (write <a>) ]

lhs:   (st99c1 st99c2 st99c3)
{{} (= myname) (class2  ↑tab1 <a>)}
{{} (class1  ↑tab2 <s>)}
{<s> ()}
rhs:   (st99a1 st99a2)
(halt )
(write   ↑rhs-terms ("<a>"))

Debug p interrupt; type ok to continue:
[14h:30:20 Elapsed 1:31, Process 0:15.5 (GC 27%), Fired 0 (avg  msec)]
↑15 ok

[14h:30:23 Elapsed 0:03, Process 0:00.1 (GC 0%), Fired 0 (avg  msec)]

st99 now in %changes
(st99)
↑16 changes

<no-file>      (st99 st5)
nil
↑17 ed st99

st99

(editr) edit
#-2

(halt)
#bk up

... --> (halt) (write <a>))
#sw 2 3

... --> (write <a>) (halt))
#2

(write <a>)
#(-2 And the answer is:)
```

```
(write And the answer is: <a>)
#ok
(excising st99)
{{ st99
    instance: "rule"
    psrl-lhs: "st99c1" "st99c2" "st99c3"
    psrl-rhs: "st99a1" "st99a2"
    psrl-variables: "<a>" "<s>"}}

Destroyed old schema (above) st99
 sc:st99
{{ st99c1
    instance: "lhs-element"
    template: "st99c1t"
    schema-variable: {}
        [instance: "relation"]
        [domain: "st99c1"]
        [slot: "schema-variable"]
        [range: (= "myname")]}}

Destroyed old schema (above) st99c1
 sc:st99c1
{{ st99c1t
    template+inv:
    psrl-instance: "class2"
        [instance: "psrl-instance"]
        [domain: "st99c1t"]
        [slot: "psrl-instance"]
    tab1: "<a>"
        [instance: "relation"]
        [domain: "st99c1t"]
        [slot: "tab1"]}}

Destroyed old schema (above) st99c1t
 sc:st99c1t
{{ st99c2
    instance: "lhs-element"
    template: "st99c2t"
    schema-variable: {}
        [instance: "relation"]
        [domain: "st99c2"]
        [slot: "schema-variable"]}}

Destroyed old schema (above) st99c2
 sc:st99c2
{{ st99c2t
    template+inv:
    psrl-instance: "class1"
        [instance: "psrl-instance"]
        [domain: "st99c2t"]
        [slot: "psrl-instance"]
    tab2: "<s>"
        [instance: "relation"]
        [domain: "st99c2t"]
        [slot: "tab2"]}}

Destroyed old schema (above) st99c2t
 sc:st99c2t
{{ st99c3
    instance: "lhs-element"
    template:
    schema-variable: "<s>"
        [instance: "relation"]
        [domain: "st99c3"]
        [slot: "schema-variable"]}}
```

```
Destroyed old schema (above) st99c3
 sc:st99c3
{{ st99a1
    psrl-action: "halt"}}

Destroyed old schema (above) st99a1
 sc:st99a1
{{ st99a2
    psrl-action: "write"
    rhs-terms: ("<a>")
        [instance: "relation"]
        [domain: "st99a2"]
        [slot: "rhs-terms"]}}

Destroyed old schema (above) st99a2
 sc:st99a2
Comp-vars: vars in lhs and rhs: (<a> <s>)

[p st99
  { (class2 ↑ tab1 <a>) = "myname" }
  { (class1 ↑ tab2 <s>)
  { nil <s> }
--> (write And the answer is: <a>)
    (halt) ]

lhs:   (st99c1 st99c2 st99c3)
{{} (= myname) (class2  ↑tab1 <a>)}
{{} (class1  ↑tab2 <s>)}
{<s> ()}
rhs:   (st99a1 st99a2)
(write   ↑rhs-terms (And the answer is: "<a>"))
(halt )

Debug p interrupt; type ok to continue:
[14h:31:46 Elapsed 1:23, Process 0:11.2 (GC 0%), Fired 0 (avg  msec)]
↑18 ok

[14h:32:05 Elapsed 0:19, Process 0:00.0 (GC 0%), Fired 0 (avg  msec)]

st99 now in %changes
(st99)
↑19 changes

<no-file>      (st99 st5)
nil
↑20 chkpt

(st99 st5)
chkpt.0
↑21 test

(make test) sc:g00041
[14h:33:41 Elapsed 1:36, Process 0:00.7 (GC 0%), Fired 0 (avg  msec)]

Running generic-psrl-ps under control of fire-sequence, for at most 20 cycles.

Attempting to match st0
LHS of st0 = (st0c1)

Comp-vars: vars in lhs and rhs: nil

Matching st0c1t in rule st0
Matching to template st0c1t ( psrl-instance test )

Candidate schemas to be matched: (g00041)
  Checking slots: nil
```

```
  Checking schema: g00041    Succeeded.

New conflict (st0)
Update-conflicts adding: (st0)

Result of PSRL match = $psrl-conflicts:

Conflict set, with dominant entry first:
("st0")
1.  ("st0")
--> make sc:g00043
Make:
(class1  ↑tab1 5  ↑tab2 val1) g00043(class1) make sc:g00044
Make:
(class2  ↑tab1 val3  ↑tab2 <a>) g00044(class2)
Control step interrupt; resume by typing ok.
[14h:33:55 Elapsed 0:14, Process 0:08.4 (GC 0%), Fired 1 (avg 8399 msec)]
↑22 ok

[14h:33:59 Elapsed 0:04, Process 0:00.0 (GC 0%), Fired 0 (avg  msec)]

Attempting to match st1
LHS of st1 = (st1c1 st1c2)

Matching st1c1t in rule st1
Matching to template st1c1t ( psrl-instance class1 )

Candidate schemas to be matched: (g00043 st99c2t st5c1t st2a1 st2c1t st1a1 st0a1)

Candidates narrowed down to: (g00043)
  Checking slots: (tab1 tab2 tab3)
  Checking schema: g00043
    Binding (<g> <a>) to (g00043 5)

Matching st1c2t in rule st1
Matching to template st1c2t ( psrl-instance class2 )

Candidate schemas to be matched: (g00044 st99c1t st3a1 st3c2t st2c3t st2c2t st1a4 st0a2)

Candidates narrowed down to: (g00044)
  Checking slots: (tab1 tab2)
  Checking schema: g00044
    Binding (<s>) to (g00044)

New conflict (st1 5 g00043 g00044)
Update-conflicts adding: (st1 5 g00043 g00044)

Result of PSRL match = $psrl-conflicts:

Conflict set, with dominant entry first:
("st1" 5 "g00043" "g00044")
2.  ("st1" 5 "g00043" "g00044")
--> make sc:g00047
Make:
(class1  ↑tab1 val3  ↑tab2 17) g00047(class1) modify
Modify:
(class1  ↑tab1 5  ↑tab2 val1)
(modify <g>  ↑tab2 <s>  ↑tab3 92)
(class1  ↑tab1 5  ↑tab2 g00044  ↑tab3 92) g00043(class1) modify
Modify:
(class2  ↑tab1 val3  ↑tab2 <a>)
(modify <s>  ↑tab1 val1  ↑tab2 <g>)
(class2  ↑tab1 val1  ↑tab2 g00043) g00044(class2) make sc:g00048
Make:
(class2  ↑is-a relation  ↑transitivity t  ↑tab3 arf) g00048(class2)
Control step interrupt; resume by typing ok.
```

```
[14h:34:29 Elapsed 0:30, Process 0:18.6 (GC 21%), Fired 1 (avg 18583 msec)]
↑23 wm

(test)
(class1  ↑tab1 5  ↑tab2 "g00044"  ↑tab3 92)
(class2  ↑tab1 val1  ↑tab2 "g00043")
(class1  ↑tab1 val3  ↑tab2 17)
(class2  ↑is-a relation  ↑transitivity t  ↑tab3 arf)
("g00041" "g00043" "g00044" "g00047" "g00048")
↑24 ok

[14h:34:52 Elapsed 0:23, Process 0:00.5 (GC 0%), Fired 0 (avg  msec)]

Attempting to match st2
LHS of st2 = (st2c1 st2c2 st2c3)

Matching st2c1t in rule st2
Matching to template st2c1t ( psrl-instance class1 )

Candidate schemas to be matched: (g00047 g00043 st99c2t st5c1t st2a1 st1a1 st1c1t st0a1)

Candidates narrowed down to: (g00047 g00043)
  Checking slots: (tab1 tab2 tab3)
  Checking schema: g00047 (fail tab1 range)    Failed.
  Checking schema: g00043
    Binding (<g>) to (g00043)

Matching st2c2t in rule st2
Matching to template st2c2t ( psrl-instance class2 )

Candidate schemas to be matched: (g00048 g00044 st99c1t st3a1 st3c2t st2c3t
  st1a4 st1c2t st0a2)

Candidates narrowed down to: (g00048 g00044)
  Checking slots: (tab1 tab2)
  Checking schema: g00048 (fail tab1)    Failed.
  Checking schema: g00044
    Binding (<s>) to (g00044)

Matching st2c3t in rule st2
Matching to template st2c3t ( psrl-instance class2 )

Candidate schemas to be matched: (g00048 g00044 st99c1t st3a1 st3c2t st2c2t
  st1a4 st1c2t st0a2)

Candidates narrowed down to: (g00048)
  Checking slots: nil
  Checking schema: g00048    Succeeded.

New conflict (st2 g00043 g00044)
Update-conflicts adding: (st2 g00043 g00044)

Result of PSRL match = $psrl-conflicts:

Conflict set, with dominant entry first:
("st2" "g00043" "g00044")
3.  ("st2" "g00043" "g00044")
--> make sc:g00053
Make:
(class1  ↑tab1 105  ↑tab2 val7) g00053(class1)
Control step interrupt; resume by typing ok.
[14h:35:06 Elapsed 0:14, Process 0:07.5 (GC 0%), Fired 1 (avg 7533 msec)]
↑25 wm

(test)
(class1  ↑tab1 5  ↑tab2 "g00044"  ↑tab3 92)
```

```
(class2  ↑tab1 val1  ↑tab2 "g00043")
(class1  ↑tab1 val3  ↑tab2 17)
(class2  ↑is-a relation  ↑transitivity t  ↑tab3 arf)
(class1  ↑tab1 105  ↑tab2 val7)
("g00041" "g00043" "g00044" "g00047" "g00048" "g00053")
↑26 ok

[14h:40:02 Elapsed 4:56, Process 0:00.7 (GC 0%), Fired 0 (avg  msec)]

Attempting to match st3
LHS of st3 = (st3c1 st3c2)

Comp-vars: vars in lhs and rhs: nil

Matching st3c1t in rule st3
Matching to template st3c1t ( is-a relation )

Candidate schemas to be matched: (g00048 st1a4 psrl-action+inv psrl-action
    template+inv template psrl-instance+inv psrl-instance sub-module
    has-sub-command sub-command-of mapped-to mapped-from elaborated-to
    elaborated-from included-to included-from instance+inv is-a+inv instance is-a
    relation)

Candidates narrowed down to: (g00048 psrl-action+inv psrl-action template+inv
    template psrl-instance+inv psrl-instance sub-module has-sub-command
    sub-command-of mapped-to mapped-from elaborated-to elaborated-from included-to
    included-from instance+inv is-a+inv instance is-a relation)
  Checking slots: (transitivity)
  Checking schema: g00048    Succeeded.

Matching st3c2t in rule st3
Matching to template st3c2t ( psrl-instance class2 )

Candidate schemas to be matched: (g00048 g00044 st99c1t st3a1 st2c3t st2c2t st1a4 st1c2t st0a2)

Candidates narrowed down to: (g00048 g00044)
  Checking slots: (tab1)
  Checking schema: g00048 (fail tab1)    Failed.
  Checking schema: g00044    Succeeded.

New conflict (st3)
Update-conflicts adding: (st3)
  Checking schema: psrl-action+inv (fail transitivity)    Failed.
  Checking schema: psrl-action (fail rel psrl-instance)    Failed.
  Checking schema: template+inv (fail transitivity)    Failed.
  Checking schema: template (fail rel psrl-instance)    Failed.
  Checking schema: psrl-instance+inv (fail transitivity)    Failed.
  Checking schema: psrl-instance (fail rel psrl-instance)    Failed.
  Checking schema: sub-module (fail transitivity)    Failed.
  Checking schema: has-sub-command (fail transitivity)    Failed.
  Checking schema: sub-command-of (fail transitivity)    Failed.
  Checking schema: mapped-to (fail transitivity)    Failed.
  Checking schema: mapped-from (fail transitivity)    Failed.
  Checking schema: elaborated-to (fail transitivity)    Failed.
  Checking schema: elaborated-from (fail transitivity)    Failed.
  Checking schema: included-to (fail transitivity)    Failed.
  Checking schema: included-from (fail transitivity)    Failed.
  Checking schema: instance+inv (fail transitivity)    Failed.
  Checking schema: is-a+inv (fail transitivity)    Failed.
  Checking schema: instance (fail rel psrl-instance)    Failed.
  Checking schema: is-a (fail rel psrl-instance)    Failed.
  Checking schema: relation (fail transitivity range)    Failed.

Result of PSRL match = $psrl-conflicts:

Conflict set, with dominant entry first:
```

```
("st3")
4.  ("st3")
--> make sc:myname
Make:
(class2  ↑tab1 57) myname(class2)
Control step interrupt; resume by typing ok.
[14h:40:57 Elapsed 0:55, Process 0:13.2 (GC 0%), Fired 1 (avg 13216 msec)]
↑27 wm

(test)
(class1  ↑tab1 5  ↑tab2 "g00044"  ↑tab3 92)
(class2  ↑tab1 val1  ↑tab2 "g00043")
(class1  ↑tab1 val3  ↑tab2 17)
(class2  ↑is-a relation  ↑transitivity t  ↑tab3 arf)
(class1  ↑tab1 105  ↑tab2 val7)
(class2  ↑tab1 57)
("g00041" "g00043" "g00044" "g00047" "g00048" "g00053" "myname")
↑28 ok

[14h:41:26 Elapsed 0:29, Process 0:00.7 (GC 0%), Fired 0 (avg  msec)]

Attempting to match st5
LHS of st5 = (st5c1)

Matching st5c1t in rule st5
Matching to template st5c1t ( psrl-instance class1 )

Candidate schemas to be matched: (g00053 g00047 g00043 st99c2t st2a1 st2c1t st1a1 st1c1t st0a1)

Candidates narrowed down to: (g00053 g00047 g00043)
  Checking slots: (tab1 tab3)
  Checking schema: g00053
    Binding (<z> <x>) to (nil 105)

New conflict (st5 105 nil)
Update-conflicts adding: (st5 105 nil)
  Checking schema: g00047
    Binding (<z> <x>) to (nil val3)

New conflict (st5 val3 nil)
Update-conflicts adding: (st5 val3 nil)
  Checking schema: g00043
    Binding (<z> <x>) to (92 5)

New conflict (st5 5 92)
Update-conflicts adding: (st5 5 92)

Result of PSRL match = $psrl-conflicts:

Conflict set, with dominant entry first:
("st5" 5 92)
("st5" val3 nil)
("st5" 105 nil)
5.  ("st5" 5 92)
--> !write
Warning: undefined RHS action, !write in st5a1 ... ignoring it.
Control step interrupt; resume by typing ok.
[14h:41:36 Elapsed 0:10, Process 0:06.6 (GC 0%), Fired 1 (avg 6583 msec)]
↑29 pp st5

[p st5
    (class1 ↑ tab1 <x> ↑ tab3 <z>)
--> (!write this is impossible) ]

t
↑30 ok
```

```
[14h:42:30 Elapsed 0:54, Process 0:00.1 (GC 0%), Fired 0 (avg  msec)]

6.  ("st5" val3 nil)
--> !write
Warning: undefined RHS action, !write in st5a1 ... ignoring it.
Control step interrupt; resume by typing ok.
[14h:42:31 Elapsed 0:01, Process 0:00.3 (GC 0%), Fired 1 (avg 283 msec)]
↑31 ok

[14h:42:36 Elapsed 0:05, Process 0:00.1 (GC 0%), Fired 0 (avg  msec)]

7.  ("st5" 105 nil)
--> !write
Warning: undefined RHS action, !write in st5a1 ... ignoring it.
Control step interrupt; resume by typing ok.
[14h:42:37 Elapsed 0:01, Process 0:00.3 (GC 0%), Fired 1 (avg 249 msec)]
↑32 ok

[14h:42:43 Elapsed 0:06, Process 0:00.1 (GC 0%), Fired 0 (avg  msec)]

Attempting to match st99
LHS of st99 = (st99c1 st99c2 st99c3)

Matching st99c1t in rule st99
Matching to template st99c1t ( psrl-instance class2 )

Candidate schemas to be matched: (myname g00048 g00044 st3a1 st3c2t st2c3t
    st2c2t st1a4 st1c2t st0a2)

Candidates narrowed down to: (myname)
  Checking slots: (tab1)
  Checking schema: myname
    Binding (<a>) to (57)

Matching st99c2t in rule st99
Matching to template st99c2t ( psrl-instance class1 )

Candidate schemas to be matched: (g00053 g00047 g00043 st5c1t st2a1 st2c1t st1a1 st1c1t st0a1)

Candidates narrowed down to: (g00053 g00047 g00043)
  Checking slots: (tab2)
  Checking schema: g00053
    Binding (<s>) to (val7)

Matching nil in rule st99
Matching to template nil ( no relation in template )

Candidate schemas to be matched: (val7)
    Failed - no schemas.
  Checking schema: g00047
    Binding (<s>) to (17)

Matching nil in rule st99
Matching to template nil ( no relation in template )

Candidate schemas to be matched: (17)
    Failed - no schemas.
  Checking schema: g00043
    Binding (<s>) to (g00044)

Matching nil in rule st99
Matching to template nil ( no relation in template )

Candidate schemas to be matched: (g00044)
  Checking slots: nil
```

```
  Checking schema: g00044
    Binding (<s>) to (g00044)

New conflict (st99 57 g00044)
Update-conflicts adding: (st99 57 g00044)

Result of PSRL match = $psrl-conflicts:

Conflict set, with dominant entry first:
("st99" 57 "g00044")
8.  ("st99" 57 "g00044")
--> writeAnd the answer is: 57  halt
Halt has been executed.

Finished generic-psrl-ps under control of fire-sequence, after firing 8 rules
[14h:42:57 Elapsed 0:14, Process 0:09.6 (GC 0%), Fired 1 (avg 9633 msec)]
↑33 "<x>"
{{ <x>
    psrl-instance: "match-variable"
    value: 105}}
↑34 "<z>"
{{ <z>
    psrl-instance: "match-variable"
    value: nil}}
↑35 bind <x> 42

{42 nil}
↑36 "<x>"
{{ <x>
    psrl-instance: "match-variable"
    value: 42}}
↑37 cbind <z>

{"myname" nil}
↑38 "<z>"
{{ <z>
    psrl-instance: "match-variable"
    value: "myname"}}
↑39 backtrace

Elements are of the form (rule-name variable-bindings), oldest first:
("st0")
("st1" 5 "g00043" "g00044")
("st2" "g00043" "g00044")
("st3")
("st5" 5 92)
("st5" val3 nil)
("st5" 105 nil)
("st99" 57 "g00044")
t
↑40 wm

(test)
(class1  ↑tab1 5  ↑tab2 "g00044"  ↑tab3 92)
(class2  ↑tab1 val1  ↑tab2 "g00043")
(class1  ↑tab1 val3  ↑tab2 17)
(class2  ↑is-a relation  ↑transitivity t  ↑tab3 arf)
(class1  ↑tab1 105  ↑tab2 val7)
(class2  ↑tab1 57)
("g00041" "g00043" "g00044" "g00047" "g00048" "g00053" "myname")
↑41 pp st99

[p st99
  { (class2 ↑ tab1 <a>) = "myname" }
  { (class1 ↑ tab2 <s>)
  { nil <s> }
```

```
--> (write And the answer is: <a>)
    (halt) ]

t
↑42 ?b

Please include the following in bug reports, to ensure a prompt reply:
  The name of your script file (include complete directory and machine)
  A segment from your script file, including the bug and the following:
    If inside a Lisp debug loop (prompts ':'), do the commands: pp, bkev
    pm and pp of rules involved
    ps of rule template(s) involved
    ps or wm of schemas involved (from wm or elsewhere in database)
    ?8v, to show global parameters' values
nil
↑43 ?8v

OPS8 and SRL global parameters have these values:
  $db-cache-max = 200    $db-cache-keep = 100    $queuelength = 177
  $default-context = $root-context  db-get-all-db = (/usr/mdr/psrl/src/basicps)
  $demon = nil    $default = nil    $restrict = nil
  $meta-schema = schema    $meta-slot = relation    $meta-value = value
  $psrl-instance = psrl-instance    $psrl-instance+inv = psrl-instance+inv
  $ops8-ps = generic-psrl-ps    $ops8-exts = (op5 lit)    $psrl-watch = 4
  $psrl-silence = nil  $trace-match = t  $debug-fire = t  $debug-match = t
  $psrl-context = nil    $match-inherit = nil    $psrl-inherit-slots = nil
  $psrl-control-step = t    $demon-run = t
nil
↑44 psrl-version


PSRL Version 1.2 - 31 Dec. 1983

nil
↑45 nil
↑46 ("Emergency exit")
[14h:53:47 Elapsed 10:50, Process 0:01.9 (GC 0%), Fired 0 (avg  msec)]

3.
Do you really want to exit? (y/n) Bye
L
script done on Mon Jan  2 14:53:55 1984
```

References

[1] Bobrow, D. and Stefik, M.
The LOOPS Manual (Preliminary Version).
Technical Report, XEROX Palo Alto Research Center, Palo Alto, CA, 1983.

[2] Brownston, L., Farrell, R., Kant, E., Martin, N.
Production Systems: A Tutorial Introduction.
Forthcoming, 1984.

[3] Charniak, E., Riesbeck, C. K., McDermott, D. V.
Artificial Intelligence Programming.
Lawrence Erlbaum Associates, Hillsdale, NJ, 1980.

[4] Fain, J., Gorlin, D., Hayes-Roth, F., Rosenschein, S. J., Sowizral, H., and Waterman, D.
The ROSIE Language Reference Manual.
Technical Report N-1646-ARPA, The Rand Corp., Santa Monica, CA, 1981.

[5] Forgy, C. and McDermott, J.
OPS, a domain-independent production system language.
In *Proc. Fifth International Joint Conference on Artificial Intelligence*, pages 933-939. Cambridge, MA, 1977.

[6] Forgy, C. L.
OPS5 User's Manual.
Technical Report CMU-CS-81-135, Carnegie-Mellon University, Dept. of Computer Science, july, 1981.

[7] Fox, M. S., Greenberg, M., Sathi, A., Mattis, J., and Rychener, M. D.
Callisto: An Intelligent Project Management System.
Technical Report, Carnegie-Mellon University, Robotics Institute, 1984.
Forthcoming.

[8] Fox, M. S.
On Inheritance in Knowledge Representation.
In *Proc. Sixth International Joint Conference on Artificial Intelligence*, pages 282-284. Tokyo, Japan, 1979.

[9] Langley, P. and Neches, R.
PRISM User's Manual.
Technical Report, Carnegie-Mellon University, Dept. of Psychology, July, 1981.

[10] Maher, M. L.
HI-RISE: A Knowledge-Based Expert System for the Preliminary Structural Design of High Rise Buildings.
PhD thesis, Carnegie-Mellon University, Dept. of Civil Engineering, 1984.

[11] Maher, M. L. and Fenves, S. J.
HI-RISE: an expert system for the preliminary structural design of high rise buildings.
In *Submitted to IFIP WG5.2 Working Conference on Knowledge Engineering in Computer-Aided Design, Sept. 11-14.* Budapest, Hungary, 1984.

[12] Mattis, J.
Autograph - A System for Knowledge-Based Graphics.
Technical Report, Carnegie-Mellon University, Robotics Institute, 1984.
Forthcoming.

[13] Rychener, M. D.
Production systems as a programming language for artificial intelligence applications.
PhD thesis, Carnegie-Mellon University, 1976.
University Microfilms International order no. 77-13,843.

[14] Rychener, M. D.
Control requirements for the design of production system architectures.
SIGART Newsletter 64:37-44, August, 1977.
ACM.

[15] Waterman, D. A. and Hayes-Roth, F., Eds.
Pattern-Directed Inference Systems.
Academic, New York, NY, 1978.

[16] Wright, J. M., Fox, M. S., and Adam, D.
SRL/1.5 Users Manual.
Technical Report, Carnegie-Mellon University, Robotics Institute, 1984.
See also Intelligent Systems Lab Software Manual.

Index