The Soar User's Manual

Version 8.2

Edition 1

John E. Laird, Clare Bates Congdon and Karen J. Coulter
Electrical Engineering and Computer Science Department
University of Michigan

Draft of: June 23, 1999

Development of earlier versions of this manual were supported under contract N00014-92-K-2015 from the Advanced Systems Technology Office of the Advanced Research Projects Agency and the Naval Research Laboratory, and contract N66001-95-C-6013 from the Advanced Systems Technology Office of the Advanced Research Projects Agency and the Naval Command and Ocean Surveillance Center, RDT&E division.

comment:

Contents

List of Figures

Preface

Soar is an integrated architecture for knowledge-based problem solving, learning, and interaction with external environments. The authors of this manual assume a basic understanding of artificial intelligence, and/or information processing psychology. For further background on Soar, we recommend *The Soar Papers: Readings on Integrated Intelligence*, Rosenbloom, Laird, and Newell (1993), and *Unified Theories of Cognition*, Newell (1990).

This manual is specific to Version 8 of Soar, which is significantly different from previous versions of Soar.

Acknowledgements

Special thanks to Erik Altmann and Robert Doorenbos who helped author previous versions of the manual, which made this version considerably easier to create, and to Gary Pelton and Scott Huffman, who contributed text for previous versions of the manual. We would also like to thank Karl Schwamb for extensive comments and technical advice, and the Soar group for their patient support.

Chapter 1

Introduction

Soar has been developed to be an architecture for constructing general intelligent systems. It has been in use since 1983, and has evolved through many different versions. This manual documents the most current of these: Soar, version 8.0.

Our goals for Soar include that it is to be an architecture that can:

- be used to build systems that work on the full range of tasks expected of an intelligent agent, from highly routine to extremely difficult, open-ended problems;
- represent and use appropriate forms of knowledge, such as procedural, declarative, episodic, and possibly iconic;
- employ the full range of problem solving methods;
- interact with the outside world; and
- learn about all aspects of the tasks and its performance on them.

In other words, our intention is for Soar to support all the capabilities required of a general intelligent agent. Below are the major principles that are the cornerstones of Soar's design:

1. The number of distinct architectural mechanisms should be minimized. In Soar there is a single representation of permanent knowledge (productions), a single representation of temporary knowledge (objects with attributes and values), a single mechanism for generating goals (automatic subgoaling), and a single learning mechanism (chunking).
2. All decisions are made through the combination of relevant knowledge at run-time. In Soar, every decision is based on the current interpretation of sensory data and any relevant knowledge retrieved from permanent memory. Decisions are never precompiled into uninterruptible sequences.

1.1 Using this Manual

We expect that novice Soar users will read the manual in the order it is presented (perhaps skipping Chapter 5):

Chapter 2 and Chapter 3 describe Soar from different perspectives: **Chapter 2** describes the Soar architecture, but avoids issues of syntax, while **Chapter 3** describes the syntax of Soar, including the specific conditions and actions allowed in Soar productions.

Chapter 4 describes chunking, Soar's learning mechanism. Not all users will make use of chunking, but it is important to know that this capability exists.

Chapter 5 briefly describes advanced features of Soar, such as using input and output to interface with a real or simulated world, writing Tcl/Tk routines, and running Soar with multiple agents. Again, not all users will make use of these capabilities, but it is important to know that they exist.

Chapter 6 describes the Soar user interface — how the user interacts with Soar. The user-interface commands are presented in "beginning", "intermediate", and "advanced" functionality to assist new users in identifying those commands that are expected to be most useful to them. The commands in the user interface are listed in alphabetical order on the back cover of the manual for quick reference.

Advanced users will refer most often to Chapter 6, flipping back to Chapters 2 and 3 to answer specific questions.

There are several appendices included with this manual:

Appendix ?? is a glossary of terminology used in this manual.

Appendix ?? contains an example Soar program for a simple version of the blocks world. This blocks-world program is used as an example throughout the manual.

Appendix ?? is an overview of example programs currently available (provided with the Soar distribution) with explanations of how to run them, and pointers to other help sources available for novices.

Appendix ?? describes Soar's default knowledge, which can be used (or not) with any Soar task.

Appendix ?? provides a grammar for Soar productions.

Appendix ?? provides a formal description of how o-support is determined.

Appendix ?? provides a detailed explanation of the preference resolution process.

Additional Back Matter

The appendices are followed by a brief bibliography and an index; the last two pages of this manual contain a summary and index of the user-interface functions for quick reference.

Not Described in This Manual

Some of the more advanced features of Soar are not described in this manual, such as how to interface with a simulator, or how to create Soar applications using multiple interacting agents. Detailed discussion of these topics are in a separate document, *The Soar Advanced Applications Manual*; see Section 1.2 for more information.

1.2 Other Soar Documentation

In addition to this manual, there are other documents that you may want to obtain for more information about different aspects of Soar:

The Soar 8 Tutorial is written for novice Soar users, and guides the reader through several example tasks and exercises.

The Soar Advanced Applications Manual is written for advanced Soar users. This guide describes how to add input and output routines to Soar programs, how to run multiple Soar "agents" from a single Soar process, and how to extend Soar by adding your own user-interface functions, simulators, or graphical user interfaces. It also includes some discussion of the Tcl-Soar Interface, or TSI.

See the Soar Home Page, `http://ai.eecs.umich.edu/soar`, and Section 1.3 for information about obtaining Soar documentation.

Users who need to know more about Tcl should consult a Tcl reference, many of which are listed on the Tcl Web Site, `http://www.scriptics.com`

1.3 Contacting the Soar Group

The Soar project supports several internet mailing lists. These include:

`soar-requests@umich.edu` — For requests for copies of Soar.

`soar-doc@isi.edu` — For requests for documentation (including this manual).

`soar-bugs@umich.edu` — For reporting Soar bugs, and for asking questions about Soar. (Novice-level questions can be sent to this address.)

We also maintain an electronic mailing list for researchers actively involved in Soar research. If you would like to join this group, send email to `soar-requests@umich.edu`.

World-Wide Web sites

There are many institutions throughout the world working on Soar research, and similarly, there are numerous pages on the world-wide web concerning Soar. The primary site is:

`http://ai.eecs.umich.edu/soar`

The page listed above provides links to information about specific Soar research projects and researchers, as well as a FAQ (list of frequently asked questions) about Soar.

The online FAQ will usually contain the most current information on Soar. It is available at:

`http://www.nottingham.ac.uk/pub/soar/nottinghame/soar-faq.html`

For Those Without Internet Access

If you cannot reach us on the internet, please write to us at the following address:

The Soar Group
Artificial Intelligence Laboratory
University of Michigan
1101 Beal Ave.
Ann Arbor, MI 48109-2110
USA

1.4 A Note on Different Platforms and Operating Systems

Soar runs on a wide variety of computers, including Unix (and Linux) machines, PowerPC Macintoshes, and PCs running the Windows (95, 98, NT) operating system.

This manual documents Soar generally, although all references to files and directories use Unix format rather than Macintosh or Windows folders.

1.5 A Note on Soar and Tcl

Soar uses Tcl, which is a simple interpreted shell language, to implement the Soar user interface. Tcl stands for “Tool Command Language”, and is pronounced “tickle”. Tcl was originally written by John Ousterhout. It is a simple scripting language that allows a Soar user to write extensions to Soar without having to recompile Soar. Tcl also allows the addition of Tk (pronounced “tee-KAY”), a toolkit for writing graphical interfaces. More information on Tcl/Tk is available at http://www.scriptics.com.

The addition of Tcl to Soar allows Soar users to add their own routines without the need to recompile all of Soar. Tcl also makes it easier for users to interface Soar with external programs and simulators, and makes it possible for users to write simple simulators and simple programs to monitor problem solving in Soar.

Although from the Soar perspective, we have added Tcl (and Tk) to Soar, technically speaking, we have added Soar to Tcl. This may seem to be a pedantic distinction, but it has implications for all Soar users, and not just those who want to write their own functions. Most significantly, because Soar is added to Tcl (and not vice versa), we are restricted by some Tcl syntax and naming conventions. These will be discussed more thoroughly in Chapter 5.

Chapter 2

The Soar Architecture

This chapter describes the Soar architecture. It covers all aspects of Soar except for the specific syntax of Soar's memories and descriptions of the Soar user-interface commands.

This chapter gives an abstract description of Soar. It starts by giving an overview of Soar and then goes into more detail for each of Soar's main memories (working memory, production memory, and preference memory) and processes (the decision procedure, learning, and input and output).

2.1 An Overview of Soar

The design of Soar is based on the hypothesis that all deliberate *goal*-oriented behavior can be cast as the selection and application of *operators* to a *state*. A state is a representation of the current problem-solving situation; an operator transforms a state (makes changes to the representation); and a goal is a desired outcome of the problem-solving activity.

As Soar runs, it is continually trying to apply the current operator and select the next operator (a state can have only one operator at a time), until the goal has been achieved. The selection and application of operators is illustrated in Figure 2.1.

Soar has separate memories (and different representations) for descriptions of its current situation and its long-term knowledge. In Soar, the current situation, including data from sensors, results of intermediate inferences, active goals, and active operators is held in *working memory.* Working memory is organized as *objects.* Objects are described in terms of their *attributes*; the values of the attributes may correspond to sub-objects, so the description of the state can have a hierarchical organization. (This need not be a strict hierarchy; for example, there's nothing to prevent two objects from being "substructure" of each other.)

The long-term knowledge, which specifies how to respond to different situations in

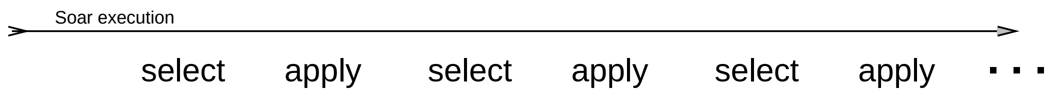

Figure 2.1: Soar is continually trying to select and apply operators.

working memory, can be thought of as the program for Soar. The Soar architecture cannot solve any problems without the addition of long-term knowledge. (Note the distinction between the "Soar architecture" and the "Soar program": The former refers to the system described in this manual, common to all users, and the latter refers to knowledge added to the architecture.)

A Soar program contains the knowledge to be used for solving a specific task (or set of tasks), including information about how to select and apply operators to transform the states of the problem, and a means of recognizing that the goal has been achieved.

2.1.1 Problem-Solving Functions in Soar

All of Soar's long-term knowledge is organized around the functions of operator selection and application. These functions are composed of four distinct types of knowledge:

Knowledge to select an operator

1. *Operator Proposal:* Knowledge that an operator is appropriate for the current situation.
2. *Operator Comparison:* Knowledge to compare candidate operators.
3. *Operator Selection:* Knowledge to select a single operator, based on the comparisons.

Knowledge to apply an operator

4. *Operator Application:* Knowledge of how a specific operator modifies the state.

In addition, there is a fifth type of knowledge in Soar that is indirectly connected to both operator selection and application:

5. Knowledge of monotonic inferences that can be made about the state (*state elaboration*).

State elaborations indirectly affect operator selection and application by creating new descriptions of the current situation that can cue the selection and application of operators.

These problem-solving functions are the primitives for generating behavior in Soar. Four of the functions require retrieving long-term knowledge that is relevant to the current situation: elaborating the state, proposing candidate operators, comparing the candidates, and applying the operator by modifying the state. These functions are driven by the knowledge encoded in a Soar program. Soar represents that knowledge as *production* rules. Production rules are similar to "if-then" statements in conventional programming languages. (For example, a production might say something like "if there are two blocks on the table, then suggest an operator to move one block ontop of the other block"). The "if" part of the production is called its *conditions* and the "then" part of the production is called its *actions*. When the conditions are met in the current situation as defined by working memory, the production is *matched* and it will *fire*, which means that its actions are executed, making changes to working memory. Some productions *retract* their actions when the conditions are no longer met; this will be discussed later.

The other function, selecting the current operator, involves making a decision once sufficient knowledge has been retrieved. This is performed by Soar's *decision procedure*, which is a fixed procedure that interprets *preferences* that have been created by the retrieval functions. The knowledge-retrieval and decision-making functions combine to form Soar's *decision cycle*.

When the knowledge to perform the problem-solving functions is not directly available in productions, Soar is unable to make progress and reaches an *impasse*. There are three types of possible impasses in Soar:

1. An operator cannot be selected because none are proposed.
2. An operator cannot be selected because multiple operators are proposed and the comparisons are insufficient to determine which one should be selected.
3. An operator has been selected, but there is insufficient knowledge to apply it.

In response to an impasse, the Soar architecture creates a *substate* in which operators can be selected and applied to generate or deliberately retrieve the knowledge that was not directly available; the goal in the substate is to resolve the impasse. For example, in a substate, a Soar program may do a lookahead search to compare candidate operators if comparison knowledge is not directly available. Impasses and substates are described in more detail in Section 2.6.

2.1.2 An Example Task: The Blocks-World

We will use a task called the blocks-world as an example throughout this manual. In the blocks-world task, the initial state has three blocks named `A`, `B`, and `C` on a table;

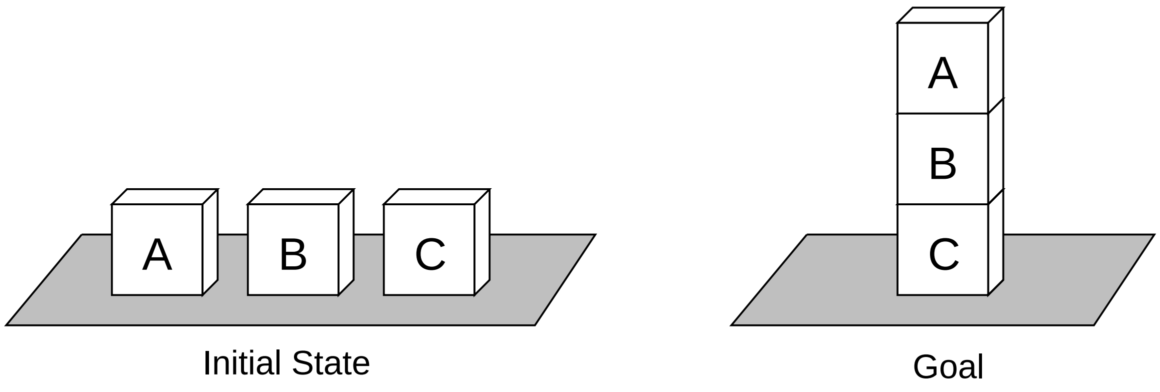

Figure 2.2: The initial state and goal of the “blocks-world” task.

the operators move one block at a time to another location (on top of another block or onto the table); and the goal is to build a tower with `A` on top, `B` in the middle, and `C` on the bottom. The initial state and the goal are illustrated in Figure 2.2.

The Soar code for this task is included in Appendix **??**; it is also included with the Soar release (along with other example programs), in the file `manual-blocks.soar`. You do not need to look at the code at this point.

The operators in this task move a single block from its current location to a new location; each operator is represented with the following information:

- the name of the block being moved
- the current location of the block (the “thing” it is on top of)
- the destination of the block (the “thing” it will be on top of)

The goal in this task is to stack the blocks so that `C` is on the table, with block `B` on block `C`, and block `A` on top of block `B`.

2.1.3 Representation of States, Operators, and Goals

The initial state in our blocks-world task — before any operators have been proposed or selected — is illustrated in Figure 2.3.

A state can have only one operator at a time, and the operator is represented as substructure of the state. A state may also have as substructure a number of *potential* operators that are in consideration; however, these suggested operators should not be confused with the current operator.

Figure 2.4 illustrates working memory after the first operator has been selected. There are six operators proposed, and only one of these is actually selected.

Goals are either represented explicitly as substructure of the state with general rules that recognize when the goal is achieved, or are implicitly represented in the Soar program by goal-specific rules that test the state for specific features and recognize

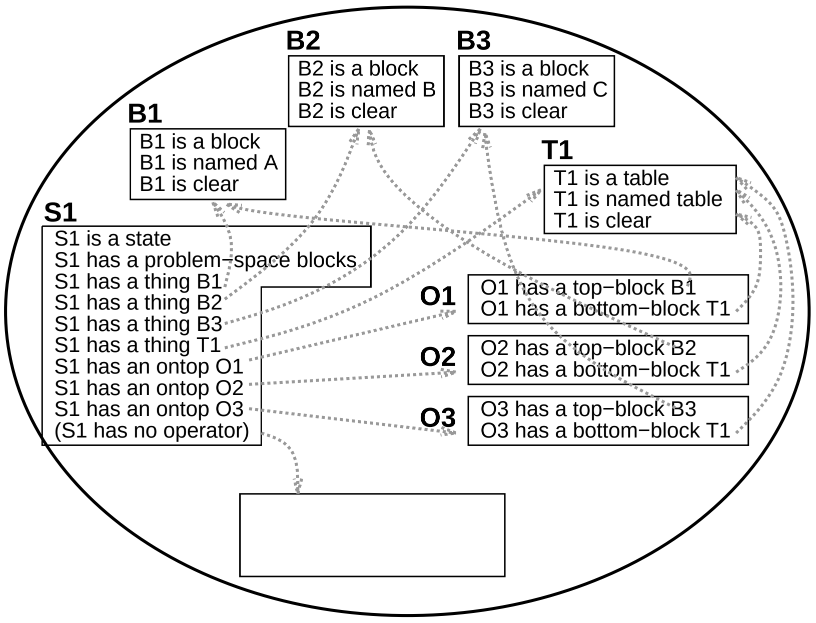

Figure 2.3: An abstract illustration of the initial state of the blocks world as working memory objects. At this stage of problem solving, no operators have been proposed or selected.

---

when the goal is achieved. The point is that sometimes a description of the goal will be available in the state for focusing the problem solving, whereas other times it may not. Although representing a goal explicitly has many advantages, some goals are difficult to explicitly represent on the state.

The goal in our blocks-world task is represented implicitly in the Soar program. A single production rule monitors the state for completion of the goal and halts Soar when the goal is achieved.

2.1.4 Proposing candidate operators

As a first step in selecting an operator, one or more candidate operators are proposed. Operators are proposed by rules that test features of the current state. When the blocks-world task is run, the Soar program will propose six different (but similar) operators for the initial state as illustrated in Figure 2.5. These operators correspond to the six different actions that are possible given the initial state.

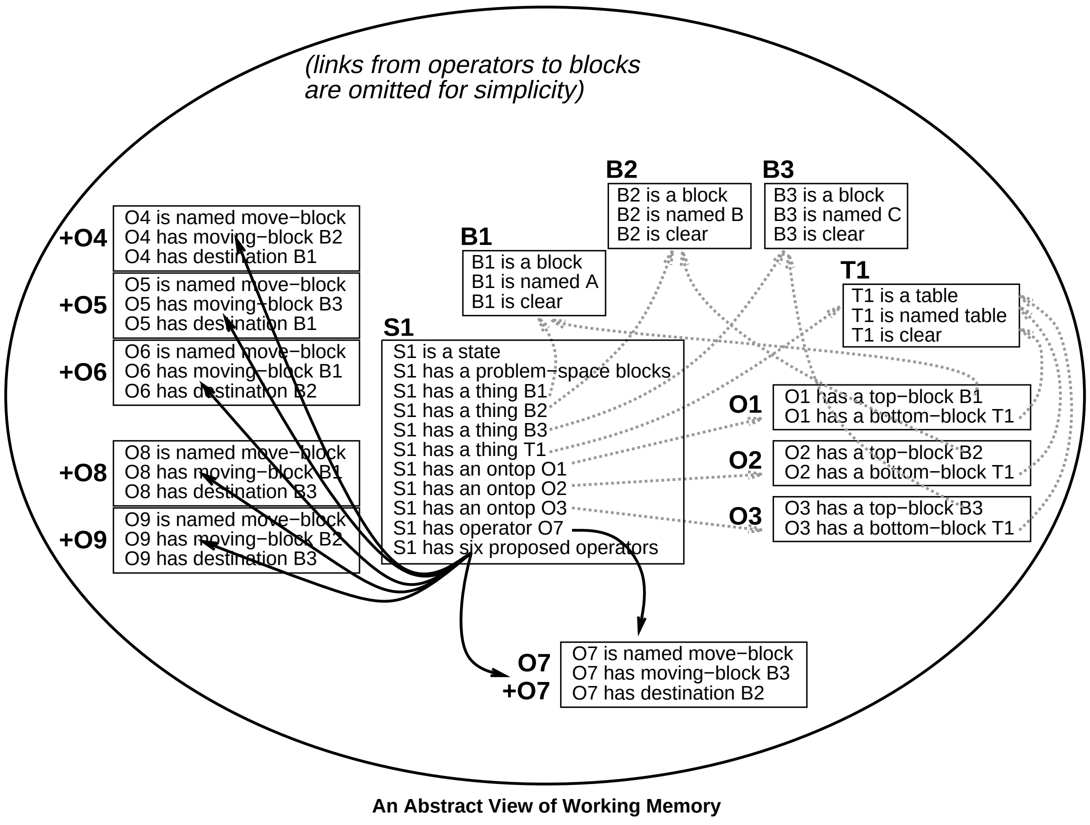

Figure 2.4: An abstract illustration of working memory in the blocks world after the first operator has been selected.

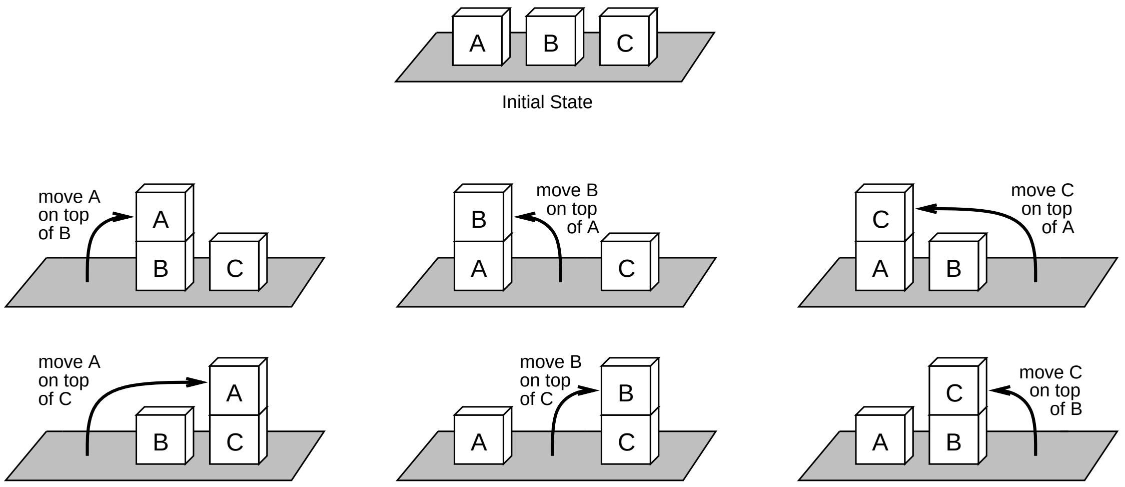

Figure 2.5: The six operators proposed for the initial state of the blocks world each move one block to a new location.

2.1.5 Comparing candidate operators: Preferences

The second step Soar takes in selecting an operator is to evaluate or compare the candidate operators. In Soar, this is done via rules that test the proposed operators, the current state, and then create *preferences*. Preferences assert the relative or absolute merits of the candidate operators. For example, a preference may say that operator A is a "better" choice than operator B at this particular time, or a preference may say that operator A is the "best" thing to do at this particular time.

2.1.6 Selecting a single operator

Soar attempts to select a single operator based on the preferences available for the candidate operators. There are four different situations that may arise:

1. The available preferences unambiguously prefer a single operator.
2. The available preferences suggest multiple operators, and prefer a subset that can be selected from randomly.
3. The available preferences suggest multiple operators,but neither case 1 or 2 above hold.
4. The available preferences do not suggest any operators.

In the first case, the preferred operator is selected. In the second case, one of the subset is selected randomly. In the third and fourth cases, Soar has reached an "impasse" in problem solving, and a new substate is created. Impasses are discussed in Section 2.6.

In our blocks-world example, the second case holds, and Soar can select one of the operators at random.

2.1.7 Applying the operator

An operator applies by making changes to the state; the specific changes that are appropriate depend on the operator and the current state.

There are two primary approaches to modifying the state: indirect and direct. *Indirect* changes are used in Soar programs that interact with an external environment: The Soar program sends motor commands to the external environment and monitors the external environment for changes. The changes are reflected in an updated state description, garnered from sensors. Soar may also make *direct* changes to the state; these correspond to Soar doing problem solving "in its head". Soar programs that do not interact with an external environment can make only direct changes to the state.

Internal and external problem solving should not be viewed as mutually exclusive activities in Soar. Soar programs that interact with an external environment will generally have operators that make direct and indirect changes to the state: The motor command is represented as substructure of the state. Also, a Soar program may maintain an internal model of how it expects an external operator will modify the world; if so, the operator must update the internal model (which is substructure of the state).

When Soar is doing internal problem solving, it must know how to modify the state descriptions appropriately when an operator is being applied. If it is solving the problem in an external environment, it must know what possible motor commands it can issue in order to affect its environment.

The example blocks-world task shown here does not interact with an external environment. Therefore, the Soar program directly makes changes to the state when operators are applied. There are four changes that may need to be made when a block is moved in our task:

1. The block that is being moved is no longer where it was (it is no longer "on top" of the same thing).
2. The block that is being moved is now in a new location (it is "on top" of a new thing).
3. The place that the block used to be is now clear.
4. The place that the block is moving to is no longer clear — unless it is the table, which is always considered "clear"[1].

The blocks-world task could also be implemented using an external simulator (as is done in the examples in *The Soar Coloring Book*). In this case, the Soar program does not update all the "on top" and "clear" relations; the updated state description comes from the simulator.

2.1.8 Making inferences about the state

Making monotonic inferences about the state is the other role that Soar long-term knowledge may fulfill. Such elaboration knowledge can simplify the encoding of operators because entailments of a set of core features of a state do not have to be explicitly included in application of the operator. In Soar, these inferences will be automatically retracted when the situation changes, such as through operator applications or changes in sensory data.

Our example blocks-world task does not use elaborations. However, keeping track of whether a block is "clear" or not could be done with elaborations instead of operator applications. If it were implemented this way, an elaborations would test for the absence of a block that is "on top" of a particular block; if there is no such "on top", the block is "clear".

[1]In this blocks-world task, the table always has room for another block, so it is represented as always being "clear".

2.1.9 Problem Spaces

If we were to construct a Soar system that worked on a large number of different types of problems, we would need to include large numbers of operators in our Soar program. For a specific problem and a particular stage in problem solving, only a subset of all possible operators are actually relevant. For example, if our goal is to *count* the blocks on the table, operators having to do with moving blocks are probably not important, although they may still be "legal". The operators that are relevant to current problem-solving activity define the space of possible states that might be considered in solving a problem, that is, they define the *problem space*.

Soar programs are implicitly organized in terms of problem spaces because the conditions for proposing operators will restrict an operator to be considered only when it is relevant. The complete problem space for the blocks world is show in Figure 2.6. Typically, when Soar solves a problem in this problem space, it does not explicitly generate all of the states, examine them, and then create a path. Instead, Soar is *in* a specific state at a given time (represented in working memory), attempting to select an operator that will move it to a new state. It uses whatever knowledge it has about selecting operators given the current situation, and if its knowledge is sufficient, it will move toward its goal. The same problem could be recast in Soar as a planning problem, where the goal is to develop a plan to solve the problem, instead of just solving the problem. In that case, a state in Soar would consist of a plan, which in turn would have representations of Blocks World states and operators from the original space. The operators would perform editing operations on the plan, such as adding new Blocks World operators, simulating those operators, etc. In both formulations of the problem, Soar is still applying operators to generate new states, it is just that the states and operators have different content.

The following sections describe the memories and processes of Soar: working memory, production memory, preference memory, the decision procedure, learning, and input and output.

2.2 Working memory: The Current Situation

Soar represents the current problem-solving situation in its *working memory*. Thus, working memory holds the current state and operator (as well as any substates and operators generated because of impasses) and is Soar's "short-term" knowledge, reflecting the current knowledge of the world and the status in problem solving.

Working memory contains elements called working memory elements, or WME's for short. Each WME contains a very small piece of information; for example, a WME might say that "B1 is a block". Several WME's collectively may provide more information about the same *object*, for example, "B1 is a block", "B1 is named A", "B1 is on the table", etc. These WME's are related because they are all contributing to the description of something that is internally known to Soar as "B1". B1 is called an

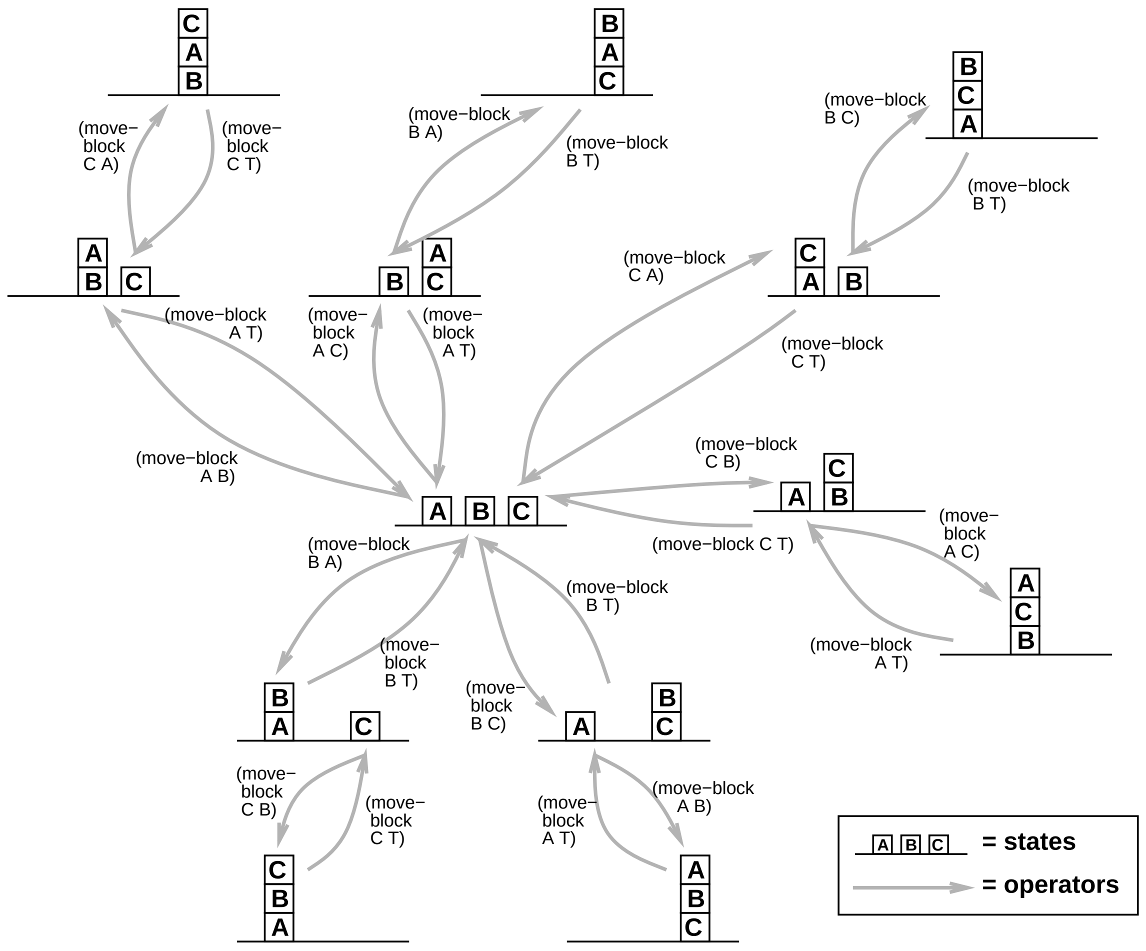

Figure 2.6: The problem space in the blocks-world includes all operators that move blocks from one location to another and all possible configurations of the three blocks.

identifier; the group of WME's that share this identifier are called an *object* in working memory. Each WME describes a different *attribute* of the object, for example, its name or type or location; each attribute has a *value* associated with it, for example, the name is A, the type is block, and the position is on the table. Therefore, each WME is an identifier-attribute-value triple, and all WME's with the same identifier are part of the same object.

Objects in working memory are *linked* to other objects: The value of one WME may be an identifier of another object. For example, a WME might say that "B1 is ontop of T1", and another collection of WME's might describe the object T1: "T1 is a table", "T1 is brown", and "T1 is ontop of F1". And still another collection of WME's might describe the object F1: "F1 is a floor", etc. All objects in working memory must be linked to a state, either directly or indirectly (through other objects). Objects that are not linked to a state will be automatically removed from working memory by the Soar architecture.

WME's are also often called *augmentations* because they "augment" the object, providing more detail about it. While these two terms are somewhat redundant, WME is a term that is used more often to refer to the contents of working memory, while augmentation is a term that is used more often to refer to the description of an object. Working memory is illustrated at an abstract level in Figure 2.3 on page 11.

The attribute of an augmentation is usually a constant, such as `name` or `type`, because in a sense, the attribute is just a label used to distinguish one link in working memory from another.[2]

The value of an augmentation may be either a constant, such as `red`, or an identifier, such as `O6`. When the value is an identifier, it refers to an object in working memory that may have additional substructure. In semantic net terms, if a value is a constant, then it is a terminal node with no links; if it is an identifier it is a nonterminal node.

Working memory is a set, which means that there can never be two elements in working memory at the same time that have the same identifier-attribute-value triple (this is prevented by the architecture). However, it is allowed to have multiple working memory elements that have the same identifier and attribute, but that each have different values. When this happens, we say the attribute is a *multi-valued attribute*, which is often shortened to be *multi-attribute*.

An object is, in a sense, defined by its augmentations and *not* by its identifier. On subsequent runs of the same Soar program, there may be an object with exactly the same augmentations, but a different identifier, and the program will still reason about the object appropriately. Identifiers are internal markers for Soar, so they can appear in working memory, but they never appear in a production.

There is no predefined relationship between objects in working memory and "real objects" in the outside world. Objects in working memory may refer to real objects, such as `block A`; a feature of an object, such as the color `red`; a relation between objects, such as `ontop`; classes of objects, such as `blocks`; etc. The names of attributes and values have no meaning to the Soar architecture (aside from a few WME's created by the architecture itself). For example, Soar doesn't care whether the things in the blocks world are called "blocks" or "cubes" or "chandeliers". It is up to the Soar programmer to pick suitable names and to use them consistently.

The elements in working memory come from one of four sources:

1. The actions of productions create most working memory elements. The actions of productions must not create or modify the working memory elements created by the decision procedure or the I/O system (described below).
2. The decision procedure automatically creates some special state augmentations (type, superstate, impasse, ...) when a state is created. States are created during initialization (the first state) or because of an impasse (a substate).
3. The decision procedure creates the operator augmentation of the state based on preferences. This records the selection of the current operator.

[2]In order to allow these links to have some substructure, the attribute name may be an identifier, which means that the attribute may itself have attributes and values, as specified by additional working memory elements.

4. The I/O system creates working memory elements on the input-link for sensory data.

The elements in working memory are removed in six different ways:

1. The decision procedure automatically removes all state augmentations it creates when the impasse that led to their creation is resolved.
2. The decision procedure removes the operator augmentation of the state when that operator is no longer selected as the current operator.
3. Production actions that use `reject` preferences remove working memory elements.
4. i-supported working memory elements are removed when the productions that created them no longer match.
5. The I/O system removes sensory data from the input-link when it is no longer valid.
6. The architecture automatically removes WME's that are no longer linked to a state (because some other WME has been removed).

For the most part, the user is free to use any attributes and values that are appropriate for the task. However, states have special augmentations that cannot be directly created, removed, or modified by rules. These include the augmentations created when a state is created, and the state's operator augmentation that signifies the current operator (and is created based on preferences). The specific attributes that Soar automatically creates are listed in Section 3.4. Productions may create any other attributes for states.

Preferences are held in *preference memory* where they cannot be tested by productions; however, `acceptable` preferences are held in *both* preference memory and in working memory. By making the `acceptable` preferences available in working memory, the acceptable preferences can be tested for in productions allowing the candidates operators to be compared before they are selected.

2.3 Productions: Long-term Knowledge

Soar represents long-term knowledge as *productions* that are stored in *production memory*, illustrated in Figure 2.7. Each production has a set of conditions and a set of actions. If the conditions of a production match working memory, the production *fires*, and the actions are performed.

2.3.1 The structure of a production

In the simplest form of a production, conditions and actions refer directly to the presence (or absence) of objects in working memory. For example, a production might say:

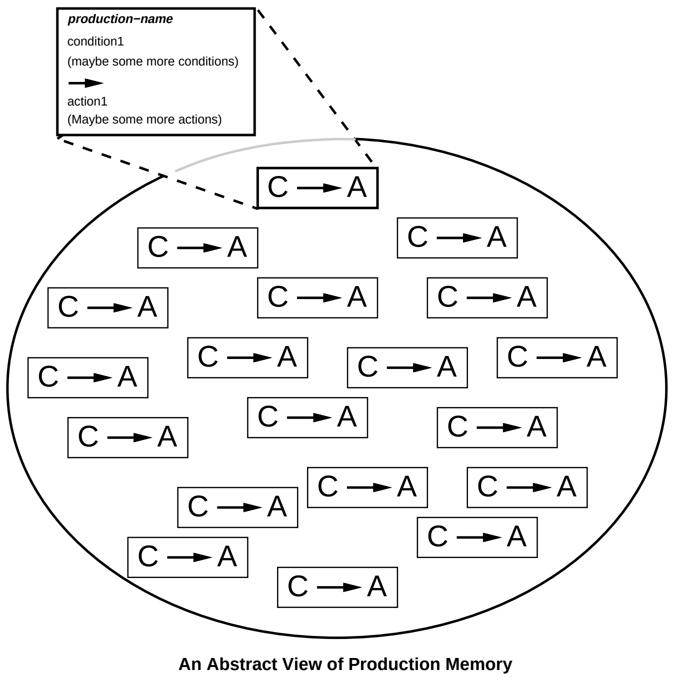
production−name
condition1
(maybe some more conditions)
action1
(Maybe some more actions)
C → A
An Abstract View of Production Memory

Figure 2.7: An abstract view of production memory. The productions are not related to one another.

```
CONDITIONS: block A is clear
            block B is clear
ACTIONS:    suggest an operator to move block A ontop of block B
```

This is not the literal syntax of productions, but a simplification. The actual syntax is presented in Chapter 3.

The conditions of a production may also specify the *absence* of patterns in working memory. For example, the conditions could also specify that "block A is not red" or "there are no red blocks on the table". But since these are not needed for our example production, there are no examples of negated conditions for now.

The order of the conditions of a production do not matter to Soar except that the first condition must directly test the state. Internally, Soar will reorder the conditions so that the matching process can be more efficient. This is a mechanical detail that need not concern most users. However, you may print your productions to the screen or save them in a file; if they are not in the order that you expected them to be, it is likely that the conditions have been reordered by Soar.

2.3.1.1 Variables in productions and multiple instantiations

In the example production above, the names of the blocks are "hardcoded", that is, they are named specifically. In Soar productions, variables are used so that a production can apply to a wider range of situations.

The variables are bound to specific symbols (identifiers, attributes, or values) in working memory elements by Soar's matching process. A production along with a specific and consistent set of variable bindings is called an *instantiation*. A production instantiation is consistent only if every occurrence of a variable is bound to the same value. Since the same production may match multiple times, each with different variable bindings, several instantiations of the same production may match at the same time and, therefore, fire at the same time. If blocks `A` and `B` are clear, the first production (without variables) will suggest one operator. However, if a production was created that used variables to test the names, this second production will be instantiated twice and therefore suggest *two* operators: one operator to move block `A` ontop of block `B` and a second operator to move block `B` ontop of block `A`.

Because the identifiers of objects are determined at runtime, literal identifiers cannot appear in productions. Since identifiers occur in every working memory element, variables must be used to test for identifiers, and multiple occurrences of the same variable is used to link conditions together.

Just as the elements of working memory must be linked to a state in working memory, so must the objects referred to in a production's conditions. That is, one condition must test a state object *and* all other conditions must test that same state or objects that are linked to that state.

2.3.2 Architectural roles of productions

Soar productions can fulfill four different roles, including the three knowledge-retrieval problem-solving functions and state elaboration function, as described on page 8:

1. Operator proposal
2. Operator comparison
3. *(Operator selection is not an act of knowledge retrieval)*
4. Operator application
5. State elaboration

A single production should not fulfill more than one of these roles (except for proposing an operator and creating an absolute preference for it). Although productions are not declared to be of one type or the other, Soar examines the structure of each production and classifies the rules automatically based on whether they propose and compare operators, apply operators, or elaborate the state.

2.3.3 Production Actions and Persistence

The two main actions of a production are to create preferences for operator selection, and create or remove working memory elements. For operator proposal and comparison, a production creates preferences for operator selection. These preferences should persist only as long as the production instantiation that created them continues to match. When the production instantiation no longer matches, the situation has changed, making the preference no longer relevant. Soar automatically removes the preferences in such cases. These preferences are said to have *I-support* (for "instantiation support"). Similarly, state elaborations are simple inferences are valid only so long as the production matches. Working memory elements created as state elaborations also have I-support and remain in working memory only as long as the production instantiation that created them continues to match working memory. For example, the set of relevant operators change as the state changes, so that the proposal of operators done with I-supported preferences. This way, the operator proposals will be retracted when they no longer apply to the current situation.

However, the actions of productions that apply an operator, either by adding or removing elements from working memory, need to persist even after the operator is no longer selected and operator application production instantiation no longer matches. For example, in placing a block on another, a condition is that the second block be clear. However, the action of placing the first block removes the fact that the second block is clear, so the condition will no longer be satisfied.

Thus, operator application productions do not retract their actions, even if they no longer match working memory. This is called *O-support* (for "operator support"). Working memory elements that participate in the application of operators are maintained throughout the existence of the state in which the operator is applied, unless explicitly remove (or if they become unlinked). Working memory elements are removed by a *reject* action of a operator-application rule.

Whether a working memory element receives O-support or I-support is determined by the structure of the production instantiation that create the working memory element. O-support is given only to working memory elements created by operator-application productions.

An operator-application production tests the current operator of a state and modifies the state. Thus, a working memory element receives O-support if it is for an augmentation of the current state or substructure of the state, and the conditions of the instantiation that created it test augmentations of the current operator.

When productions are matched, all productions that have their conditions met fire creating or removing working memory elements. Also, working memory elements and preferences that lose I-support are removed from working memory. Thus, several new working memory elements and preferences may be created, and several existing working memory elements and preferences may be removed at the same time. (Of course, all this doesn't happen literally at the same time, but the order of firings and retractions is unimportant, and happens in parallel from a functional perspective.)

2.4 Preference memory: Selection Knowledge

The selection of the current operator is determined by the *preferences* in *preference memory.* Preferences are suggestions or imperatives about the current operator, or information about how suggested operators compare to other operators. Preferences refer to operators by using the identifier of a working memory element that stands for the operator. After preferences have been created for a state, the decision procedures evaluates them to select the current operator for that state.

For an operator to be selected, there will be at least one preference for it, specifically, a preference to say that the value is a candidate for the operator attribute of a state (this is done with either an "`acceptable`" or "`require`" preference). There may also be others, for example to say that the value is "best".

The different preferences available and the semantics of preferences are explained in Section 2.4.1. Preferences remain in preference memory until removed for one of the reasons previously discussed in Section 2.3.3.

2.4.1 Preference semantics

This section describes the semantics of each type of preference. More details on the preference resolution process are provided in Appendix **??**.

Only a single value can be selected as the current operator, that is, all values are mutually exclusive. In addition, there is no implicit transitivity in the semantics of preferences. If A is indifferent to B, and B is indifferent to C, A and C will not be indifferent to one another unless there is a preference that A is indifferent to C (or C and A are both indifferent to all competing values).

Acceptable (+) An `acceptable` preference states that a value is a candidate for selection. All values, except those with `require` preferences, must have an `acceptable` preference in order to be selected. If there is only one value with an `acceptable` preference (and none with a `require` preference), that value will be selected as long as it does not also have a `reject` or a `prohibit` preference.

Reject (−) A `reject` preference states that the value is not a candidate for selection.

Better (>), Worse (<) A `better` or `worse` preference states, for the two values involved, that one value should not be selected if the other value is a candidate. `Better` and `worse` allow for the creation of a partial ordering between candidate values. `Better` and `worse` are simple inverses of each other, so that `A` better than `B` is equivalent to `B` worse than `A`.

Best (>) A `best` preference states that the value may be better than any competing value (unless there are other competing values that are also "best"). If a value is `best` (and not `rejected`, `prohibited`, or `worse` than another), it will be selected over any other value that is not also `best` (or `required`). If two such values are `best`, then any remaining preferences for those candidates (`worst`, `parallel`, `indifferent`) will be examined to determine the selection. Note that if a value (that is not `rejected` or `prohibited`) is `better` than a `best` value, the `better` value will be selected. (This result is counter-intuitive, but allows explicit knowledge about the relative worth of two values to dominate knowledge of only a single value. A `require` preference should be used when a value *must* be selected for the goal to be achieved.)

Worst (<) A `worst` preference states that the value should be selected only if there are no alternatives. It allows for a simple type of default specification. The semantics of the `worst` preference are similar to those for the `best` preference.

Indifferent (=) An `indifferent` preference states that there is positive knowledge that it does not matter which value is selected. This may be a binary preference, to say that two values are mutually indifferent, or a unary preference, to say that a single value is as good or as bad a choice as other expected alternatives.

When `indifferent` preferences are used to signal that it does not matter which operator is selected, by default, Soar chooses randomly from among the alternatives. (The `indifferent-selection` function can be used to change this behavior as described on page 142 in Chapter 6.)

Require (!) A `require` preference states that the value must be selected if the goal is to be achieved.

Prohibit (~) A `prohibit` preference states that the value cannot be selected if the goal is to be achieved. If a value has a `prohibit` preference, it will not be selected for a value of an augmentation, independent of the other preferences.

If there is an `acceptable` preference for a value of an operator, and there are no other competing values, that operator will be selected. If there are multiple `acceptable` preferences for the same state but with different values, the preferences must be evaluated to determine which candidate is selected.

If the preferences can be evaluated without conflict, the appropriate operator augmentation of the state will be are added to working memory. This can happen when they all suggest the same operator or when one operator is preferable to the others that have been suggested. When the preferences conflict, Soar reaches an impasse, as described in Section 2.6.

Preferences can be confusing; for example, there can be two suggested values that are both "best" (which again will lead to an impasse unless additional preferences resolve this conflict); or there may be one preference to say that value `A` is better than value `B` and a second preference to say that value `B` is better than value `A`.

2.5 Soar's Execution Cycle: Without Substates

The execution of a Soar program proceeds through a number of *cycles*. Each cycle has five phases:

1. Input: New sensory data comes into working memory.

2. Proposal: Productions fire (and retract) to interpret new data (state elaboration) propose operators for the current situation (operator proposal) and compare proposed operators (operator comparison). All of the actions of these productions are I-supported. All matched productions fire in parallel (and all retractions occur in parallel), and matching and firing continues until there are no more additional complete matches or retractions of productions (*quiescence*).
3. Decision: A new operator is selected, or an impasse is detected and a new state is created.
4. Application: Productions fire to apply the operator (operator application). The actions of these productions will be O-supported. Because of changes from operator application productions, other productions with I-supported actions may also match or retract. Just as during proposal, productions fire and retract in parallel until quiescence.
5. Output: Output commands are sent to the external environment.

The cycles continue until the halt action is issued from the Soar program (as the action of a production) or until Soar is interrupted by the user.

During the processing of these phases, it is possible that the preferences that resulted in the selection of the current operator could change. Whenever operator preferences change, the preferences are re-evaluated and if a different operator selection would be made, then the current operator augmentation of the state is immediately removed. However, a new operator is not selected until the next decision phase, when all knowledge has had a chance to be retrieved.

2.6 Impasses and Substates

When the decision procedure is applied to evaluate preferences and determine the operator augmentation of the state, it is possible that the preferences are either incomplete or inconsistent. The preferences can be incomplete in that no `acceptable` operators are suggested, or that there are insufficient preferences to distinguish among `acceptable` operators. The preferences can be inconsistent if, for instance, operator `A` is preferred to operator `B`, and operator `B` is preferred to operator `A`. Since preferences are generated independently, from different production instantiations, there is no guarantee that they will be consistent.

2.6.1 Impasse Types

There are four types of impasses that can arise from the preference scheme.

Tie impasse — A *tie* impasse arises if the preferences do not distinguish between two or more operators with `acceptable` preferences. If two operators both have

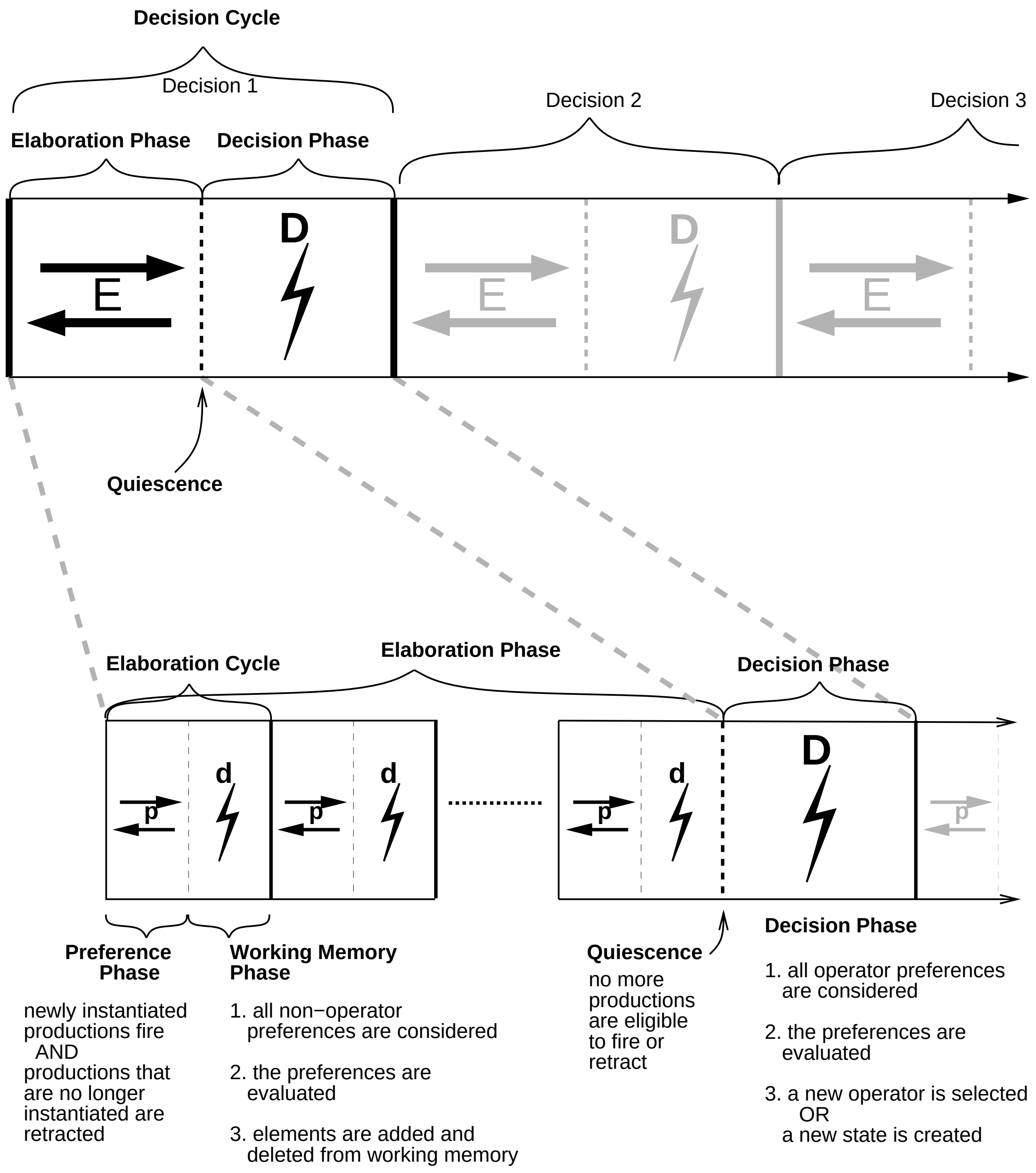
Decision Cycle
Decision 1
Decision 2
Decision 3
Elaboration Phase
Decision Phase
D
E
D
E
E
Quiescence
Elaboration Cycle
Elaboration Phase
Decision Phase
d
p
d
p
d
p
D
p
Preference Phase
newly instantiated productions fire AND productions that are no longer instantiated are retracted
Working Memory Phase
1. all non−operator preferences are considered
2. the preferences are evaluated
3. elements are added and deleted from working memory
Quiescence
no more productions are eligible to fire or retract
Decision Phase
1. all operator preferences are considered
2. the preferences are evaluated
3. a new operator is selected OR a new state is created

Figure 2.8: A detailed illustration of Soar's decision cycle: out of date

`best` or `worst` preferences, they will tie unless additional preferences distinguish between them.

Conflict impasse — A *conflict* impasse arises if at least two values have conflicting better or worse preferences (such as `A` is better than `B` and `B` is better than `A`) for an operator, and neither one is rejected, prohibited, or `required`.

Constraint-failure impasse — A *constraint-failure* impasse arises if there is more than one `required` value for an operator, or if a value has both a `require` and

```
Soar
  while (HALT not true) Cycle;

Cycle
  InputPhase;
  ProposalPhase;
  DecisionPhase;
  ApplicationPhase;
  OutputPhase;


ProposalPhase
  while (some I-supported productions are waiting to fire or retract)
    FireNewlyMatchedProductions;
    RetractNewlyUnmatchedProductions;

DecisionPhase
  for (each state in the stack,
       starting with the top-level state)
  until (a new decision is reached)
    EvaluateOperatorPreferences; /* for the state being considered */
    if (one operator preferred after preference evaluation)
      SelectNewOperator;
    else                 /* could be no operator available or */
      CreateNewSubstate;  /* unable to decide between more than one */

ApplicationPhase
  while (some productions are waiting to fire or retract)
    FireNewlyMatchedProductions;
    RetractNewlyUnmatchedProductions;
```

Figure 2.9: A simplified version of the Soar algorithm.

a `prohibit` preference. These preferences represent constraints on the legal selections that can be made for a decision and if they conflict, no progress can be made from the current situation and the impasse cannot be resolved by additional preferences.

No-change impasse — A *no-change* impasse arises if a new operator is not selected during the decision procedure. There are two types of no-change impasses: state no-change and operator no-change:

State no-change impasse — A state no-change impasse occurs when there are no `acceptable` (or `require`) preferences to suggest operators for the current state (or all the `acceptable` values have also been `rejected`). The decision procedure cannot select a new operator.

Operator no-change impasse — An operator no-change impasse occurs when either a new operator is selected for the current state but no additional productions match during the application phase, or a new operator is not selected during the next decision phase.

There can be only one type of impasse at a given level of subgoaling at a time. Given the semantics of the preferences, it is possible to have a tie or conflict impasse and a constraint-failure impasse at the same time. In these cases, Soar detects only the constraint-failure impasse.

The impasse is detected *during* the selection of the operator, but happens *because* one of the other four problem-solving functions was incomplete.

2.6.2 Creating New States

Soar handles these inconsistencies by creating a new state in which the goal of the problem solving is to resolve the impasse. Thus, in the substate, operators will be selected and applied in an attempt either to discover which or the tied operators should be selected, or to apply the selected operator piece by piece. The substate is often called a *subgoal* because it exists to resolve the impasse, but is sometimes called a substate because the representation of the subgoal in Soar is as a state.

The initial state in the subgoal contains a complete description of the cause of the impasse, such as the operators that could not be decided among (or that there were no operators proposed) and the state that the impasse arose in. From the perspective of the new state, the latter is called the *superstate*. Thus, the superstate is part of the substructure of each state, represented by the Soar architecture using the `superstate` attribute. (The initial state, created in the 0th decision cycle, contains a `superstate` attribute with the value of `nil` — the top-level state has no superstate.)

The knowledge to resolve the impasse may be retrieved by any type of problem solving, from searching to discover the implications of different decisions, to asking an outside agent for advice. There is no *a priori* restriction on the processing, except that it involves applying operators to states.

In the substate, operators can be selected and applied as Soar attempts to solve the subgoal. (The operators proposed for solving the subgoal may be similar to the operators in the superstate, or they may be entirely different.) While problem solving in the subgoal, additional impasses may be encountered, leading to new subgoals. Thus, it is possible for Soar to have a *stack* of subgoals, represented as states: Each state has a single superstate (except the initial state) and each state may have at most one substate. Newly created subgoals are considered to be added to the bottom of the stack; the first state is therefore called the *top-level state*.[3] See Figure 2.10 for a simplified illustrations of a subgoal stack.

[3]The original state is the "top" of the stack because as Soar runs, this state (created first), will be at the top of the computer screen, and substates will appear on the screen below the top-level state.

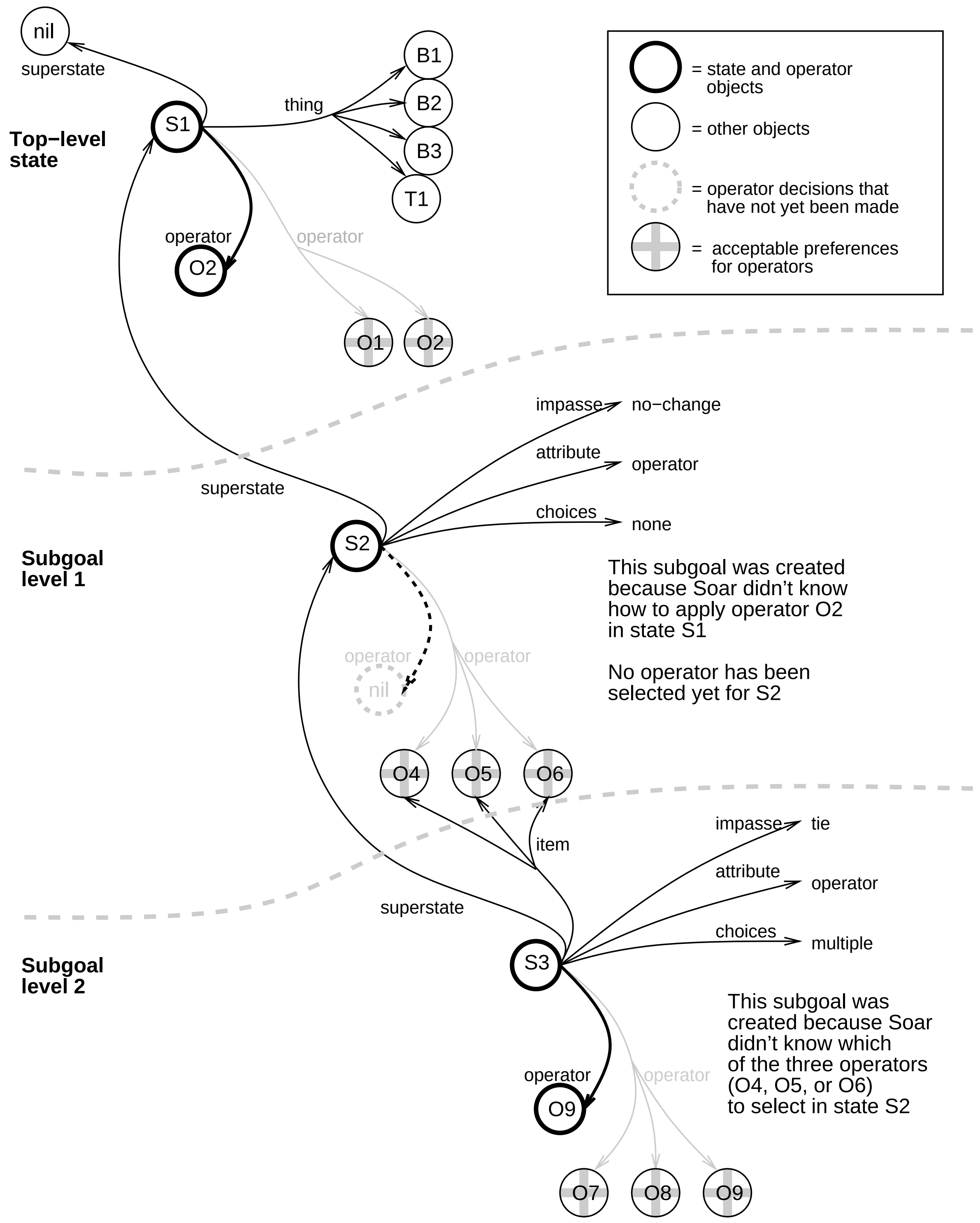
nil
superstate
S1
thing
B1
B2
B3
T1
Top−level state
operator
O2
operator
O1
O2
= state and operator objects
= other objects
= operator decisions that have not yet been made
= acceptable preferences for operators
impasse
no−change
attribute
operator
choices
none
superstate
S2
Subgoal level 1
This subgoal was created because Soar didn't know how to apply operator O2 in state S1
No operator has been selected yet for S2
operator
nil
operator
O4
O5
O6
item
impasse
tie
attribute
operator
choices
multiple
superstate
S3
Subgoal level 2
This subgoal was created because Soar didn't know which of the three operators (O4, O5, or O6) to select in state S2
operator
O9
operator
O7
O8
O9

Figure 2.10: A simplified illustration of a subgoal stack.

Soar continually attempts to retrieve knowledge relevant to all goals in the subgoal stack, although problem-solving activity will tend to focus on the most recently created state. However, problem solving is active at all levels, and productions that match at any level will fire.

2.6.3 Results

In order to resolve impasses, subgoals must generate results that allow the problem solving at higher levels to proceed. The *results* of a subgoal are the working memory elements and preferences that were created in the substate, and that are also linked directly or indirectly to a superstate (*any* superstate in the stack). A preference or working memory element is said to be created in a state if the production that created it tested that state and this is the most recent state that the production tested. Thus, if a production tests multiple states, the preferences and working memory elements in its actions are considered to be created in the most recent of those states (and is not considered to have been created in the other states). The architecture automatically detects if a preference or working memory element created in a substate is also linked to a superstate.

These working memory elements and preferences will not be removed when the impasse is resolved because they are still linked to a superstate, and therefore, they are called the *results of the subgoal.* A result has either I-support or O-support; the determination of support is described below.

A working memory element or preference will be a result if its identifier is already linked to a superstate. A working memory element or preference can also become a result indirectly if, after it is created and still in working memory or preference memory, its identifier becomes linked to a superstate through the creation of another result. For example, if the problem solving in a state constructs an operator for a superstate, it may wait until the operator structure is complete before creating an `acceptable` preference for the operator in the superstate. The `acceptable` preference is a result because it was created in the state and is linked to the superstate (and, through the superstate, is linked to the top-level state). The substructures of the operator then become results because the operator's identifier is now linked to the superstate. An indirect result is illustrated in Figure **??**).

Justifications: Determination of support for results

Some results receive I-support, while others receive O-support. The type of support received by a result is determined by the function it plays in the superstate, and not the function it played in the state in which it was created. For example, a result might be created through operator application in the state that created it; however, it might only be a state elaboration in the superstate. The first function would be lead to O-support, but the second would lead to I-support.

In order for the architecture to determine whether a result receives I-support or O-support, Soar must first determine the function that the working memory element or preference performs (that is, whether the result should be considered an operator application or not). To do this, Soar creates a temporary production, called a *justification.* The justification summarizes the processing in the substate that led to the result:

The conditions of a justification are those working memory elements that exist in the superstate (and above) that were necessary for producing the result. This is determined by collecting all of the working memory elements tested by the productions that fired in the subgoal that led to the creation of the result, and then removing those conditions that test working memory elements created in the subgoal.

The action of the justification is the result of the subgoal.

Soar determines I-support or O-support for the justification just as it would for any other production, as described in Section 2.3.3. If the justification is an operator application, the result will receive O-support. Otherwise, the result gets I-support from the justification. If a such a result loses I-support from the justification, it will be retracted if there is no other support. Justification are not added to production memory, but are otherwise treated as an instantiated productions that have already fired.

Justifications include any negated conditions that were in the original productions that participated in producing the results, and that test for the absence of superstate working memory elements. Negated conditions that test for the absence of working memory elements that are local to the substate are not included, which can lead to overgeneralization in the justification (see Section 4.6 on page 82 for details).

2.6.4 Removal of Substates: Impasse Resolution

Problem solving in substates is an important part of what Soar *does*, and an operator impasse does not necessarily indicate a problem in the Soar program. They are a way to decompose a complex problem into smaller parts and they provide a context for a program to deliberate about which operator to select. Operator impasses are necessary, for example, for Soar to do any learning about problem solving (as will be discussed in Chapter 4). This section describes how impasses may be resolved during the execution of a Soar program, how they may be eliminated during execution without being resolved, and some tips on how to modify a Soar program to prevent a specific impasse from occurring in the first place.

Resolving Impasses

An impasse is *resolved* when processing in a subgoal creates results that lead to the selection of a new operator at for the state where the impasse arose. When an operator impasse is resolved, Soar has an opportunity to learn, and the substate (and all its substructure) is removed from working memory.

Listed below are possible approaches for resolving specific types of impasses:

Tie impasse — A tie impasse can be resolved by productions that create preferences that prefer one option (`better`, `best`, `require`), eliminate alternatives (`worse`, `worst`, `reject`, `prohibit`), or make the all of the objects indifferent (`indifferent`).

Conflict impasse — A conflict impasse can be resolved by productions that create preferences to `require` one option (`require`), or eliminate the alternatives (reject, prohibit).

Constraint-failure impasse — A constraint-failure impasse cannot be resolved by additional preferences, but may be prevented by changing productions so that they create fewer `require` or `prohibit` preferences.

State no-change impasse — A state no-change impasse can be resolved by productions that create `acceptable` or `require` preferences for operators.

Operator no-change impasse — An operator no-change impasse can be resolved by productions that apply the operator, changing the state so the operator proposal no longer matches, or other operators are proposed and preferred.

Eliminating Impasses

An impasse is resolved when results are created that allow progress to be made in the state where the impasse arose. In Soar, impasse can be *eliminated* (but not resolved) when a higher level impasse is resolved, eliminated, or regenerated. In these cases, the impasse becomes irrelevant because higher-level processing can proceed. An impasse can also become irrelevant if input from the outside world changes working memory which in turn causes productions to fire that make it possible to select an operator. In all these cases, the impasse is eliminated, but not "resolved", and Soar does not learn in this situation.

Regenerating Impasses

An impasse is *regenerated* when the problem solving in the subgoal becomes *inconsistent* with the current situation. During problem solving in a subgoal, Soar monitors which aspect of the surrounding situation (the working memory elements that exist in superstates) the problem solving in the subgoal has depeneded upon. If those aspects of the surronding situation change, either because of changes in input or because of results, the problem solving in the subgoal is inconsistent, and the state created in response to the original impasse is removed and a new state is created. Problem solving will now continue from this new state. The impasse is not "resolved", and Soar does not learn in this situation.

The reason for regeneration is to guarantee that the working memory elements and preferences created in a substate are consistent with higher level states. As stated above, inconsistency can arise when a higher level state changes either as a result of changes in what is sensed in the external environment, or from results produced in the subgoal. The problem with inconsistency is that once inconsistency arises, the problem being solved in the subgoal may no longer be the problem that actually needs to be solved. Luckily, not all changes to a superstate lead to inconsistency.

In order to detect inconsistencies, Soar maintains a *dependency set* for every subgoal/substate. The dependency set consists of all working memory elements that were tested in the conditions of productions that created O-supported working memory elements that are directly or indirectly linked to the substate. Thus, whenever such an O-supported working memory element is created, Soar records which working memory elements that exist in a superstate were tested, directly or indirectly in creating that working memory element. dependency-set Whenever any of the working memory elements in the dependency set of a substate change, the substate is regenerated.

Note that the creation of I-supported structures in a subgoal does not increase the dependency set, nor do O-supported results. Thus, only subgoals that involve the creation of internal O-support working memory elements risk regeneration, and then only when the basis for the creation of those elements changes.

Substate Removal

Whenever a substate is removed, all working memory elements and preferences that were created in the substate that are not results are removed from working memory. In Figure 2.10, state `S3` will be removed from working memory when the impasse that created it is resolved. That is, when sufficient preferences have been generated so that one of the operators for state `S2` can be selected. When state `S3` is removed, operator `O9` and problem space `P3` will also be removed, as will the acceptable preferences for `O7`, `O8`, and `O9`, and the `impasse`, `attribute`, and `choices` augmentations of state `S3`. These working memory elements are removed because they are no longer linked to the subgoal stack. The acceptable preferences for operators `O4`, `O5`, and `O6` remain in working memory. They were linked to state `S3`, but since they are also linked to state `S2`, so they will stay in working memory until `S2` is removed (or until they are retracted or rejected).

2.6.5 Soar's Execution Cycle: With Substates

When there are multiple substates, Soar's cycle remains basically the same but has a few minor changes.

The first change is that during the decision procedure, Soar will detect impasses and create new substates. For example, following the proposal phase, the decision phase will detect if a decision cannot be made given the current preferences. If an impasse arises, a new substate is created and added to working memory.

The decision procedure will detect an operator no-change impasse as soon as an operator is selected and added to working memory by checking to see whether or not productions will create O-supported actions during the next application phae. If no O-suppored actions will be created, the decision procedure will immediately create an operator no-change impasse, and then proceed to output, input, and so on. In these cases, the operator no-change is made in the same decision as the operator selection. There will be cases where the operator no-change happens on the following decisions, such as when there are O-supported productions that will fire, but do not lead to a change in the selected operator.

The second change when there are multiple substates is that at each phase, Soar goes through the substates, from oldest (highest) to newest (lowest), completing any necessary processing at that level for that phase before doing any processing in the next substate. When firing productions for the proposal or application phases, Soar processes the firing (and retraction) of rules, starting from those matching the oldest substate to the newest. Whenever a production fires or retracts, changes are made to working memory and preference memory, possibly changing which productions will match at the lower levels (productions firing within a given level are fired in parallel – simulated). Productions firings at higher levels can resolve impasses and thus eliminate lower states before the productions at the lower level ever fire. Thus, whenever a level in the state stack is reached, all production activity is guaranteed to be consistent with any processing that has occurred at higher levels.

2.7 Learning

When an operator impasse is resolved, it means that Soar has, through problem solving, gained access to knowledge that was not readily available before. Therefore, when an impasse is resolved, Soar has an opportunity to learn, by summarizing and generalizing the processing in the substate.

Soar's learning mechanism is called *chunking*; it attempts to create a new production, called a chunk. The conditions of the chunk are the elements of the state that (through some chain of production firings) allowed the impasse to be resolved; the action of the production is the working memory element or preference that resolved the impasse (the result of the impasse). The conditions and action are variablized so that this new production may match in a similar situation in the future and prevent an impasse from arising.

Chunks are very similar to justifications in that they are both formed via the backtracing process and both create a result in their actions. However, there are some important distinctions:

1. Chunks are productions and are added to production memory. Justifications do not appear in production memory.
2. Justifications disappear as soon as the working memory element or preference they provide support for is removed.

3. Chunks contain variables so that they may match working memory in other situations; justifications are similar to an instantiated chunk.

2.8 Input and Output

Many Soar users will want their programs to interact with a real or simulated environment. For example, Soar programs may control a robot, receiving sensory inputs and sending command outputs. Soar programs may also interact with simulated environments, such as a flight simulator. Input is viewed as Soar's perception and output is viewed as Soar's motor abilities.

When Soar interacts with an external environment, it must make use of mechanisms that allow it to receive input from that environment and to effect changes in that environment; the mechanisms provided in Soar are called *input functions* and *output functions.*

Input functions add and delete elements from working memory in response to changes in the external environment.

Output functions attempt to effect changes in the external environment.

Input is processed at the beginning of each execution cycle and output occurs at the end of each execution cycle.

Using input and output functions with Soar will be discussed briefly in Chapter 3.5, and in more detail in *The Soar Advanced Applications Manual*.

Chapter 3

The Syntax of Soar Programs

This chapter describes in detail the syntax of elements in working memory, preference memory, and production memory, and how impasses and I/O are represented in working memory and in productions. Working memory elements and preferences are created as Soar runs, while productions are created by the user or through chunking. The bulk of this chapter explains the syntax for writing productions.

The first section of this chapter describes the structure of working memory elements in Soar; the second section describes the structure of preferences; and the third section describes the structure of productions. The fourth section describes the structure of impasses. An overview of how input and output appear in working memory is presented in the fifth section; the full discussion of Soar I/O found in *The Soar Advanced Applications Manual*.

This chapter assumes that you understand the operating principles of Soar, as presented in Chapter 2.

3.1 Working Memory

Working memory contains *working memory elements* (WME's). As described in Section 2.2, WME's can be created by the actions of productions, the evaluation of preferences, the Soar architecture, and via the input/output system.

A WME is a list consisting of three symbols: an *identifier*, an *attribute*, and a *value*, where the entire WME is enclosed in parentheses and the attribute is preceded by an up-arrow (^). A template for a working memory element is:

```
(identifier ^attribute value)
```

The identifier is an internal symbol, generated by the Soar architecture as it runs. The attribute and value can be either identifiers or constants; if they are identifiers, there are other working memory elements that have that identifier in their first position. As the previous sentences demonstrate, identifier is used to refer both to the first position of a working memory element, as well as to the symbols that occupy that position.

3.1.1 Symbols

Soar distinguishes between two types of working memory symbols: *identifiers* and *constants*.

Identifiers: An identifier is a unique symbol, created at runtime when a new object is added to working memory. The names of identifiers are created by Soar, and consist of a single uppercase letter followed by a string of digits, such as `G37` or `O22`.

(The Soar user interface will also allow users to specify identifiers using lowercase letters, for example, when using the `print` command. But internally, they are actually uppercase letters.)

Constants: There are three types of constants: integers, floating-point, and symbolic constants:

- Integer constants (numbers). The range of values depends on the machine and implementation you're using, but it is at least [-2 billion..2 billion].
- Floating-point constants (numbers). The range depends on the machine and implementation you're using.
- Symbolic constants. These are symbols with arbitrary names. A constant can use any combination of letters, digits, or `$%&*+-/:<=>?_` Other characters (such as blank spaces) can be included by surrounding the complete constant name with vertical bars: `|This is a constant|`. (The vertical bars aren't part of the name; they're just notation.) A vertical bar can be included by prefacing it with a backslash inside surrounding vertical bars: `|Odd-symbol\|name|`

Identifiers should not be confused with constants, although they may "look the same"; identifiers are generated (by the Soar architecture) at runtime and will not necessarily be the same for repeated runs of the same program. Constants are specified in the Soar program and will be the same for repeated runs.

Even when a constant "looks like" an identifier, it will not act like an identifier in terms of matching. A constant is printed surrounded by vertical bars whenever there is a possibility of confusing it with an identifier: `|G37|` is a constant while `G37` is an identifier. To avoid possible confusion, you should not use letter-number combinations as constants or for production names.

3.1.2 Objects

Recall from Section 2.2 that all WME's that share an identifier are collectively called an *object* in working memory. The individual working memory elements that make up an object are often called *augmentations*, because they augment the object. A template for an object in working memory is:

```
(identifier ^attribute-1 value-1 ^attribute-2 value-2
            ^attribute-3 value-3... ^attribute-n value-n)
```

For example, if you run Soar with the example blocks-world program described in Appendix **??**, after one elaboration cycle, you can look at the top-level state by using the `print` command:

```
soar> print s1
(S1 ^io I1 ^ontop O2 ^ontop O3 ^ontop O1 ^problem-space blocks
    ^superstate nil ^thing B3 ^thing T1 ^thing B1 ^thing B2
    ^type state)
```

The attributes of an object are printed in alphabetical order to make it easier to find a specific attribute.

Working memory is a set, so that at any time, there are never duplicate versions of working memory elements. However, it is possible for several working memory elements to share the same identifier and attribute but have different values. Such attributes are called multi-valued attributes or *multi-attributes*. For example, state `S1`, above, has two attributes that are multi-valued: `thing` and `ontop`.

3.1.3 Timetags

When a working memory element is created, Soar assigns it a unique integer *timetag*. The timetag is a part of the working memory element, and therefore, WME's are actually quadruples, rather than triples. However, the timetags are not represented in working memory and cannot be matched by productions. The timetags are used to distinguish between multiple occurrences of the same WME. As preferences change and elements are added and deleted from working memory, it is possible for a WME to be created, removed, and created again. The second creation of the WME — which bears the same identifier, attribute, and value as the first WME — is *different*, and therefore is assigned a different timetag. This is important because a production will fire only once for a given instantiation, and the instantiation is determined by the timetags that match the production and not by the identifier-attribute-value triples.

To look at the timetags of WMEs, the `wmes` command can be used:

```
soar> wmes s1
(3: S1 ^io I1)
(10: S1 ^ontop O2)
(9: S1 ^ontop O3)
(11: S1 ^ontop O1)
(4: S1 ^problem-space blocks)
(2: S1 ^superstate nil)
(6: S1 ^thing B3)
(5: S1 ^thing T1)
```

```
(8: S1 ^thing B1)
(7: S1 ^thing B2)
(1: S1 ^type state)
```

This shows all the individual augmentations of `S1`, each is preceded by an integer *timetag*.

3.1.4 Acceptable preferences in working memory

The acceptable preferences for the operator augmentations of states appear in working memory as identifier-attribute-value-preference quadruples. No other preferences appear in working memory. A template for an acceptable preference in working memory is:

```
(identifier ^operator value +)
```

For example, if you run Soar with the example blocks-world program described in Appendix ??, after the first operator has been selected, you can again look at the top-level state using the `wmes` command:

```
soar> wmes s1
(3: S1 ^io I1)
(9: S1 ^ontop O3)
(10: S1 ^ontop O2)
(11: S1 ^ontop O1)
(48: S1 ^operator O4 +)
(49: S1 ^operator O5 +)
(50: S1 ^operator O6 +)
(51: S1 ^operator O7 +)
(54: S1 ^operator O7)
(52: S1 ^operator O8 +)
(53: S1 ^operator O9 +)
(4: S1 ^problem-space blocks)
(2: S1 ^superstate nil)
(5: S1 ^thing T1)
(8: S1 ^thing B1)
(6: S1 ^thing B3)
(7: S1 ^thing B2)
(1: S1 ^type state)
```

The state `S1` has six augmentations of acceptable preferences for different operators (`O4` through `O9`). These have plus signs following the value to denote that they are acceptable preferences. The state has exactly one operator, `O7`. This state corresponds to the illustration of working memory in Figure 2.4.

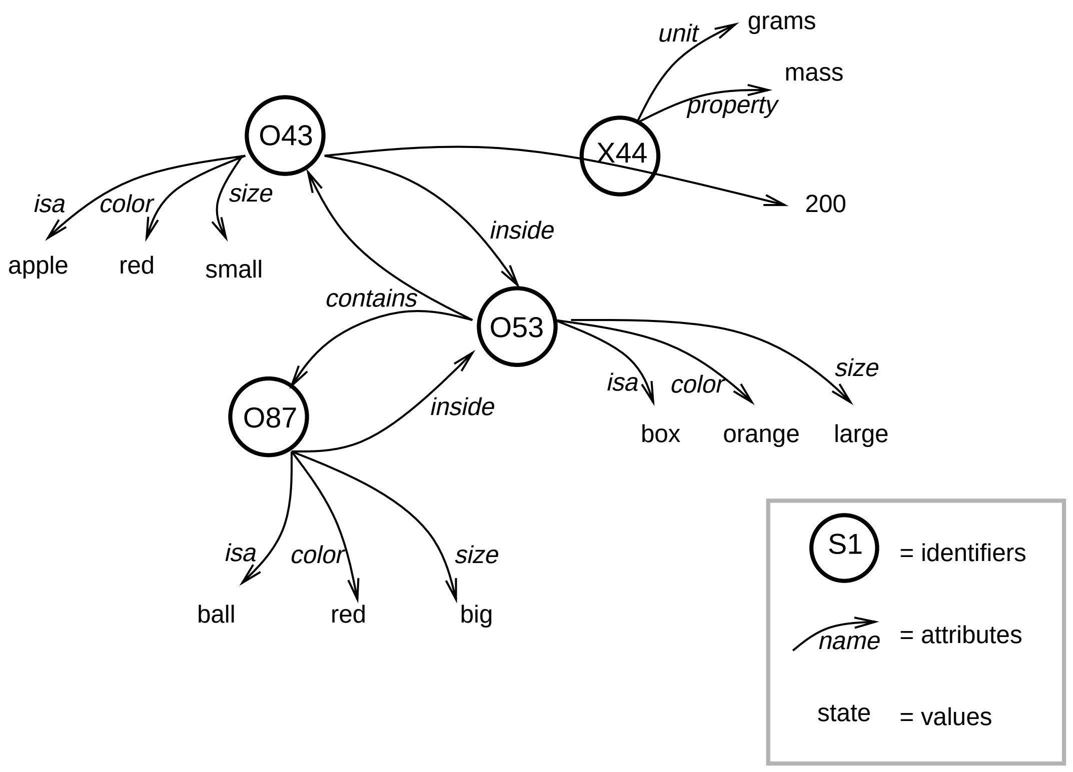

Figure 3.1: A semantic net illustration of four objects in working memory.

3.1.5 Working Memory as a Graph

Not only is working memory a set, it is also a graph structure where the identifiers are nodes, attributes are links, and constants are terminal nodes. Working memory is not an arbitrary graph, but a graph rooted in the states. Therefore, all WMEs are *linked* either directly or indirectly to a state. The impact of this constraint is that all WMEs created by actions are linked to WMEs tested in the conditions. The link is one-way, from the identifier to the value. Less commonly, the attribute of a WME may be an identifier.

Figure 3.1 illustrates four objects in working memory; the object with identifier `X44` has been linked to the object with identifier `O43`, using the attribute as the link, rather than the value. The objects in working memory illustrated by this figure are:

```
(O43 ^isa apple ^color red ^inside O53 ^size small ^X44 200)
(O87 ^isa ball ^color red ^inside O53 ^size big)
(O53 ^isa box ^size large ^color orange ^contains O43 O87)
(X44 ^unit grams ^property mass)
```

In this example, object `O43` and object `O87` are both linked to object `O53` through `(O53 ^contains O43)` and `(O53 ^contains O87)`, respectively (the `contains` attribute is a multi-valued attribute). Likewise, object `O53` is linked to object `O43` through `(O43`

`^inside O53`) and linked to object `O87` through (`O87 ^inside O53`). Object `X44` is linked to object `O43` through (`O43 ^X44 200`).

Links are transitive so that `X44` is linked to `O53` (because `O43` is linked to `O53` and `X44` is linked to `O43`). However, since links are not symmetric, `O53` is not linked to `X44`.

3.2 Preference Memory

Preferences are created by production firings and express the relative or absolute merits for selecting an operator for a state. When preferences express an absolute rating, they are identifier-attribute-value-preference quadruples; when preferences express relative ratings, they are identifier-attribute-value-preference-value quintuples

For example,

```
(S1 ^operator O3 +)
```

is a preference that asserts that operator O3 is an acceptable operator for state S1, while

```
(S1 ^operator O3 > O4)
```

is a preference that asserts that operator O3 is a better choice for the operator of state S1 than operator O4.

The semantics of preferences and how they are processed were described in Section 2.4, which also described each of the twelve different types of preferences. Multiple production instantiations may create identical preferences. Unlike working memory, preference memory is not a set: Duplicate preferences are allowed in preference memory.

3.3 Production Memory

Production memory contains productions, which can be loaded in by a user (typed in while Soar is running or `source`d from a file) or generated by chunking while Soar is running. Productions (both user-defined productions and chunks) may be examined using the `print` command, described in Section 6.2.6 on page 107.

Each production has three required components: a name, a set of conditions (also called the left-hand side, or LHS), and a set of actions (also called the right-hand side, or RHS). There are also two optional components: a documentation string and a type.

Syntactically, each production consists of the symbol `sp`, followed by: an opening curly brace, {; the production's name; the documentation string (optional); the production

```
sp {blocks-world*propose*move-block
   (state <s> ^problem-space blocks
              ^thing <thing1> {<> <thing1> <thing2>}
              ^ontop <ontop>)
   (<thing1> ^type block ^clear yes)
   (<thing2> ^clear yes)
   (<ontop> ^top-block <thing1>
            ^bottom-block <> <thing2>)
   -->
   (<s> ^operator <o> +)
   (<o> ^name move-block
        ^moving-block <thing1>
        ^destination <thing2>)}
```

Figure 3.2: An example production from the example blocks-world task.

type (optional); comments (optional); the production's conditions; the symbol `-->` (literally: dash-dash-greaterthan); the production's actions; and a closing curly brace, }. Each element of a production is separated by white space. Indentation and linefeeds are used by convention, but are not necessary.

```
sp {production-name
    Documentation string
    :type
    CONDITIONS
    -->
    ACTIONS
    }
```

An example production, named "`blocks-world*propose*move-block`", is shown in Figure 3.2. This production proposes operators named `move-block` that move blocks from one location to another. The details of this production will be described in the following sections.

Conventions for indenting productions

Productions in this manual are formatted using conventions designed to improve their readability. These conventions are not part of the required syntax. First, the name of the production immediately follows the first curly bracket after the `sp`. All conditions are aligned with the first letter after the first curly brace, and attributes of an object are all aligned The arrow is indented to align with the conditions and actions and the closing curly brace follows the last action.

3.3.1 Production Names

The name of the production is an almost arbitrary constant. (See Section 3.1.1 for a description of constants.) By convention, the name describes the role of the production, but functionally, the name is just a label primarily for the use of the programmer.

A production name should never be a single letter followed by numbers, which is the format of identifiers.

The convention for naming productions is to separate important elements with asterisks; the important elements that tend to appear in the name are:

1. The name of the task or goal (e.g., `blocks-world`).
2. The name of the architectural function (e.g., `propose`).
3. The name of the operator (or other object) at issue. (e.g., `move-block`)
4. Any other relevant details.

This name convention enables one to have a good idea of the function of a production just by examining its name. This can help, for example, when you are watching Soar run and looking at the specific productions that are firing and retracting. Since Soar uses white space to delimit components of a production, if whitespace inadvertently occurs in the production name, soar will complain that an open parenthesis was expected to start the first condition.

3.3.2 Documentation string (optional)

A production may contain an optional documentation string. The syntax for a documentation string is that it is enclosed in double quotes and appears after the name of the production and before the first condition (and may carry over to multiple lines). The documentation string allows the inclusion of internal documentation about the production; it will be printed out when the production is printed using the `print` command.

3.3.3 Production type (optional)

A production may also include an optional *production type*, which may specify that the production should be considered a default production (`:default`) or a chunk (`:chunk`), or may specify that a production should be given O- support (`:o-support`) or I-support (`:i-support`). Users are discouraged from using these types. These types are described in Section 6.1.9, which begins on Page 95.

3.3.4 Comments (optional)

Productions may contain comments, which are not stored in Soar when the production is loaded, and are therefore not printed out by the `print` command. A comment is begun with a pound sign character `#` and ends at the end of the line. Thus, everything following the `#` is not considered part of the production, and comments that run across multiple lines must each begin with a `#`.

For example:

```
sp {blocks-world*propose*move-block
   (state <s> ^problem-space blocks
              ^thing <thing1> {<> <thing1> <thing2>}
              ^ontop <ontop>)
   (<thing1> ^type block ^clear yes)
   (<thing2> ^clear yes)
#   (<ontop> ^top-block <thing1>
#           ^bottom-block <> <thing2>)
   -->
   (<s> ^operator <o> +)
   (<o> ^name move-block         # you can also use in-line comments
        ^moving-block <thing1>
        ^destination <thing2>)}
```

When commenting out conditions or actions, be sure that all parentheses remain balanced outside the comment.

External comments

Comments may also appear in a file with Soar productions, outside the curly braces of the `sp` command. Comments must either start a new line with a `#` or start with `;#`. In both cases, the comment runs to the end of the line.

```
# imagine that this is part of a "Soar program" that contains
# Soar productions as well as some other code.

source blocks.soar      ;# this is also a comment
```

3.3.5 The condition side of productions (or LHS)

The condition side of a production, also called the left-hand side (or LHS) of the production, is a pattern for matching one or more WMEs. When all of the conditions of a production match elements in working memory, the production is said to be instantiated, and is ready to perform its action.

The following subsections describe the condition side of a production, including predicates, disjunctions, conjunctions, negations, acceptable preferences for operators, and a few advanced topics. A grammar for the condition side is given in Appendix ??.

3.3.5.1 Conditions

The condition side of a production consists of a set of conditions. Each condition tests for the existence or absence (explained later in Section 3.3.5.6) of working memory elements. Each condition consists of a open parenthesis, followed by a test for the identifier, and the tests for augmentations of that identifier, in terms of attributes and values. The condition is terminated with a close parenthesis. Thus, a single condition might test properties of a single working memory element, or properties of multiple working memory elements that constitute an object.

```
(identifier-test ^attribute1-test value1-test
                 ^attribute2-test value2-test
                 ^attribute3-test value3-test
                 ...)
```

The first condition in a production must match against a state in working memory. Thus, the first condition must begin with the additional symbol "state". All other conditions and actions must be *linked* directly or indirectly to this condition. This linkage may be direct to the state, or it may be indirect, through objects specified in the conditions. If the identifiers of the actions are not linked to the state, a warning is printed when the production is parsed, and the production is not stored in production memory. In the actions of the example production shown in Figure 3.2, the operator preference is directly linked to the state and the remaining actions are linked indirectly via the operator first preference.

Although all of the attribute tests in the template above are followed by value tests, it is possible to test for only the existence of an attribute and not test any specific value by just including the attribute and no value. Another exception to the above template is operator preferences, which have the following structure where a plus sign follows the value test.

```
(state-identifier-test ^operator value1-test +
                 ...)
```

In the remainder of this section, we describe the different tests that can be used for identifiers, attributes, and values. The simplest of these is a constant, where the constant specified in the attribute or value must match the same constant in a working memory element.

3.3.5.2 Variables in productions

Variables match against constants in working memory elements in the identifier, attribute, or value positions. Variables can be further constrained by additional tests (described in later sections) or by multiple occurrences in conditions. If a variable occurs more than once in the condition of a production, the production will match only if the variables match the same identifier or constant. However, there is no restriction that prevents different variables from binding to the same identifier or constant.

Because identifiers are generated by Soar at run time, it impossible to include tests for specific identifiers in conditions. Therefore, variables are used in conditions whenever an identifier is to be matched.

Variables also provide a mechanism for passing identifiers and constants which match in conditions to the action side of a rule.

Syntactically, a variable is a symbol that begins with a left angle-bracket (i.e., `<`), ends with a right angle-bracket (i.e., `>`), and contains at least one alphanumeric symbol in between.

In the example production in Figure 3.2, there are seven variables: `<s>`, `<clear1>`, `<clear2>`, `<ontop>`, `<block1>`, `<block2>`, and `<o>`.

The following table gives examples of legal and illegal variable names.

Legal variables	**Illegal variables**
`<s>`	`<>`
`<1>`	`<1`
`<variable1>`	`variable>`
`<abc1>`	`<a b>`

3.3.5.3 Predicates for values

A test for an identifier, attribute, or value in a condition (whether constant or variable) can be modified by a preceding predicate. There are six predicates that can be used: `<>, <=>, <, <=, >=, >`.

Predicate	**Semantics of Predicate**
`<>`	Not equal. Matches anything except the value immediately following it.
`<=>`	Same type. Matches any symbol that is the same type (identifier, integer, floating-point, non-numeric constant) as the value immediately following it.
`<`	Numerically less than the value immediately following it.
`<=`	Numerically less than or equal to the value immediately following it.
`>=`	Numerically greater than or equal to the value immediately following it.
`>`	Numerically greater than the value immediately following it.

The following table shows examples of legal and illegal predicates:

Legal predicates	**Illegal predicates**
`> <valuex>`	`> > <valuey>`
`< 1`	`1 >`
`<=> <y>`	`= 10`

Example Production

```
sp {propose-operator*to-show-example-predicate
   (state <s> ^car <c>)
   (<c> ^style convertible ^color <> rust)
   -->
   (<s> ^operator <o> +)
   (<o> ^name drive-car ^car <c>) }
```

In this production, there must be a "color" attribute for the working memory object that matches `<c>`, and the value of that attribute must not be "rust".

3.3.5.4 Disjunctions of values

A test for an identifier, attribute, or value may also be for a disjunction of constants. With a disjunction, there will be a match if any one of the constants is found in a working memory element (and the other parts of the working memory element matches). Variables and predicates may not be used within disjunctive tests.

Syntactically, a disjunctive test is specified with double angle brackets (i.e., `<<` and `>>`). There must be spaces separating the brackets from the constants.

The following table provides examples of legal and illegal disjunctions:

Legal disjunctions	**Illegal disjunctions**
`<< A B C 45 I17 >>`	`<< <A> A >>`
`<< 5 10 >>`	`<< < 5 > 10 >>`
`<< good-morning good-evening >>`	`<<A B C >>`

Example Production

For example, the third condition of the following production contains a disjunction that restricts the color of the table to `red` or `blue`:

```
sp {blocks*example-production-conditions
   (state ^operator <o> + ^table <t>)
   (<o> ^name move-block)
   (<t> ^type table ^color << red blue >> )
   -->
   ... }
```

Note

Disjunctions of complete conditions are not allowed in Soar. Multiple (similar) productions fulfill this role.

3.3.5.5 Conjunctions of values

A test for an identifier, attribute, or value in a condition may include a conjunction of tests, all of which must hold for there to be a match.

Syntactically, conjuncts are contained within curly braces (i.e., { and }). The following table shows some examples of legal and illegal conjunctive tests:

Legal conjunctions	**Illegal conjunctions**
`{ <= <a> >= <b> }`	`{ <x> < <a> + <b> }`
`{ <x> > <y> }`	`{ > > <b> }`
`{ <> <x> <y> }`	
`{ << A B C >> <x> }`	
`{ <=> <x> > <y> << 1 2 3 4 >> <z> }`	

Because those examples are a bit difficult to interpret, let's go over the legal examples one by one to understand what each is doing.

In the first example, the value must be less than or equal to the value bound to variable `<a>` and greater than or equal to the value bound to variable `<b>`.

In the second example, the value is bound to the variable `<x>`, which must also be greater that the value bound to variable `<y>`.

In the third example, the value must not be equal to the value bound to variable `<x>` and should be bound to variable `<y>`. Note the importance of order when using conjunctions with predicates: in the second example, the predicate modifies `<y>`, but in the third example, the predicate modifies `<x>`.

In the fourth example, the value must be one of `A`, `B`, or `C`, and the second conjunctive test binds the value to variable `<x>`.

In the fifth example, there are four conjunctive tests. First, the value must be the same type as the value bound to variable `<x>`. Second, the value must be greater than the value bound to variable `<y>`. Third, the value must be equal to `1`, `2`, `3`, or `4`. Finally, the value should be bound to variable `<z>`.

In Figure 3.2, a conjunctive test is used for the `thing` attribute in the first condition.

3.3.5.6 Negated conditions

In addition to the positive tests for elements in working memory, conditions can also test for the absence of patterns. A *negated condition* will be matched only if there does not exist a working memory element consistent with its tests and variable bindings. Thus, it is a test for the *absence* of a working memory element.

Syntactically, a negated condition is specified by preceding a condition with a dash (i.e., "`-`").

For example, the following condition tests the absence of a working memory element of the object bound to `<p1>` ^`type father`.

```
-(<p1> ^type father)
```

A negation can be used within an object with many attribute-value pairs by having it precede a specific attribute:

```
(<p1> ^name john -^type father ^spouse <p2>)
```

In that example, the condition would match if there is a working memory element that matches `(<p1>` ^`name john)` and another that matches `(<p1>` ^`spouse <p2>)`, but is no working memory element that matches `(<p1>` ^`type father)` (when `p1` is bound to the same identifier).

On the other hand, the condition:

```
-(<p1> ^name john ^type father ^spouse <p2>)
```

would match only if there is no object in working memory that matches all three attribute-value tests.

Example Production

```
sp {default*evaluate-object
   (state <ss> ^operator <so>)
   (<so> ^type evaluation
         ^superproblem-space <p>)
  -(<p> ^default-state-copy no)
   -->
   (<so> ^default-state-copy yes) }
```

Notes

One use of negated conditions to avoid is testing for the absence of the working memory element that a production creates with I-support; this would lead to an "infinite loop" in your Soar program, as Soar would repeatedly fire and retract the production.

3.3.5.7 Negated conjunctions of conditions

Conditions can be grouped into conjunctive sets by surrounding the set of conditions with { and }. The production compiler groups the test in these conditions together. This grouping allows for negated tests of more than one working memory element at a time. In the example below, the state is tested to ensure that it does not have an object on the table.

```
sp {blocks*negated-conjunction-example
   (state <s> ^name top-state)
  -{(<s> ^ontop <on>)
    (<on> ^bottom-object <bo>)
    (<bo> ^type table)}
   -->
   (<s> ^nothing-ontop-table true) }
```

When using negated conjunctions of conditions, the production has nested curly braces. One set of curly braces delimits the production, while the other set delimits the conditions to be conjunctively negated.

If only the last condition, `(<bo> ^type table)` were negated, the production would match only if the state *had* an ontop relation, and the ontop relation had a bottom-object, but the bottom object wasn't a table. Using the negated conjunction, the production will also match when the state has no ontop augmentation or when it has an ontop augmentation that doesn't have a bottom-object augmentation.

The semantics of negated conjunctions can be thought of in terms of mathematical logic, where the negation of $(A \wedge B \wedge C)$:

$\neg(A \wedge B \wedge C)$

can be rewritten as:

$(\neg A) \vee (\neg B) \vee (\neg C)$

That is, "not (A and B and C)" becomes "(not A) or (not B) or (not C)".

3.3.5.8 Multi-valued attributes

An object in working memory may have multiple augmentations that specify the same attribute with different values; these are called multi-valued attributes, or multi-attributes for short. To shorten the specification of a condition, tests for multi-valued attributes can be shortened so that the value tests are together.

For example, the condition:

```
(<p1> ^type father ^child sally ^child sue)
```

could also be written as:

```
(<p1> ^type father ^child sally sue)
```

Multi-valued attributes and variables

When variables are used with multi-valued attributes, remember that variable bindings are not unique unless explicitly forced to be so. For example, to test that an object has two values for attribute `child`, the variables in following condition can match to the same value.

```
(<p1> ^type father ^child <c1> <c2>)
```

To do tests for multi-valued attributes with variables correctly, conjunctive tests must be used, as in:

```
(<p1> ^type father ^child <c1> {<> <c1> <c2>})
```

The conjunctive test {`<> <c1> <c2>`} ensures that `<c2>` will bind to a different value than `<c1>` binds to.

Negated conditions and multi-valued attributes

A negation can also precede an attribute with multiple values. In this case it tests for the absence of the conjunction of the values. For example

```
(<p1> ^name john -^child oprah uma)
```

is the same as

```
(<p1> ^name john)
-{(<p1> ^child oprah)
  (<p1> ^child uma)}
```

and the match is possible if either `(<p1> ^child oprah)` or `(<p1> ^child uma)` cannot be found in working memory with the binding for `<p1>` (but not if both are present).

3.3.5.9 Acceptable preferences for operators

The only preferences that can appear in working memory are acceptable preferences for operators, and therefore, the only preferences that may appear in the conditions of a production are acceptable preferences for operators.

Acceptable preferences for operators can be matched in a condition by testing for a "+" following the value. This allows a production to test the existence of a candidate operator and its properties, and possibly create a preference for it, before it is selected.

In the example below, ^`operator <o> +` matches the acceptable preference for the operator augmentation of the state. *This does not test that operator* `<o>` *has been selected as the current operator.*

```
sp {blocks*example-production-conditions
   (state ^operator <o> + ^table <t>)
   (<o> ^name move-block)
   -->
   ... }
```

In the example below, the production tests the state for acceptable preferences for two different operators (and also tests that these operators move different blocks):

```
sp {blocks*example-production-conditions
   (state ^operator <o1> + <o2> + ^table <t>)
   (<o1> ^name move-block ^moving-block <m1> ^destination <d1>)
   (<o2> ^name move-block ^moving-block {<m2> <> <m1>}
         ^destination <d2>)
   -->
   ... }
```

3.3.5.10 Attribute tests

The previous examples applied all of the different test to the values of working memory elements. All of the tests that can be used for values can also be used for attributes and identifiers (except those including constants).

Variables in attributes

Variables may be used with attributes, as in:

```
sp {blocks*example-production-conditions
   (state <s> ^operator <o> +
              ^thing <t> {<> <t> <t2>} )
   (operator <o> ^name group
                 ^by-attribute <a>
                 ^moving-block <t>
                 ^destination <t2>)
   (<t> ^type block ^<a> <x>)
   (<t2> ^type block ^<a> <x>)
   -->
   (<s> ^operator <o> >) }
```

This production tests that there is acceptable operator that is trying to group blocks according to some attribute, `<a>`, and that block `<t>` and `<t2>` both have this attribute (whatever it is), and have the same value for the attribute.

Predicates in attributes

Predicates may be used with attributes, as in:

```
sp {blocks*example-production-conditions
   (state ^operator <o> + ^table <t>)
   (<t> ^<> type table)
   -->
   ... }
```

which tests that the object with its identifier bound to `<t>` must have an attribute whose value is `table`, but the name of this attribute is not `type`.

Disjunctions of attributes

Disjunctions may also be used with attributes, as in:

```
sp {blocks*example-production-conditions
   (state ^operator <o> + ^table <t>)
   (<t> ^<< type name>> table)
   -->
   ... }
```

which tests that the object with its identifier bound to `<t>` must have either an attribute `type` whose value is `table` or an attribute `name` whose value is `table`.

Conjunctive tests for attributes

Section 3.3.5.5 illustrated the use of conjunctions for the values in conditions. Conjunctive tests may also be used with attributes, as in:

```
sp {blocks*example-production-conditions
   (state ^operator <o> + ^table <t>)
   (<t> ^{<ta> <> name} table)
   -->
   ... }
```

which tests that the object with its identifier bound to `<t>` must have an attribute whose value is `table`, and the name of this attribute is not `name`, and the name of this attribute (whatever it is) is bound to the variable `<ta>`.

When attribute predicates or attribute disjunctions are used with multi-valued attributes, the production is rewritten internally to use a conjunctive test for the attribute; the conjunctive test includes a variable used to bind to the attribute name. Thus,

```
(<p1> ^type father ^ <> name sue sally)
```

is interpreted to mean:

```
(<p1> ^type father ^ {<> name <a*1>} sue ^ <a*1> sally)
```

3.3.5.11 Attribute-path notation

Often, variables appear in the conditions of productions only to link the value of one attribute with the identifier of another attribute. Attribute-path notation provides a shorthand so that these intermediate variables do not need to be included.

Syntactically, path notation lists a sequence of attributes separated by dots (.), after the ^ in a condition.

For example, using attribute path notation, the production:

```
sp {blocks-world*monitor*move-block
   (state <s> ^operator <o>)
   (<o> ^name move-block
        ^moving-block <block1>
        ^destination <block2>)
   (<block1> ^name <block1-name>)
   (<block2> ^name <block2-name>)
   -->
   (write (crlf) |Moving Block: | <block1-name>
                 | to: | <block2-name> ) }
```

could be written as:

```
sp {blocks-world*monitor*move-block
   (state <s> ^operator <o>)
   (<o> ^name move-block
        ^moving-block.name <block1-name>
        ^destination.name <block2-name>)
   -->
   (write (crlf) |Moving Block: | <block1-name>
                 | to: | <block2-name> ) }
```

Attribute-path notation yields shorter productions that are easier to write, less prone to errors, and easier to understand.

When attribute-path notation is used, Soar internally expands the conditions into the multiple Soar objects, creating its own variables as needed. Therefore, when you print a production (using the `print` command), the production will not be represented using attribute-path notation.

Negations and attribute path notation

A negation may be used with attribute path notation, in which case it amounts to a negated conjunction. For example, the production:

```
sp {blocks*negated-conjunction-example
   (state <s> ^name top-state)
  -{(<s> ^ontop <on>)
    (<on> ^bottom-object <bo>)
    (<bo> ^type table)}
   -->
   (<s> ^nothing-ontop-table true) }
```

could be rewritten as:

```
sp {blocks*negated-conjunction-example
   (state <s> ^name top-state -^ontop.bottom-object.type table)
   -->
   (<s> ^nothing-ontop-table true) }
```

Multi-valued attributes and attribute path notation

Attribute path notation may also be used with multi-valued attributes, such as:

```
sp {blocks-world*propose*move-block
   (state <s> ^problem-space blocks
              ^clear.block <block1> { <> <block1> <block2> }
              ^ontop <ontop>)
   (<block1> ^type block)
   (<ontop> ^top-block <block1>
            ^bottom-block <> <block2>)
   -->
   (<s> ^operator <o> +)
   (<o> ^name move-block +
        ^moving-block <block1> +
        ^destination <block2> +) }
```

Multi-attributes and attribute-path notation

Note: It would not be advisable to write the production in Figure 3.2 using attribute-path notation as follows:

```
sp {blocks-world*propose*move-block*dont-do-this
```

```
   (state <s> ^problem-space blocks
              ^clear.block <block1>
              ^clear.block { <> <block1> <block2> }
              ^ontop.top-block <block1>
              ^ontop.bottom-block <> <block2>)
   (<block1> ^type block)
   -->
   ...
   }
```

This is not advisable because it corresponds to a different set of conditions than those in the original production (the `top-block` and `bottom-block` need not correspond to the same `ontop` relation). To check this, we could print the original production at the Soar prompt:

```
soar> print blocks-world*propose*move-block*dont-do-this
sp {blocks-world*propose*move-block*dont-do-this
    (state <s> ^problem-space blocks ^thing <thing2>
          ^thing { <> <thing2> <thing1> } ^ontop <o*1> ^ontop <o*2>)
    (<thing2> ^clear yes)
    (<thing1> ^clear yes ^type block)
    (<o*1> ^top-block <thing1>)
    (<o*2> ^bottom-block { <> <thing2> <b*1> })
    -->
    (<s> ^operator <o> +)
    (<o> ^name move-block
         ^moving-block <thing1>
         ^destination <thing2>) }
```

Soar has expanded the production into the longer form, and created two distinctive variables, <`o*1`> and <`o*2`> to represent the `ontop` attribute. These two variables will not necessarily bind to the same identifiers in working memory.

Negated multi-valued attributes and attribute-path notation

Negations of multi-valued attributes can be combined with attribute-path notation. However; it is very easy to make mistakes when using negated multi-valued attributes with attribute-path notation. Although it is possible to do it correctly, we strongly discourage its use.

For example,

```
sp {blocks*negated-conjunction-example
   (state <s> ^name top-state -^ontop.bottom-object.name table A)
   -->
   (<s> ^nothing-ontop-A-or-table true) }
```

gets expanded to:

```
sp {blocks*negated-conjunction-example
   (state <s> ^name top-state)
  -{(<s> ^ontop <o*1>)
    (<o*1> ^bottom-object <b*1>)
    (<b*1> ^name A)
    (<b*1> ^name table)}
   -->
   (<s> ^nothing-ontop-A-or-table true) }
```

This example does not refer to two different blocks with different names. It tests that there is not an `ontop` relation with a `bottom-block` that is named `A` and named `table`. Thus, this production probably should have been written as:

```
sp {blocks*negated-conjunction-example
   (state <s> ^name top-state
              -^ontop.bottom-object.name table
              -^ontop.bottom-object.name A)
   -->
   (<s> ^nothing-ontop-A-or-table true) }
```

which expands to:

```
sp {blocks*negated-conjunction-example
   (state <s> ^name top-state)
  -{(<s> ^ontop <o*2>)
    (<o*2> ^bottom-object <b*2>)
    (<b*2> ^name a)}
  -{(<s> ^ontop <o*1>)
    (<o*1> ^bottom-object <b*1>)
    (<b*1> ^name table)}
   -->
   (<s> ^nothing-ontop-a-or-table true +) }
```

Notes on attribute-path notation

- Attributes specified in attribute-path notation may not start with a digit. For example, if you type ^`foo.3.bar`, Soar thinks the `.3` is a floating-point number. (Attributes that don't appear in path notation can begin with a number.)
- Attribute-path notation may be used to any depth.
- Attribute-path notation may be combined with structured values, described in Section 3.3.5.12.

3.3.5.12 Structured-value notation

Another convenience that eliminates the use of intermediate variables is structured-value notation.

Syntactically, the attributes and values of a condition may be written where a variable would normally be written. The attribute-value structure is delimited by parentheses.

Using structured-value notation, the production in Figure 3.2 (on page 41) may also be written as:

```
sp {blocks-world*propose*move-block
   (state <s> ^problem-space blocks
              ^thing <thing1> {<> <thing1> <thing2>}
              ^ontop (^top-block <thing1>
                      ^bottom-block <> <thing2>))
   (<thing1> ^type block ^clear yes)
   (<thing2> ^clear yes)
-->
   (<s> ^operator <o> +)
   (<o> ^name move-block
        ^moving-block <thing1>
        ^destination <thing2>) }
```

Thus, several conditions may be “collapsed” into a single condition.

Using variables within structured-value notation

Variables are allowed within the parentheses of structured-value notation to specify an identifier to be matched elsewhere in the production. For example, the variable `<ontop>` could be added to the conditions (although it are not referenced again, so this is not helpful in this instance):

```
sp {blocks-world*propose*move-block
   (state <s> ^problem-space blocks
              ^thing <thing1> {<> <thing1> <thing2>}
              ^ontop (<ontop>
                      ^top-block <thing1>
                      ^bottom-block <> <thing2>))
   (<thing1> ^type block ^clear yes)
   (<thing2> ^clear yes)
   -->
   (<s> ^operator <o> +)
   (<o> ^name move-block
        ^moving-block <thing1>
        ^destination <thing2>) }
```

Structured values may be nested to any depth. Thus, it is possible to write our example production using a single condition with multiple structured values:

```
sp {blocks-world*propose*move-block
   (state <s> ^problem-space blocks
              ^thing <thing1>
                     ({<> <thing1> <thing2>}
                      ^clear yes)
              ^ontop (^top-block
                        (<thing1>
                         ^type block
                         ^clear yes)
                      ^bottom-block <> <thing2>) )
   -->
   (<s> ^operator <o> +)
   (<o> ^name move-block
        ^moving-block <thing1>
        ^destination <thing2>) }
```

Notes on structured-value notation

- Attribute-path notation and structured-value notation are orthogonal and can be combined in any way. A structured value can contain an attribute path, or a structure can be given as the value for an attribute path.
- Structured-value notation may also be combined with negations and with multi-attributes.
- Structured-value notation may not be used in the actions of productions.

3.3.6 The action side of productions (or RHS)

The action side of a production, also called the right-hand side (or RHS) of the production, consists of individual actions that can:

- Add new elements to working memory.
- Remove elements from working memory.
- Create preferences.
- Perform other actions

When the conditions of a production match working memory, the production is said to be instantiated, and the production will fire during the next elaboration cycle. Firing the production involves performing the actions *using the same variable bindings* that formed the instantiation.

3.3.6.1 Variables in Actions

Variables can be used in actions. A variable that appeared in the condition side will be replaced with the value that is was bound to in the condition. A variable that appears only in the action side will be bound to a new identifier that begins with the first letter of that variable (e.g., `<o>` might be bound to `o234`). This symbol is guaranteed to be unique and it will be used for all occurrences of the variable in the action side, appearing in all working memory elements and preferences that are created by the production action.

3.3.6.2 Creating Working Memory Elements

An element is created in working memory by specifying it as an action. Multiple augmentations of an object can be combined into a single action, using the same syntax as in conditions, including path notation and multi-valued attributes.

```
-->
(<s> ^block.color red
     ^thing <t1> <t2>) }
```

The action above is expanded to be:

```
-->
(<s> ^block <*b>)
(<*b> ^color red)
(<s> ^thing <t1>)
(<s> ^thing <t2>) }
```

This will add four elements to working memory with the variables replaced with whatever values they were bound to on the condition side.

Since Soar is case sensitive, different combinations of upper- and lowercase letters represent *different* constants. For example, "`red`", "`Red`", and "`RED`" are all distinct symbols in Soar. In many cases, it is prudent to choose one of uppercase or lowercase and write all constants in that case to avoid confusion (and bugs).

The constants that are used for attributes and values have a few restrictions on them:

1. There are a number of architecturally created augmentations for state and impasse objects; see Section 3.4 for a listing of these special augmentations. User-defined productions can not create or remove augmentations of states that use these attribute names.
2. Attribute names should not begin with a number if these attributes will be used in attribute-path notation.

3.3.6.3 Removing Working Memory Elements

A element is explicitly removed from working memory by following the value with a dash: `-`, also called a reject.

```
  -->
  (<s> ^block <b> -)}
```

If the removal of a working memory element removes the only link between the state and working memory elements that had the value of the removed element as an identifier, those working memory elements will be removed. This is applied recursively, so that all item that become unlinked are removed.

The reject should be used with an action that will be o-supported. If reject is attempted with I-support, the working memory element will reappear if the reject loses I-support and the element still has support.

3.3.6.4 The syntax of preferences

Below are the ten types of preferences as they can appear in the actions of a production for the selection of operators:

RHS preferences	Semantics
`(id ^operator value)`	acceptable
`(id ^operator value +)`	acceptable
`(id ^operator value !)`	require
`(id ^operator value ~)`	prohibit
`(id ^operator value -)`	reject
`(id ^operator value > value2)`	better
`(id ^operator value < value2)`	worse
`(id ^operator value >)`	best
`(id ^operator value <)`	worst
`(id ^operator value =)`	unary indifferent
`(id ^operator value = value2)`	binary indifferent

The identifier and value will always be variables, such as `(<s1> ^operator <o1> > <o2>)`.

The preference notation appears similar to the predicate tests that appear on the left-hand side of productions, but has very different meaning. Predicates cannot be used on the right-hand side of a production and you cannot restrict the bindings of variables on the right-hand side of a production. (Such restrictions can happen only in the conditions.)

Also notice that the `+` symbol is optional when specifying acceptable preferences in the actions of a production, although using this symbol will make the semantics of your productions clearer in many instances. The `+` symbol will always appear when you inspect preference memory (with the `preferences` command).

Productions are never needed to delete preferences because preferences will be retracted when the production no longer matches. Preferences should never be created by operator application rules, and they should always be created by rules that will give only I-support to their actions.

3.3.6.5 Shorthand notations for preference creation

There are a few shorthand notations allowed for the creation of operator preferences on the right-hand side of productions.

Acceptable preferences do not need to be specified with a `+` symbol. `(<s> ^operator <op1>)` is assumed to mean `(<s> ^operator <op1> +)`.

Ambiguity can easily arise when using a preference that can be either binary or unary: `> < =`. The default assumption is that if a value follows the preference, then the preference is binary. It will be unary if a carat (up-arrow), a closing parenthesis, another preference, or a comma follows it.

Below are four examples of legal, although unrealistic, actions that have the same effect.

```
(<s> ^operator <o1> <o2> + <o2> < <o1> <o3> =, <o4>)
(<s> ^operator <o1> + <o2> +
            <o2> < <o1> <o3> =, <o4> +)
(<s> ^operator <o1> <o2> <o2> < <o1> <o4> <o3> =)
(<s> ^operator <o1> ^operator <o2>
           ^operator <o2> < <o1> ^operator <o4> <o3> =)
```

Any one of those actions could be expanded to the following list of preferences:

```
(<s> ^operator <o1> +)
(<s> ^operator <o2> +)
(<s> ^operator <o2> < <o1>)
(<s> ^operator <o3> =)
(<s> ^operator <o4> +)
```

Note that structured-value notation may not be used in the actions of productions.

3.3.6.6 Righthand-side Functions

The fourth type of action that can occur in productions is called a *righthand-side function*. Righthand-side functions allow productions to create side effects other than changing working memory. The RHS functions are described below, organized by the type of side effect they have.

3.3.6.7 Stopping and pausing Soar

`halt` — Terminates Soar's execution and returns to the user prompt. A `halt` action irreversibly terminates the running of a Soar program. It should not be used if Soar is to be restarted (see the `interrupt` RHS action below.)

```
sp {
    ...
    -->
    (halt) }
```

`interrupt` — Executing this function causes Soar to stop at the end of the current phase, and return to the user prompt. This is similar to `halt`, but does not terminate the run. The run may be continued by issuing a `run` command from the user interface. The `interrupt` RHS function has the same effect as typing `ctrl-c` at the prompt, except that there is more control because it takes effect exactly at the end of the phase that fires the production.

```
sp {
    ...
    -->
    (interrupt) }
```

3.3.6.8 Text input and output

The functions `write` and `accept` are provided as production actions to do simple input and output of text in Soar. Soar applications that do extensive input and output of text should use I/O in Soar or make use of Tcl or Tk functionality. (Adding I/O and using Tcl and Tk functions are considered advanced usage and are beyond the scope of this manual. Consult the *Soar Advanced Applications Manual* for instructions.)

`write` — This function writes its arguments to the standard output. It does not automatically insert blanks, linefeeds, or carriage returns. For example, if `<o>` is bound to 4, then

```
sp {
    ...
    -->
    (write  <o> <o> <o> | x| <o> | | <o>) }
```

prints

```
444 x4 4
```

Although `write` is provided as an action in Soar, it should be used only for simple monitoring or debugging. For more extensive text output, Tcl and Tk should be used.

`crlf` — Short for "carriage return, line feed", this function can be called only within `write`. It forces a new line at its position in the `write` action.

```
sp {
    ...
    -->
    (write <x> (crlf) <y>) }
```

`accept` — Suspends Soar's execution and waits for the user to type a constant, followed by a carriage return. The result of `accept` is the constant. The accept function does not read in strings. It accepts a single constant (which may look like a string). Soar applications that make extensive use of text input should be implemented using Tcl and Tk functionality, described in the *Soar Advanced Applications Manual.*

The **`accept`** function does not work properly under the TSI (Tcl-Soar Interface), or any other Soar program that has a separate "Agent Window" instead of a Tcl or Wish Console. In this instance, users should employ the `tcl` RHS function (described on page 67) to get user input through a text widget.

```
sp {
    ...
    -->
    (<s> ^input (accept)) }
```

3.3.6.9 Mathematical functions

The expressions described in this section can be nested to any depth. For all of the functions in this section, missing or non-numeric arguments result in an error.

`+, -, *, /` — These symbols provide prefix notation mathematical functions. These symbols work similarly to C functions. They will take either integer or real-number arguments. The first three functions return an integer when all arguments are integers and otherwise return a real number, and the last two functions always return a real number. The `-` symbol is also a unary function which, given a single argument, returns the product of the argument and `-1`.

```
sp {
    ...
    -->
    (<s> ^sum (+ <x> <y>)
          ^product-sum (* (+ <v> <w>) (+ <x> <y>))
          ^big-sum (+ <x> <y> <z> 402)
          ^negative-x (- <x>))
}
```

div, mod — These symbols provide prefix notation binary mathematical functions (they each take two arguments). These symbols work similarly to C functions: They will take only integer arguments (using reals results in an error) and return an integer: `div` takes two integers and returns their integer quotient; `mod` returns their remainder.

```
sp {
    ...
    -->
    (<s> ^quotient (div <x> <y>)
         ^remainder (mod <x> <y>)) }
```

abs, atan2, sqrt, sin, cos — These symbols provide prefix notation unary mathematical functions (they each take one argument). These symbols work similarly to C functions: They will take either integer or real-number arguments. The first function (`abs`) returns an integer when its argument is an integer and otherwise returns a real number, and the last four functions always return a real number. `atan2` returns as a float in radians, the arctangent of (first_arg / second_arg). `sin` and `cos` take as arguments the angle in radians.

```
sp {
    ...
    -->
    (<s> ^abs-value (abs <x>)
         ^sqrt (sqrt <x>)) }
```

int — Converts a single symbol to an integer constant. This function expects either an integer constant, symbolic constant, or floating point constant. The symbolic constant must be a string which can be interpreted as a single integer. The floating point constant is truncated to only the integer portion. This function essentially operates as a type casting function.

For example, the expression `2 + sqrt(6)` could be printed as an integer using the following:

```
sp {
    ...
    -->
    (write (+ 2 (int sqrt(6))) ) }
```

float — Converts a single symbol to a floating point constant. This function expects either an integer constant, symbolic constant, or floating point constant. The symbolic constant must be a string which can be interpreted as a single floating point number. This function essentially operates as a type casting function.

For example, if you wanted to print out an integer expression as a floating-point number, you could do the following:

```
sp {
     ...
     -->
     (write (float (+ 2 3))) }
```

3.3.6.10 Generating and manipulating symbols

A new symbol (an identifier) is generated on the right-hand side of a production whenever a previously unbound variable is used. This section describes other ways of generating and manipulating symbols on the right-hand side.

timestamp — This function returns a symbol whose print name is a representation of the current date and time.

For example:

```
sp {
     ...
     -->
     (write (timestamp)) }
```

When this production fires, it will print out a representation of the current date and time, such as:

```
soar> run 1 e
8/1/96-15:22:49
```

make-constant-symbol — This function returns a new constant symbol guaranteed to be different from all symbols currently present in the system. With no arguments, it returns a symbol whose name starts with "`constant`". With one or more arguments, it takes those argument symbols, concatenates them, and uses that as the prefix for the new symbol. (It may also append a number to the resulting symbol, if a symbol with that prefix as its name already exists.)

```
sp {
     ...
     -->
     (<s> ^new-symbol (make-constant-symbol)) }
```

When this production fires, it will create an augmentation in working memory such as:

```
(S1 ^new-symbol constant5)
```

The production:

```
sp {
    ...
    -->
    (<s> ^new-symbol (make-constant-symbol <s> )) }
```

will create an augmentation in working memory such as:

```
(S1 ^new-symbol |S14|)
```

when it fires. The vertical bars denote that the symbol is a constant, rather than an identifier; in this example, the number 4 has been appended to the symbol S1.

This can be particularly useful when used in conjunction with the `timestamp` function; by using `timestamp` as an argument to `make-constant-symbol`, you can get a new symbol that is guaranteed to be unique. For example:

```
sp {
    ...
    -->
    (<s> ^new-symbol (make-constant-symbol (timestamp))) }
```

When this production fires, it will create an augmentation in working memory such as:

```
(S1 ^new-symbol 8/1/96-15:22:49)
```

`capitalize-symbol` — Given a symbol, this function returns a new symbol with the first character capitalized. This function is provided primarily for text output, for example, to allow the first word in a sentence to be capitalized.

```
(capitalize-symbol foo)
```

3.3.6.11 Tcl functions as RHS actions

Any Tcl command, including the Soar Interface commands described in Chapter 6, can be issued on the righthand-side of productions through the `tcl` RHS function. There are no safety nets with this function, and users are warned that they can get themselves into trouble if not careful. Users should *never* use the `tcl` RHS function to invoke `add-wme`, `remove-wme` or `sp`.

`tcl` — Concatenates each of its arguments into a string which is then sent to the agent's Tcl interpreter for evaluation. It does not automatically insert spaces between arguments. If `<o>` is bound to `x`, then

```
sp {
     ...
     -->
    (tcl |MakeANote | <o> 1) }
```

will produce the string "`MakeANote x1`" which will then be executed in the Tcl interpreter. This will call the Tcl procedure "MakeANote" (presumably defined by the user) with the single argument "`x1`".

The `tcl` RHS function returns its result as a symbolic constant so that Tcl results can be used in functional compositions. For example, the log of a number `<x>` could be printed this way:

```
sp {
     ...
     -->
     (write |The log of | <x> | is: | (tcl |expr log(|<x>|)| ))
}
```

3.3.6.12 Controlling learning

Soar's learning mechanism, called Chunking, is described in Chapter 4.

The following two functions are provided as RHS actions to assist in development of Soar programs; they are not intended to correspond to any theory of learning in Soar. This functionality is provided as a development tool, so that learning may be turned off in specific problem spaces, preventing otherwise buggy behavior.

The `dont-learn` and `force-learn` RHS actions are to be used with specific settings for the `learn` command (see page ??.) Using the `learn` command, learning may be set to one of `on`, `off`, `except`, or `only`; learning must be set to `except` for the `dont-learn` RHS action to have any effect and learning must be set to `only` for the `force-learn` RHS action to have any effect.

dont-learn — When learning is set to `except`, by default chunks can be formed in all states; the `dont-learn` RHS action will cause learning to be turned off for the specified state.

```
sp {turn-learning-off
    (state <s> ^feature 1 ^feature 2 -^feature 3)
     -->
    (dont-learn <s>) }
```

The `dont-learn` RHS action applies when `learn` is set to `-except`, and has no effect when other settings for `learn` are used.

force-learn — When learning is set to `only`, by default chunks are not formed in any state; the `force-learn` RHS action will cause learning to be turned on for the specified state.

```
sp {turn-learning-on
    (state <s> ^feature 1 ^feature 2 -^feature 3)
     -->
    (force-learn <s>) }
```

The `force-learn` RHS action applies when `learn` is set to `-only`, and has no effect when other settings for `learn` are used.

3.4 Impasses in Working Memory and in Productions

When the preferences in preference memory cannot be resolved unambiguously, Soar reaches an impasse, as described in Section 2.6:

- When Soar is unable to select a new operator (in the decision cycle), it is said to reach an operator impasse.

All impasses appear as states in working memory, where they can be tested by productions. This section describes the structure of state objects in working memory.

3.4.1 Impasses in working memory

There are four types of impasses.

Below is a short description of the four types of impasses. (This was described in more detail in Section 2.6 on page 24.)

1. *tie*: when there is a collection of equally eligible operators competing for the value of a particular attribute;

2. *conflict*: when two or more objects are better than each other, and they are not dominated by a third operator;
3. *constraint-failure*: when there are conflicting necessity preferences;
4. *no-change*: when the proposal phase runs to quiescence without suggesting a new operator.

The list below gives the seven augmentations that the architecture creates on the substate generated when an impasse is reached, and the values that each augmentation can contain:
`^type state`

`^impasse` Contains the impasse type: `tie`, `conflict`, `constraint-failure`, or `no-change`.

`^choices` Either `multiple` (for tie and conflict impasses), `constraint-failure` (for constraint-failure impasses), or `none` (for no-change impasses).

`^superstate` Contains the identifier of the state in which the impasse arose.

`^attribute` For multi-choice and constraint-failure impasses, this contains `operator`. For no-change impasses, this contains the attribute of the last decision with a value (`state` or `operator`).

`^item` For multi-choice and constraint-failure impasses, this contains all values involved in the tie, conflict, or constraint-failure. If the set of items that tie or conflict changes during the impasse, the architecture removes or adds the appropriate item augmentations without terminating the existing impasse.

`^quiescence` States are the only objects with `quiescence t`, which is an explicit statement that quiescence (exhaustion of the elaboration cycle) was reached in the superstate. If problem solving in the subgoal is contingent on quiescence having been reached, the substate should test this flag. The side-effect is that no chunk will be built if it depended on that test. See Section 4.1 on page 77 for details. This attribute can be ignored when learning is turned off.

Knowing the names of these architecturally defined attributes and their possible values will help you to write productions that test for the presence of specific types of impasses so that you can attempt to resolve the impasse in a manner appropriate to your program. Many of the default productions in the `demos/defaults` directory of the Soar distribution provide means for resolving certain types of impasses. You may wish to make use of some of all of these productions or merely use them as guides for writing your own set of productions to respond to impasses.

Examples

The following is an example of a substate that is created for a tie among three operators:

```
(S12 ^type state ^impasse tie ^choices multiple ^attribute operator
     ^superstate S3 ^item O9 O10 O11 ^quiescence t)
```

The following is an example of a substate that is created for a no-change impasse to apply an operator:

```
(S12 ^type state ^impasse no-change ^choices none ^attribute operator
     ^superstate S3 ^quiescence t)
(S3 ^operator O2)
```

3.4.2 Testing for impasses in productions

Since states appear in working memory, they may also be tested for in the conditions of productions.

There are numerous examples of this in the set of default productions (see Section 6.7.3 or Appendix ?? for more information).

For example, the following production tests for a constraint-failure impasse on the top-level state.

```
sp {default*top-goal*halt*operator*failure
   "Halt if no operator can be selected for the top goal."
   :default
   (state <s> ^superstate nil)
   (state <ss> ^impasse constraint-failure ^superstate <s>)
   -->
   (write (crlf) |No operator can be selected for top goal.| )
   (write (crlf) |Soar must halt.| )
   (halt) }
```

3.5 Soar I/O: Input and Output in Soar

Many Soar users will want their programs to interact with a real or simulated environment. For example, Soar programs could control a robot, receiving sensory *inputs* and sending command *outputs*. Soar programs might also interact with simulated environments, such as a flight simulator. The mechanisms by which Soar receives inputs and sends outputs to an external process is called *Soar I/O*.

This section describes how input and output are represented in working memory and in productions. The details of creating and registering the input and output functions for Soar are beyond the scope of this manual, but they are fully described in the Advanced Soar User's Manual. This section is provided for the sake of Soar users who will be making use of a program that has already been implemented, or for those who would simply like to understand how I/O is implemented in Soar. A simple example of Soar I/O using Tcl is provided in Section (Appendix?) ??.

3.5.1 Overview of Soar I/O

When Soar interacts with an external environment, it must make use of mechanisms that allow it to receive input from that environment and to effect changes in that environment. An external environment may be the real world or a simulation; input is usually viewed as Soar's perception and output is viewed as Soar's motor abilities.

Soar I/O is accomplished via *input functions* and *output functions*. Input functions are called at the *start* of every execution cycle, and add elements directly to specific input structures in working memory. These changes to working memory may change the set of productions that will fire or retract. Output functions are called at the *end* of every execution cycle and are processed in response to changes to specific output structures in working memory. An output function is called only if changes have been made to the output-link structures in working memory.

The structures for manipulating input and output in Soar are linked to a predefined attribute of the top-level state, called the `io` attribute. The `io` attribute has substructure to represent sensor inputs from the environment called *input links*; because these are represented in working memory, Soar productions can match against input links to respond to an external situation. Likewise, the `io` attribute has substructure to represent motor commands, called *output links*. Functions that execute motor commands in the environment use the values on the output links to determine when and how they should execute an action. Generally, input functions create and remove elements on the input link to update Soar's perception of the environment. Output functions respond to values of working memory elements that appear on Soar's output link strucure.

3.5.2 Input and output in working memory

All input and output is represented in working memory as substructure of the `io` attribute of the top-level state. By default, the architecture creates an `input-link` attribute of the `io` object and an `output-link` attribute of the io object. The values of the `input-link` and `output-link` attributes are identifiers whose augmentations are the complete set of input and output working memory elements, respectively. Some Soar systems may benefit from having multiple input and output links, or that use names which are more descriptive of the input or output function, such as `vision-input-link`, `text-input-link`, or `motor-output-link`. In addition to providing the default `io` substructure, the architecture allows users to create multiple input and output links via productions and I/O functions. Any identifiers for `io` substructure created by the user will be assigned at run time and are not guaranteed to be the same from run to run. Therefore users should always employ variables when referring to input and output links in productions.

Suppose a blocks-world task is implemented using a robot to move actual blocks around, with a camera creating input to Soar and a robotic arm executing command outputs. The camera image might be analyzed by a separate vision program; this

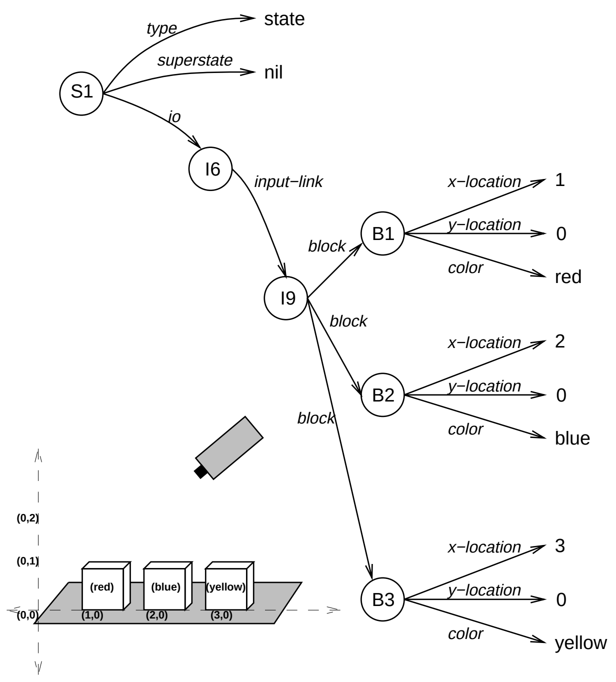

Figure 3.3: An example portion of the input link for the blocks-world task.

program could have as its output the locations of blocks on an xy plane. The Soar input function could take the output from the vision program and create the following working memory elements on the input link (all identifiers are assigned at runtime; this is just an example of possible bindings):

```
(S1 ^io I1)           [A]
(I1 ^input-link I2)   [A]
(I2 ^block B1)
(I2 ^block B2)
(I2 ^block B3)
(B1 ^x-location 1)
(B1 ^y-location 0)
(B1 ^color red)
(B2 ^x-location 2)
(B2 ^y-location 0)
(B2 ^color blue)
(B3 ^x-location 3)
(B3 ^y-location 0)
(B3 ^color yellow)
```

The '[A]' notation in the example is used to indicate the working memory elements that are created by the architecture and not by the input function. This configuration of blocks corresponds to all blocks on the table, as illustrated in the initial state in Figure 2.2.

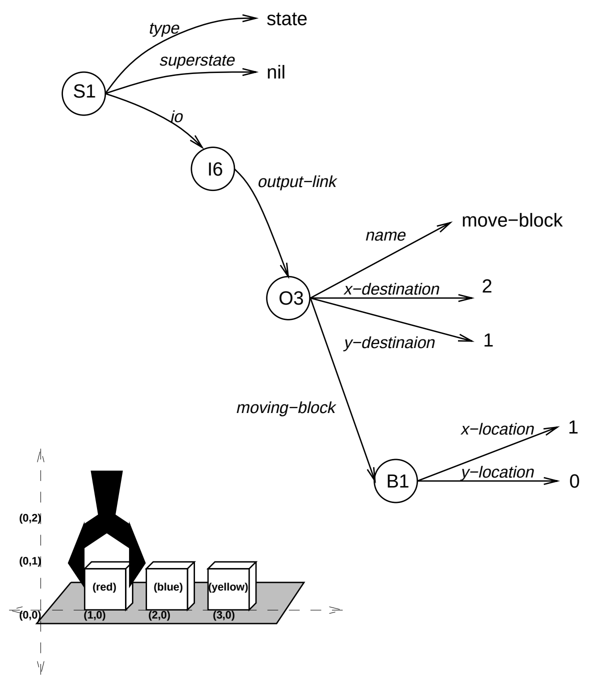

Figure 3.4: An example portion of the output link for the blocks-world task.

Then, during the Apply Phase of the execution cycle, Soar productions could respond to an operator, such as "move the red block ontop of the blue block" by creating a structure on the output link, such as:

```
(S1 ^io I1)            [A]
(I1 ^output-link I3)   [A]
(I3 ^name move-block)
(I3 ^moving-block B1)
(I3 ^x-destination 2)
(I3 ^y-destination 1)
(B1 ^x-location 1)
(B1 ^y-location 0)
(B1 ^color red)
```

The '[A]' notation is used to indicate the working memory elements that are created by the architecture and not by productions. An output function would look for specific structure in this output link and translate this into the format required by the external program that controls the robotic arm. Movement by the robotic arm would lead to changes in the vision system, which would later be reported on the input-link.

Input and output are viewed from Soar's perspective. An *input function* adds or deletes augmentations of the `input-link` providing Soar with information about some occurrence external to Soar. An *output function* responds to substructure of the `output-link` produced by production firings, and causes some occurrence external to Soar. Input and output occur through the `io` attribute of the top-level state exclusively.

The substructure of the input-link will remain in working memory until the input function that created it removes it. Thus working memory elements produced by an input function provide support for condition-matching in productions as long as the input persists in working memory, i.e. until the input function specifically removes the elements of the substructure. However, a production that tests only a single element on the input structure will result in instantiations that fire only once for each input element that matches. The instantiation will not continue to fire for each matched input element, unless the element is removed and then added again.

3.5.3 Input and output in production memory

Productions involved in *input* will test for specific attributes and values on the input-link, while productions involved in *output* will create preferences for specific attributes and values on the output link. For example, a simplified production that responds to the vision input for the blocks task might look like this:

```
sp {blocks-world*elaborate*input
    (state <s> ^io.input-link <in>)
    (<in> ^block <ib1>)
    (<ib1> ^x-location <x1> ^y-location <y1>)
    (<in> ^block {<ib2> <> <ib1>})
    (<ib2> ^x-location <x1> ^y-location {<y2> > <y1>})
    -->
    (<s> ^block <b1>)
    (<s> ^block <b2>)
    (<b1> ^x-location <x1>  ^y-location <y1> ^clear no)
    (<b2> ^x-location <x1>  ^y-location <y2> ^above <b1>)
}
```

This production "copies" two blocks and their locations directly to the top-level state. This is a generally a good idea when using input, since the input function may change the information on the link before the Soar program has finished using it. This production also adds information about the relationship between the two blocks. The variables used for the blocks on the RHS of the production are deliberately different from the variable name used for the block on the input-link in the LHS of the production. If the variable were the same, the production would create a link into the structure of the input-link, rather than copy the information. The attributes `x-location` and `y-location` are assumed to be values and not identifiers, so the same variable names may be used to do the copying.

A production that creates wmes on the output-link for the blocks task might look like this:

```
sp {blocks-world*apply*move-block*send-output-command
    (state <s> ^operator <o> ^io.output-link <out>)
    (<o> ^name move-block ^moving-block <b1> ^destination <b2>)
    (<b1> ^x-location <x1> ^y-location <y1>)
    (<b2> ^x-location <x2> ^y-location <y2>)
    -->
    (<out> ^move-block <b1>
           ^x-destination <x2> ^y-destination (+ <y2> 1))
}
```

This production would create substructure on the output-link that the output function could interpret as being a command to move the block to a new location.

Chapter 4

Learning

Chunking is Soar's learning mechanism, the sole learning mechanism in Soar. Chunking creates productions, called *chunks*, that summarize the processing required to produce the results of subgoals. When a chunk is built, it is added to production memory, where it will be matched in similar situations, avoiding the need for the subgoal. Chunks are created only when results are formed in subgoals; since most Soar programs are continuously subgoaling and returning results to higher-level states, chunks are typically created continuously as Soar runs.

This chapter begins with a discussion of when chunks are built (Section 4.1 below), followed by a detailed discussion of how Soar determines a chunk's conditions and actions (Section 4.2). Sections 4.3 through 4.4 examine the construction of chunks in further detail. Section 4.5 explains how and why chunks are prevented from matching with the WME's that led to their creation. Section 4.6 reviews the problem of overgeneral chunks.

4.1 Chunk Creation

Several factors govern when chunks are built. Soar chunks the results of every subgoal, *unless* one of the following conditions is true:

1. Learning is `off`. (See Section 6.4.3 on page 143 for details of `learn` used to turn learning off.)

   Learning can be set to `on` or `off`. When `learn` is `on` chunks are built. When `learn` is `off`, chunks are not built.

2. Learning is set to `bottom-up` and a chunk has already been built for a subgoal of the state that generated the results. (See Section **??** on page **??** for details of `learn` used to set learning to bottom-up.)

   With bottom-up learning, chunks are learned only in states in which no subgoal has yet generated a chunk. In this mode, chunks are learned only for the "bottom" of the subgoal hierarchy and not the intermediate levels. With experience, the subgoals at the bottom will be replaced by the chunks, allowing higher level subgoals to be chunked.[1]

3. The chunk duplicates a production or chunk already in production memory. In some rare cases, a duplicate production will not be detected because the order of the conditions or actions is not the same as an existing production.

4. The augmentation, `^quiescence t`, of the substate that produced the result is backtraced through.

   This mechanism is motivated by the *chunking from exhaustion* problem, where the results of a subgoal are dependent on the exhaustion of alternatives (see Section 4.6 on page 82). If this substate augmentation is encountered when determining the conditions of a chunk, then no chunk will be built for the currently considered action. This is recursive, so that if an un-chunked result is relevant to a second result, no chunk will be built for the second result. This does not prevent the creation of a chunk that would include `^quiescence t` as a condition.

5. Learning has been temporarily turned off via a call to the `dont-learn` production action (described on page 67 in Section 3.3.6.12).

   This capability is provided for debugging and system development, and it is not part of the theory of Soar.

If a result is to be chunked, Soar builds the chunk *as soon as the result is created*, rather than waiting until subgoal termination.

4.2 Determining Conditions and Actions

Chunking is an experience-based learning mechanism that summarizes as productions the problem solving that occurs within a state. In order to maintain a history of the processing to be used for chunking, Soar builds a *trace* of the productions that fire in the subgoals. This section describes how the relevant actions are determined, how information is stored in a trace, and finally, how the trace and the actions together determine the conditions for the chunk.

In order for the chunk to apply at the appropriate time, its conditions must test exactly those working memory elements that were necessary to produce the results of the subgoal. Soar computes a chunk's conditions based on the productions that fire in the subgoal, beginning with the results of the subgoal, and then *backtracing* through the productions that created each result. It recursively backtraces through the working memory elements that matched the conditions of the productions, finding

[1]For some tasks, bottom-up chunking facilitates modelling power-law speedups, although its long-term theoretical status is problematic.

the actions that led to the WME's creation, etc., until conditions are found that test elements that are linked to a superstate.

4.2.1 Determining a chunk's actions

A chunk's actions are built from the results of a subgoal. A *result* is any working memory element created in the substate that is linked to a superstate. A working memory element is linked if its identifier is either the value of a superstate WME, or the value of an augmentation for an object that is linked to a superstate.

The results produced by a single production firing are the basis for creating the actions of a chunk. A new result can lead to other results by linking a superstate to a WME in the substate. This WME may in turn link other WMEs in the substate to the superstate, making them results. Therefore, the creation of a single WME that is linked to a superstate can lead to the creation of a large number of results. All of the newly created results become the basis of the chunk's actions.

4.2.2 Tracing the creation and reference of working memory elements

Soar automatically maintains information on the creation of each working memory element in every state. When a production fires, a trace of the production is saved with the appropriate state. A *trace* is a list of the working memory elements matched by the production's conditions, together with the actions created by the production. The appropriate state is the most recently created state (i.e., the state *lowest* in the subgoal hierarchy) that occurs in the production's matched working memory elements.

Recall that when a subgoal is created, the ^item augmentation lists all values that lead to the impasse. Chunking is complicated by the fact that the `^item` augmentation of the substate is created by the architecture and not by productions. Backtracing cannot determine the cause of these substate augmentations in the same way as other working memory elements. To overcome this, Soar maps these augmentations onto the acceptable preferences for the operators in the `^item` augmentations.

Negated conditions

Negated conditions are included in a trace in the following way: when a production fires, its negated conditions are fully instantiated with its variables' appropriate values. This instantiation is based on the working memory elements that matched the production's positive conditions. If the variable is not used in any positive conditions, such as in a conjunctive negation, a dummy variable is used that will later become a variable in a chunk.

If the identifier used to instantiate a negated condition's identifier field is linked to the superstate, then the instantiated negated condition is added to the trace as a negated condition. In all other cases, the negated condition is ignored because the system cannot determine why a working memory element *was not* produced in the subgoal and thus allowed the production to fire. Ignoring these negations of conditions internal to the subgoal may lead to overgeneralization in chunking (see Section 4.6 on page 82).

4.2.3 Determining a chunk's conditions

The conditions of a chunk are determined by a dependency analysis of production traces — a process called *backtracing.* For each instantiated production that creates a subgoal result, backtracing examines the production trace to determine which working memory elements were matched. If a matched working memory element is linked to a superstate, it is included in the chunk's conditions. If it is not linked to a superstate, then backtracing recursively examines the trace of the production that created the working memory element. Thus, backtracing begins with a subgoal result, traces backwards through all working memory elements that were used to produce that result, and collects all of the working memory elements that are linked to a superstate. This method ignores when the working memory elements were created, thus allowing the conditions of one chunk to test the results of a chunk learned earlier in the subgoal. The user can observe the backtracing process by setting setting backtracing on, using the watch command: `watch backtracing -on` (see Section 6.3.8 on page 132). This prints out a trace of the conditions as they are collected.

Certain productions do not participate in backtracing. If a production creates only a `reject` preference or a desirability preference (`better`, `worse`, `indifferent`, or `parallel`), then neither the preference nor the objects that led to its creation will be included in the chunk. (The exception to this is that if the desirability or `reject` preference is a *result* of a subgoal, it will be in the chunk's actions.) Desirability and reject preferences should be used only as search control for choosing between legal alternatives and should not be used to guarantee the correctness of the problem solving. The argument is that such preferences should affect only the *efficiency* and not the *correctness* of problem solving, and therefore are not necessary to produce the results. Necessity preferences (`require` or `prohibit`) should be used to enforce the correctness of problem solving; the productions that create these preferences will be included in backtracing.

Given that results can be created at any point during a subgoal, it is possible for one result to be relevant to another result. Whether or not the first result is included in the chunk for the second result depends on the links that were used to match the first result in the subgoal. If the elements are linked to the superstate, they are included as conditions. If the elements are not linked to the superstate, then the result is traced through. In some cases, there may be more than one set of links, so it is possible for a result to be both backtraced through, and included as a condition.

4.3 Variablizing Identifiers

Chunks are constructed by examining the traces, which include working memory elements and operator preferences. To achieve any useful generality in chunks, identifiers of actual objects must be replaced by variables when the chunk is created; otherwise chunks will only ever fire when the exact same objects are matched. However, a constant value is never variablized; the actual value always appears directly in the chunk.

When a chunk is built, all occurrences of the same identifier are replaced with the same variable. This can lead to an overspecific chunk, when two variables are forced to be the same in the chunk, even though distinct variables in the original productions just happened to match the same identifier.

A chunk's conditions are also constrained by any not-equal (`<>`) tests for pairs of indentifiers used in the conditions of productions that are included in the chunk. These tests are saved in the production traces and then added in to the chunk.

4.4 Ordering Conditions

Since the efficiency of the Rete matcher [**?**] depends heavily upon the order of a production's conditions, the chunking mechanism attempts to write the chunk's conditions in the most favorable order. At each stage, the condition-ordering algorithm tries to determine which eligible condition, if placed next, will lead to the fewest number of partial instantiations when the chunk is matched. A condition that matches an object with a multi-valued attribute will lead to multiple partial instantiations, so it is generally more efficient to place these conditions later in the ordering.

This is the same process that internally reorders the conditions in user-defined productions, as mentioned briefly in Section 2.3.1.

4.5 Inhibition of Chunks

When a chunk is built, it may be able to match immediately with the same working memory elements that participated in its creation. If the production's actions include preferences for new operators, the production would immediately fire and create a preference for a new operator, which duplicates the operator preference that was the original result of the subgoal. To prevent this, *inhibition* is used. This means that each production that is built during chunking is considered to have already fired with the instantiation of the exact set of working memory elements used to create it. This does not prevent a newly learned chunk from matching other working memory elements that are present and firing with those values.

4.6 Problems that May Arise with Chunking

One of the weaknesses of Soar is that chunking can create overgeneral productions that apply in inappropriate situations, or overspecific productions that will never fire. These problems arise when chunking cannot accurately summarize the processing that led to the creation of a result. Below is a description of three known problems in chunking.

4.6.1 Using search control to determine correctness

Overgeneral chunks can be created if a result of problem solving in a subgoal is dependent on search-control knowledge. Recall that desirability preferences, such as `better`, `best`, and `worst`, are not included in the traces of problem solving used in chunking (Section 4.2 on page 78). In theory, these preferences do not affect the validity of search. In practice, however, a Soar program can be written so that search control *does* affect the correctness of search. Here are two examples:

1. Some of the tests for correctness of a result are included in productions that prefer operators that will produce correct results. The system will work correctly only when those productions are loaded.
2. An operator is given a worst preference, indicating that it should be used only when all other options have been exhausted. Because of the semantics of worst, this operator will be selected after all other operators; however, if this operator then produces a result that is dependent on the operator occurring after all others, this fact will not be captured in the conditions of the chunk.

In both of these cases, part of the test for producing a result is *implicit* in search control productions. This move allows the explicit state test to be simpler because any state to which the test is applied is guaranteed to satisfy some of the requirements for success. However, chunks created in such a problem space will be overgeneral because the implicit parts of the state test do not appear as conditions.

Solution: To avoid this problem, necessity preferences (`require` and `prohibit`) should be used whenever a control decision is being made that also incorporates goal-attainment knowledge. The necessity preferences are included in the backtrace by chunking, thereby avoiding overgenerality.

4.6.2 Testing for local negated conditions

Overgeneral chunks can be created when negated conditions test for the absence of a working memory element that, if it existed, would be local to the substate. Chunking has no mechanism for determining *why* a given working memory element does not exist, and thus a condition that occurred in a production in the subgoal is not included in the chunk. For example, if a production tests for the absence of a local flag, and that flag is copied down to the substate from a superstate, then the chunk should include a test that the flag in the superstate does not exist. Unfortunately, it is computationally expensive to determine why a given working memory element does not exist. Chunking only includes negated tests if they test for the absence of superstate working memory elements.

Solution: To avoid using negated conditions for local data, the local data can be made a result by attaching it to the superstate. This increases the number of chunks learned, but a negated condition for the superstate can be used that leads to correct chunks.

4.6.3 Testing for the substate

Overgeneral chunks can be created if a result of a subgoal is dependent on the creation of an impasse within the substate. For example, processing in a subgoal may consist of exhaustively applying all the operators in the problem space. If so, then a convenient way to recognize that all operators have applied and processing is complete is to wait for a state no-change impasse to occur. When the impasse occurs, a production can test for the resulting substate and create a result for the original subgoal. This form of state test builds overgeneral chunks because no pre-existing structure is relevant to the result that terminates the subgoal. The result is dependent only on the existence of the substate within a substate.

Solution: The current solution to this problem is to allow the problem solving to signal the architecture that the test for a substate is being made. The signal used by Soar is a test for the `^quiescence t` augmentation of the subgoal. The chunking mechanism recognizes this test and does not build a chunk when it is found in a backtrace of a subgoal. The history of this test is maintained, so that if the result of the substate is then used to produce further results for a superstate, no higher chunks will be built. However, if the result is used as search control (it is a desirability preference), then it does not prevent the creation of chunks because the original result is not included in the backtrace. If the `^quiescence t` being tested is connected to a superstate, it will not inhibit chunking and it will be included in the conditions of the chunk.

Chapter 5

Soar and Tcl: The Soar Application Interface

This chapter provides a brief introduction to the Soar Application Interface and how Soar interacts with Tcl, the Tool Command Language. It also discusses the graphical interface that is bundled with Soar, called the Tcl-Soar Interface, or TSI. This chapter is not intended to be a full discourse on Tcl or how to develop Soar simulation environments. The reader is referred to *Practical Programming in Tcl and Tk* by Brent Welch and *The Soar Advanced Applications Manual* for more detailed information.

Chapter 6

The Soar User Interface

This chapter describes the set of user interface commands for Soar. A few core Tcl commands are also included in sections 6.5 and 6.7 for completeness. All commands and examples are presented as if they are being entered at the Soar command prompt, but they could just as easily be placed in a file and loaded into Soar using the Tcl `source` command. Make sure you have read Chapter 1.5, concerning the integration of Soar and Tcl.

This chapter is organized into 7 sections:

1. Basic Commands for Running Soar
2. Examining Memory
3. Configuring Trace Information and Debugging
4. Configuring Soar's Run-Time Parameters
5. File System I/O Commands
6. Soar I/O commands
7. Miscellaneous Commands

Each section begins with a summary description of the commands covered in that section, including the role of the command and its importance to the user. Commands are then described fully, in alphabetical order.

Throughout this chapter, each function description includes a specification of its syntax and an example of its use.

For a concise overview of the Soar interface functions, see the Function Summary and Index on page **??**. This index is intended to be a quick reference into the commands described in this chapter.

Notation

The notation used to denote the syntax for each user-interface command follows some general conventions:

- The command name itself is given in a **`bold`** font.
- Optional command arguments are enclosed within square brackets, `[` and `]`.
- A vertical bar, `|`, separates alternatives.
- Curly braces, {}, are used to group arguments when at least one argument from the set is required.
- Variable arguments, such as a file name or an integer, are in an italic font, for example, *`filename`*.
- The commandline prompt that is printed by Soar, is normally the agent name, followed by '`>`'. In the examples in this manual, we use "`soar>`".
- Some of the command specifications are too long to fit on one line. In such cases, a backslash, \, is used to continue the command on the next line. The backslash is not part of the command itself; it is the Tcl syntax for continuing a long command on multiple lines.
- Following Tcl syntax, comments in the examples are preceded by a '`#`', and in-line comments are preceded by '`;#`'.

For many commands, there is some flexibility in the order in which the arguments may be given. (See the online help for each command for more information.) We have not incorporated this flexible ordering into the syntax specified for each command because doing so complicates the specification of the command. When the order of arguments will affect the output produced by a command, the reader will be alerted.

6.1 Basic Commands for Running Soar

This section describes the commands used to start, run and stop a Soar program; to invoke on-line help information; and to create and delete Soar productions. The specific commands described in this section are:

Summary

d - Run the Soar program for one decision cycle.

e - Run the Soar program for one elaboration cycle.

excise - Delete Soar productions from production memory.

exit - Terminate Soar and return to the operating system.

help - Provide formatted, on-line information about Soar commands.

init-soar - Reinitialize Soar so a program can be rerun from scratch.

quit - Close log file, terminate Soar, and return user to the operating system.

run - Begin Soar's execution cycle.

sp - Create a production and add it to production memory.

stop-soar - Interrupt a running Soar program.

These commands are all frequently used anytime Soar is run.

6.1.1 d [*n*]

The `d` alias is a shorthand for "`run d`". If a numeric argument is specified, `d` will cause Soar to run for *n* decision cycles. The default value of `n` is 1, so that if no argument is specified, Soar will run 1 decision cycle.

Example

```
soar> d 5    #; run for 5 decision cycles
```

6.1.2 e [*n*]

The `e` alias is a shorthand for "`run e`". If a numeric argument is specified, `e` will cause Soar to run to the end of the *n*th elaboration cycle. The default value of `n` is 1, so that if no argument is specified, Soar will complete 1 elaboration cycle.

Example

```
soar> e 3     #;  run for 3 elaboration cycles
```

Recall that an elaboration cycle is the sequence of firings and retractions followed by working memory changes, which occur during the Propose and Apply phases. The Input, Decide, and Output phases each count as one elaboration cycle for the purposes of this command.

6.1.3 excise *prod-name* |-all|-chunks|-default|-user|-task

The `excise` command removes productions from production memory. A pound sign (`#`) is printed for every production excised. The command must be called with either a specific production name or with a flag that indicates a particular group of productions. The optional flags are described in the table below:

argument	productions removed
prod-name	Excise only the named production
`-all`	Excise all productions; also do an `init-soar`
`-chunks`	Excise all chunks and justifications
`-default`	Excise all default productions
`-task`	Excise all non-default productions (user, chunks, justs); also do an `init-soar`
`-user`	Excise all user productions; chunks, justifications and default productions remain

The `excise` command must be called with at least one argument. Note that the `-task` and `-all` arguments also cause Soar to do an `init-soar`, described on page 92.

Example

```
soar> excise blocks-world*propose*initial-state
#

soar> excise -all
#####################################################init-soar done
```

Notes

The `excise` command prints a pound sign for each production excised. The printing of pound signs may be turned off by using the `watch` command; this is described in Section 6.3.8.

6.1.4 `exit`

The `exit` command terminates the Soar process. `exit` is actually a core Tcl command that terminates the process that executed the `exit` command. If an integer value is supplied as the argument to `exit`, then that becomes the exit status of the process. The `exit` command does not invoke any callbacks or close open log files before exiting; therefore it is recommended that users terminate Soar by using the `quit` command, described later in this section.

Example

```
soar> exit
```

(process returned to operating system.)

6.1.5 **help**, **?** [-all | *command-name* | -usage *command-name*]

The `help` command and its alias, `?`, provide online reference information about Soar commands. Only the `help` command is referred to in this chapter, but `help` and `?` may be used interchangably.

When called with no arguments, `help` will provide a brief synopsis of some of the most frequently used Soar commands.

When called with the optional argument `-all`, `help` will print a listing of all of the commands in Soar, followed by a listing of the core Tcl commands for which information is available. (Or it may print a hint as to where to look for the on-line Tcl documentation if Soar can't display it for some reason.)

When `help` is called with a specific *command-name*, `help` will display a manual page for that command. Soar will search for Soar, Tcl, and, on Unix the system man pages, and print the information for the first match to *command-name* that it finds. When Soar is being run under the Tcl-Soar Interface (TSI), the requested help page will be displayed in an independent scrollable window. When Soar is run without the TSI, the help page will temporarily overwrite the text on the window in which you are running Soar; to make the help page go away, type `q`; to scroll down through the manual page, hit the space bar.

Soar's help facility is able to do command completion, so as long as a unique substring of the *command-name* is specified, Soar will find and display the help page. If the substring is not unique, a message listing the possible choices is printed.

When called with `-usage` and a *command-name*, `help` will display only a brief description of the syntax for that command.

Example

```
soar> help

Commonly used Soar commands:
  cd           Change to another directory
  excise       Remove productions from Soar's memory
  init-soar    Reinitialize Soar
  learn        Turn learning on and off
  log          Save a Soar session to a file
  matches      Print info about the match set and partial matches
  preferences  Display items in preference memory
  print        Display productions or working memory elements
  pwd          Display the current working directory
  quit         Exit Soar
  run          Run the Soar decision cycle
  soarnews     Display information such as where to report bugs
  source       Load a file into Soar
  sp           Define a Soar production
  version       Display the version number of Soar
  watch         Set the amount of information displayed as Soar runs
  wmes          alias to display working memory elements

For a list of ALL available help topics, type "help -all"
For help on a specific command, type "help" followed by the command name.

soar>help in
Ambiguous Help topic: input-period internal-symbols interp
indifferent-selection incr inds info init-soar
```

Notes

Although we've tried to make the `help` command robust, it may function differently in different operating systems. For the most reliable and consistent output from `help`, always run Soar with the TSI.

If you have problems accessing online help, contact your local Soar administrator or send email to soar-help@umich.edu.

6.1.6 `init-soar`

The `init-soar` command re-initializes Soar. It empties working memory, wiping out the subgoal stack, and resets all runtime statistics. The firing counts for all productions is reset to zero. The `init-soar` command allows a Soar program that has been `halt`ed to be reset and start its execution from the beginning.

`init-soar` does not remove any productions from production memory; to do this, use the `excise` command. Note however, that all justifications will be removed because they will no longer be supported.

Example

```
soar> init-soar
```

6.1.7 `quit`

The `quit` command terminates Soar and returns the user to the operating system. It does not accept any arguments. It closes any open log files and invokes any system termination callbacks that are registered (see the `monitor` command). Using `quit` instead of the Tcl `exit` command allows programs to invoke procedures prior to termination. Once all callbacks have been processed, `quit` invokes the Tcl `exit` command.

Example

```
soar> quit
Exiting Soar ...
```

6.1.8 run [*n*|forever] [*unit*] [-self]

The `run` command starts the Soar execution cycle or continues any execution that was temporarily stopped. The default behavior of `run`, with no arguments, is to cause Soar to execute until it is halted or interrupted by an action of a production, or until an external interrupt is issued by the user.

The `run` command can also specify that Soar should run only for a specific number of Soar cycles or phases (which may also be prematurely stopped by a production action or a control-C). This is helpful for debugging sessions, where users may want to pay careful attention to the specific productions that are firing and retracting, perhaps in conjunction with changing the `watch` settings, described in Section 6.3.8 on page 132.

If there are multiple Soar agents that exist in the same Soar process, then issuing a `run` command in any agent will cause all agents to run with the same set of parameters, unless the flag `-self` is specified, in which case only that agent will execute.

`run` followed by the keyword '`forever`' is the same as the default behavior with no arguments: Soar executes until stopped by an external interrupt (eg: Ctrl-C), a RHS `interrupt` action, or a RHS `halt` action.

The `run` command takes two optional arguments: an integer, *n*, which specifies how many units to run; and a *unit* flag indicating what steps or increments to use.

The following is a list of available units and their meaning. In each case, if *n* is not specified as an argument to `run`, but a *unit* flag is specified, *n* defaults to 1.

unit	effect on running
`p`	run for *n* phases (a phase is either input, propose, decide, apply or output)
`e`	run Soar for *n* elaboration cycles (here input, decide and output phases are each counted as an elaboration cycle)
`d`	run for *n* decision cycles (this is the default unit for running Soar, if `run` is called with a number, but not a letter)
`s`	run until the *n*th time a state is selected
`o`	run until the *n*th time an operator is selected
`out`	run until the *n*th time output is generated on the output-link (or a maximum of 15 decision cycles with no output.)
`<s>`	run until current level of subgoaling has terminated
`<ss>`	run until superstate's level of subgoaling has terminated
`<sss>`	run until supersuperstate's level of subgoaling has terminated
`<o>`	run Soar until the nth time an operator is selected *at this level of subgoaling*, or until the current level of subgoaling is terminated
`<so>`	run Soar until the nth time a superoperator is selected, or until that level of subgoaling is terminated
`<sso>`	run Soar until the nth time a supersuperoperator is selected or until that level of subgoaling is terminated

The number, *n*, and *unit* arguments to the `run` command can appear in either order; for example, "`run 1 d`" and "`run d 1`" are equivalent. Also note that if you call `run` with a number but not a letter, that `d` is assumed and Soar will run for `n` decision cycles. Similarly, if you call `run` with a letter but not a number, *n=1* is assumed.

Note that the `<s>` argument is different from the `s` argument, and the `<o>` argument is different from the `o` argument (with and without the angle braces that signify a variable). For example, with the angle braces, operator selections that take place in subgoals will not be counted; *without* the angle braces, operator selections are counted without regard to the level of subgoaling.

The units that refer to operator variables (`<o>`, `<so>`, and `<sso>`) will not work unless a value is already in place for that operator. For example, when Soar is first started, none of these units will work because there is no current operator, much less a current superoperator or current supersuperoperator.

Examples

```
run         ;# run until halted by a control-C or a production action
run 5 d     ;# run for 5 decision cycles
run d 3     ;# run for 3 decision cycles
run 2       ;# run for 2 decision cycles
run p       ;# run for 1 phase
run 3 <o>   ;# run for 3 operator selections at this level of subgoaling
```

Notes

If Soar has been stopped due to a `halt` action, an `init-soar` command must be issued before Soar can be restarted with the `run` command.

In Soar 8, the execution cycle no longer ends after the decision phase, as it did in Soar 7. Therefore users who wish to examine memory or print the match set *after* the decision phase, but *before* any firings or retractions, must either step through the execution by `phases`, or set a `monitor` to generate a "`stop-soar -self`" after each decision phase. See the sections on the `stop-soar` (page 97) and `monitor` (page 122) commands, and the *Soar Advanced Applications Manual* for more information on stopping after the decision phase.

There are two predefined aliases for the `run` command. The `d` command alias will run Soar by decision cycles (see Section 6.1.1 on page 89), and the `e` command alias will run Soar by elaboration cycles (see Section 6.1.2 on page 89). You may, of course, define your own aliases if you find an increment that is particularly useful for your debugging session. (See the `alias` command on page 161.)

6.1.9 sp { *production-body* }

The `sp` command creates a new production and loads it into production memory. If the production name is the same as an existing production, the old production will be overwritten (excised). This section provides only a brief overview of the `sp` command. The syntax of productions is described completely in Section 3.3 on page 40.

Syntax

Syntactically, each production consists of the symbol `sp`, followed by: an opening curly brace, {, the production's name, the production's conditions, the symbol `-->`, the production's actions, and a closing curly brace, }:

```
sp { production-name
    CONDITIONS
    -->
    ACTIONS
}
```

An optional comment string can be included following the name of the production. This string is set off with double quotes when curly braces are used to define the production:

```
sp { production-name
    "optional documentation string"
    CONDITIONS
    -->
    ACTIONS
}
```

One or more optional flags may be used to force the production to be considered a certain type (regardless of what would otherwise be true).

sp { *production-name*

```
    "optional documentation string"
    flag*
    CONDITIONS
    -->
    ACTIONS
}
```

The optional flags are as follows:

:o-support specifies that all the RHS actions are to be given O-support when the production fires.

:i-support specifies that all the RHS actions are only to be given I-support when the production fires.

:default specifies that this production is a default production. (This matters for the `excise -task` command).

:chunk specifies that this production is a chunk. (This matters for the `explain-backtraces` command.)

Multiple flags may be used, but not both of `o-support` and `i-support`.

Although you could force your productions to provide O-support or I-support by using these commands — regardless of the structure of the conditions and actions of the production — this is not proper coding style. The `o-support` and `i-support` flags are included to help with debugging, but should not be used in a standard Soar program.

Example

```
sp {blocks*create-problem-space
    "This creates the top-level space"
    (state <s> ^superstate nil)
    -->
    (<s> ^name solve-blocks-world ^problem-space <p>)
    (<p> ^name blocks-world)
}
```

Notes

The syntax of the `sp` command is explained in Section 3.3, and a grammar appears in Appendix ??. Consult these two sections for additional details.

The syntax of productions changes when Tcl variables appear in productions. This is described briefly in Section ??. Since using Tcl in productions is considered an advanced usage, consult the *Soar Advanced Applications Manual* for more details.

The `sp` command prints one asterisk for each production successfully loaded into production memory and one pound sign # for each production redefined (excised and loaded). The printing of asterisks and pound signs may be turned off by using the `watch` command; this is described in Section 6.3.8 on 132.

6.1.10 **stop-soar** `[-self [`*reason-string*`]]`

The `stop-soar` command stops any running Soar agents. It sets a flag in the Soar kernel so that Soar will stop running at a "safe" point and return control to the user. This operates exactly as if the user had issued a control-C (SIGINT) interrupt to the Soar process, or Soar had issued a RHS `interrupt` action: It causes all currently running Soar interpreters to stop.

If the argument `-self` is specified, only the Soar agent that issued the command is interrupted; all other agents continue running. An optional *reason-string* following `-self` will be printed when Soar is stopped, to indicate why it was stopped. If left blank, no message will be printed when Soar is stopped.

A common use of this command is as an action resulting from a button press on a Graphical User Interface (GUI), or as a `monitor` to be executed at a specific Soar Event. For example, a user may wish to examine an agent's "matches" after the Soar Decision Phase. In order to do this in Soar 8, the user must register a monitor, or callback, to issue the `"stop-soar -self"` command for the after-decision-phase-cycle event.

Example

```
soar> monitor -add {stop-soar -self "after decision phase"} \
                      after-decision-phase-cycle
```

The above example shows how to stop Soar8 after the decision phase using "stop-soar -self" in a `monitor` to stop after the decision phase so that memory can be examined.

Notes

If the graphical interface doesn't periodically do a Tcl "update" command, then it may not be possible to interrupt a Soar agent from the command line.

When using the TSI, the `stop-soar` command is redefined by the TSI to also pause the interface itself.

6.2 Examining Memory

This section describes the commands used to inspect production memory, working memory, and preference memory; to see what productions will match and fire in the next Propose or Apply phase; and to examine the goal dependency set. These commands are particularly useful when running or debugging Soar, as they let users see what Soar is "thinking." The specific commands described in this section are:

Summary

gds_print - Print the WMEs in the goal dependency set for each goal.

internal-symbols - Print information about the Soar symbol table.

matches - Print information about the match set and partial matches.

memories - Print memory usage for production matches.

preferences - Examine items in preference memory.

print - Print items in working memory or production memory.

production-find - Find productions that contain a given pattern.

default-wme-depth - Set the level of detail used to print WME's.

wmes - An alias for the print command; prints items in working memory.

Of these commands, `print` is the most often used (and the most complex) followed by `matches` and `memories`. `preferences` is used to examine which candidate operators have been proposed. `production-find` is especially useful when the number of productions loaded is high. `gds_print` is useful for examining the goal dependecy set when subgoals seem to be disappearing unexpectedly. `default-wme-depth` and `wmes` are both related to the `print` command. `internal-symbols` is not often used but is helpful when debugging Soar extensions or trying to locate memory leaks.

6.2.1 `gds_print`

This is a debugging command for examining the Goal Dependency Set for each goal in the stack. First it steps through all the working memory elements in the rete, looking for any that are included in *any* goal dependency set, and prints each one. Then it also lists each goal in the stack and prints the wmes in the goal dependency set for that particular goal. This command is useful when trying to determine why subgoals are disappear unexpectedly: often something has changed in the goal dependency set, causing a subgoal to be regenerated prior to producing a result.

Example

```
soar> gds_print
******************** Current GDS ************************
stepping thru all wmes in rete, looking for any that are in a gds...
  For Goal  S2  (128: S2 ^superstate S1)
  For Goal  S2  (131: S2 ^choices multiple)
  For Goal  S2  (124: S1 ^operator O1 +)
  For Goal  S2  (125: S1 ^operator O2 +)
  For Goal  S2  (126: S1 ^operator O3 +)
  For Goal  S2  (9: S1 ^desired D1)
  For Goal  S2  (10: S1 ^problem-space P1)
  For Goal  S2  (12: P1 ^default-state-copy yes)
  For Goal  S2  (13: P1 ^default-operator-copy no)
*******************************************************
  For Goal  S1  : No GDS for this goal.
  For Goal  S2
                (13: P1 ^default-operator-copy no)
                (12: P1 ^default-state-copy yes)
                (9: S1 ^desired D1)
                (10: S1 ^problem-space P1)
                (128: S2 ^superstate S1)
                (131: S2 ^choices multiple)
                (124: S1 ^operator O1 +)
                (125: S1 ^operator O2 +)
                (126: S1 ^operator O3 +)
  For Goal  S3  : No GDS for this goal.
*******************************************************
soar>
```

Notes

This command is quite inefficient and can be very slow.

6.2.2 internal-symbols

The `internal-symbols` command prints information about the Soar symbol table. Such information is typically only useful for users attempting to debug Soar by locating memory leaks or examining I/O structure.

Example

```
soar> internal-symbols

--- Symbolic Constants: ---
```

```
operator
accept
evaluate-object
problem-space
sqrt
interrupt
mod
goal
io

  (...additional symbols deleted for brevity...)

--- Integer Constants: ---

--- Floating-Point Constants: ---

--- Identifiers: ---

--- Variables: ---
<o>
<sso>
<to>
<ss>
<ts>
<so>
<sss>
<s>
soar>
```

6.2.3 **matches** [-assertions|-retractions][0|1|2] \ [-names|-timetags|-wmes] **matches** [*prodname*] [0|1|2|-count|-timetags|-wmes]

The `matches` command prints a list of productions that have instantiations in the match set, i.e., those productions that will retract or fire in the next Propose or Apply phase. It also will print partial match information for a single, named production.

The optional arguments to the `matches` command are described in the following table. The numeric and named flags are redundant; either the number or name may be used to specify the level of detail.

matches		Effect on output
-assertions		Print match set information only about assertions
-retractions		Print match set information only about retractions
prodname		Print information about partial matches for the named production.
0	`-names`	For the match set, print only the names of the productions that are about to fire or retract (default)
0	`-count`	For named productions, print only partial match counts (default)
1	`-timetags`	Also print the timetags of the WME's that match the productions
2	`-wmes`	Also print the WME's that match the productions

6.2.3.1 Printing the match set

When printing the match set (i.e., no production name is specified), the default action prints only the names of the productions which are about to fire or retract. If there are multiple instantiations of a production, the total number of instantiations of that production is printed after the production name, unless `-timetags|1` or `-wmes|2` are specified, in which case each instantiation is printed on a separate line.

When printing the match set, the `-assertions` and `-retractions` arguments may be specified to restrict the output to print only the assertions or retractions.

Example

```
soar> matches
Assertions:
  blocks-world*select*move-block*indifferent (2)
Retractions:
  blocks-world*select*move-block*indifferent (5)

soar> matches 1
Assertions:
  blocks-world*select*move-block*indifferent 68 62
  blocks-world*select*move-block*indifferent 69 65
Retractions:
  blocks-world*select*move-block*indifferent
 57 49
  blocks-world*select*move-block*indifferent
 56 46
  blocks-world*select*move-block*indifferent
 55 43
  blocks-world*select*move-block*indifferent
 53 37
```

```
  blocks-world*select*move-block*indifferent
 52 34

soar> matches 2
Assertions:
  blocks-world*select*move-block*indifferent (68: S1 ^operator O8 +)
 (62: O8 ^name move-block)

  blocks-world*select*move-block*indifferent (69: S1 ^operator O9 +)
 (65: O9 ^name move-block)

Retractions:
  blocks-world*select*move-block*indifferent
 (57: S1 ^operator O6 +)
 (49: O6 ^name move-block)

  blocks-world*select*move-block*indifferent
 (56: S1 ^operator O5 +)
 (46: O5 ^name move-block)

  blocks-world*select*move-block*indifferent
 (55: S1 ^operator O4 +)
 (43: O4 ^name move-block)

  blocks-world*select*move-block*indifferent
 (53: S1 ^operator O2 +)
 (37: O2 ^name move-block)

  blocks-world*select*move-block*indifferent
 (52: S1 ^operator O1 +)
 (34: O1 ^name move-block)
```

6.2.3.2 Printing partial matches for productions

In addition to printing the current match set, the `matches` command can be used to print information about partial matches for a named production. In this case, the conditions of the production are listed, each preceded by the number of currently active matches for that condition. If a condition is negated, it is preceded by a minus sign (`-`). The pointer `>>>>` before a condition indicates that this is the first condition that failed to match.

When printing partial matches, the default action is to print only the counts of the number of WME's that match, and is a handy tool for determining which condition failed to match for a production that you thought should have fired. At levels `1` and

2 (or `-timetags` and `-wmes` arguments) the `matches` command displays the WME's immediately after the first condition that failed to match — temporarily interrupting the printing of the production conditions themselves.

Examples

```
soar> run 1 d

     0: ==>S: S1
Initial state has A on B and B and C on the table.
The goal is to get A on B on C on the table.
     1:    O: O2 (build-tower)

soar> matches top-goal*terminate*operator*build-tower
   1 (state <g> ^operator <o>)
   1 (<o> ^name build-tower)
   1 (<g> ^problem-space <p>)
   1 (<p> ^name top-ps)
   4 (<g> ^object-dynamic <tower>)
>>>> (<tower> ^bottom-block <blockc>)
     (<blockc> ^name c)
     (<tower> ^middle-block <blockb>)
     (<blockb> ^name b)
     (<tower> ^top-block <blocka>)
     (<blocka> ^name a)

0 complete matches.

soar> matches top-goal*terminate*operator*build-tower -wmes
   1 (state <g> ^operator <o>)
   1 (<o> ^name build-tower)
   1 (<g> ^problem-space <p>)
   1 (<p> ^name top-ps)
   4 (<g> ^object-dynamic <tower>)
>>>> (<tower> ^bottom-block <blockc>)
*** Matches For Left ***
(48: S1 ^operator O2)
 (35: O2 ^name build-tower)
 (7: S1 ^problem-space P1)
 (9: P1 ^name top-ps)
 (13: S1 ^object-dynamic T1)

(48: S1 ^operator O2)
 (35: O2 ^name build-tower)
 (7: S1 ^problem-space P1)
 (9: P1 ^name top-ps)
 (12: S1 ^object-dynamic B1)

(48: S1 ^operator O2)
 (35: O2 ^name build-tower)
 (7: S1 ^problem-space P1)
 (9: P1 ^name top-ps)
 (11: S1 ^object-dynamic B2)

(48: S1 ^operator O2)
 (35: O2 ^name build-tower)
 (7: S1 ^problem-space P1)
 (9: P1 ^name top-ps)
 (10: S1 ^object-dynamic B3)

*** Matches for Right ***

     (<blockc> ^name c)
     (<tower> ^middle-block <blockb>)
     (<blockb> ^name b)
     (<tower> ^top-block <blocka>)
     (<blocka> ^name a)

0 complete matches.
```

Notes

When printing partial match information, some of the matches displayed by this command may have already fired, depending on when in the execution cycle this command is called. To check for the matches that are about to fire, use the `matches` command without a named production.

In Soar 8, the execution cycle (decision cycle) is input, propose, decide, apply output; it no longer stops for user input after the decision phase when running by decision cycles (`run 1 d`). If a user wishes to print the match set immediately after the decision phase and before the apply phase, then the user must either run Soar by *phases* (`run 1 p`), or register a callback using the `monitor` command to pause Soar after the decision phase.

6.2.4 **memories** [*prodname* | *n* | *prodtype*]

The `memories` command prints out the internal memory usage for full and partial matches of production instantiations, with the productions using the most memory printed first.

Memory usage is recorded according to the *tokens* that are allocated in the Rete network for the given production(s). This number is a function of the number of elements in working memory that match each production. Therefore, this command will not provide useful information at the beginning of a Soar run (when working memory is empty) and should be called in the middle (or at the end) of a Soar run.

With no arguments, the `memories` command prints memory usage for all productions. If a *production name* is specified, memory usage will be printed only for that production; if a positive integer *n* is given, only `n` productions will be printed: the `n` productions that use the most memory.

Output may be restricted to print memory usage for particular *types* of productions, using one or more of the flags in the following table:

prod type	effect on printing behavior
-user	print memory usage of user-defined productions
-default	print memory usage of default productions
-chunks	print memory usage of chunks
-justifications	print memory usage of justifications

The `memories` command is used to find the productions that are using the most memory and, therefore, may be taking the longest time to match (this is only a heuristic). By identifying these productions, you may be able to rewrite your program so that it will run more quickly. Note that memory usage is just a heuristic measure of the match time: A production might not use much memory relative to others but may still be time-consuming to match, and excising a production that uses a large number of tokens may not speed up your program, because the Rete matcher shares common structure among different productions.

As a rule of thumb, numbers less than 100 mean that the production is using a small amount of memory, numbers above 1000 mean that the production is using a large amount of memory, and numbers above 10,000 mean that the production is using a *very* large amount of memory.

Example

```
soar> memories -chunk

Memory use for productions:

chunk-3: 1
chunk-2: 1
chunk-1: 1
```

6.2.5 **preferences** `[id] [[^] attribute] [0|1|2|3] \ [-none|-names|-timetags|-wmes]`

The `preferences` command prints the current preferences for the specified identifier and attribute; an optional third argument increases the level of detail. If no arguments are specified, the `id` and `attribute` default to the current state and operator, respectively. This command is useful for examining which candidate operators have been proposed and what relationships, if any, exist among them. If a preference has O-support, the string, ":O" will also be printed.

The optional arguments to the `preferences` command are described in the following table. The numeric and named flags are redundant; one may use either a number or a name to specify the level of detail.

preferences		Effect on output
0	`-none`	Print only the preferences themselves (default)
1	`-names`	Also print the names of the productions that generated these preferences
2	`-timetags`	Also print the timetags of the WME's that matched the productions that generated the preferences
3	`-wmes`	Also print the WME's that matched the productions

Example

```
soar> preferences
Preferences for S1 ^operator:

acceptables:
  O20 (move-block) +
  O21 (move-block) +
  O9 (move-block) +

reconsiders:
  O15 (move-block) @

unary indifferents:
  O20 (move-block) =
  O21 (move-block) =
  O9 (move-block) =
soar>
soar> preferences s1 operator 1

Preferences for S1 ^operator:

acceptables:
  O20 (move-block) +
    From blocks-world*propose*move-block
  O21 (move-block) +
    From blocks-world*propose*move-block
  O9 (move-block) +
    From blocks-world*propose*move-block

reconsiders:
  O15 (move-block) @
    From blocks-world*terminate*move-block

unary indifferents:
  O20 (move-block) =
    From blocks-world*compare*move-block*indifferent
  O21 (move-block) =
    From blocks-world*compare*move-block*indifferent
  O9 (move-block) =
    From blocks-world*compare*move-block*indifferent
soar>
```

6.2.6 **print** [-internal] [-name|-full] [-filename] \ {*prodname* |-all|-chunks|-defaults| \ -justifications|-user} **print** [-depth *n*] [-internal] \ {*identifier*|*timetag*|*pattern*} **print -stack** [-state|-operator]

The `print` command prints information about items in production memory and working memory. It will also print the current subgoal stack. The `print` command must take an argument, such as the *name* or *type* of a production (to print productions); an *identifier*, *timetag* or *pattern* (to print an object or element in working memory); or `-stack` to print the current subgoal stack. It is a rather overloaded and complex command, so it is broken down into its three different functions below. Each section defines the pertinent arguments and gives examples.

The output of `print` is dependent on the order of the arguments. Arguments such as `-internal`, `-full`, and `-filename` are applied only to arguments that follow them on the commandline. For example:

```
soar> print -chunks -full  ;# print only the names of chunks (default)

soar> print -full -chunks  ;# print each chunk entirely
```

This may seem inflexible, but it does allow users to print any number of items in memory and have different modifiers apply to different arguments, with a single command.

6.2.6.1 Printing items in production memory.

Usage: **`print`** `[-internal] [-name|-full] [-filename] \`
`{`*`prodname`*`|-all|-chunks|-defaults|-justifications|-user}`

argument	prints
`-internal`	Print productions in their "internal" form
`-name`	Print only the name of the production. This is the default when a production *type* is specified.
`-full`	Print the full production. This is the default when a production *name* is specified.
`-filename`	Print the name of the file that contains the production. Useful when many files are `source`d to load productions.
`prodname(s)`	prints the full named production(s)
`-all`	Print the names of all productions. If `-full` is specified first, print the full productions.
`-chunks`	Print the names of all chunks. If `-full` is specified first, print the full productions.
`-defaults`	Print the names of all default productions. If `-full` is specified first, print the full productions.
`-justifications`	Print the names of all justifications. If `-full` is specified first, print the full productions.
`-user`	Print the names of all user productions. If `-full` is specified first, print the full productions.

comment: underline the unique 2-char substrings that are allowed.

The print command is used to print items in production memory. If a production *name* is specified, Soar will by default print the entire production. If a production *type* is specified, Soar will by default print the names of all productions of that type. The flags `[-name|-full]` can be used to override the defaults and print the entire production for all of the specified type, or print only the name for named productions (which seems rather silly).

The flag `-filename` modifies the output of `print` to include the name of the file that stores the production. This is particularly useful when productions are stored in multiple files that all get loaded when the program starts.

The flag `-internal` is used to print productions as they appear in the Rete, with the conditions in their reordered form.

Examples

```
soar> print -filename blocks-world*compare*move-block*indifferent
# sourcefile : blocks-world.soar
sp {blocks-world*compare*move-block*indifferent
    (state <s> ^operator <o> +)
    (<o> ^name move-block)
    -->
    (<s> ^operator <o> =)
}
soar> print -u
```

comment: ... need the info for the example ...

Notes

The flags `-internal`, `-name|-full`, and `-filename`, must be specified on the commandline *before* the production name or type.

6.2.6.2 Printing items in working memory.

Usage: **`print`** `[-depth` *`n`*`] [-internal]` {*`identifier`*|*`timetag`*|*`pattern`*}

argument	prints
`-depth` *`n`*	for working memory elements only, follows links to print the subobjects of objects to the specified depth
`-internal`	Print the individual WMEs for the item specified
`identifier`	prints the working memory object specified by the identifier
`timetag`	print the object or WME with this integer timetag
`pattern`	print object or WME that matches this pattern

`print` is also used to print items in working memory. The syntax of printing items from working memory is a bit tricky; including the optional flags can alter the output dramatically. By default, `print` retrieves the object associated with the *identifier*, *timetag*, or *pattern* specified as the argument to `print`. (Recall that an object comprises all WMEs with a common identifier.) But depending on whether `-internal` is specified, and what the `-depth` value is, the output generated might be a long list of individual WMEs, or several copies of the same object. The effects of these flags will be described in text, but the reader is encouraged to study the examples.

To print items from working memory, one of the following arguments must be given: an *identifier*, such as `S1`; an integer *timetag*, e.g., `15`; or a *pattern* that will match against one or more elements of working memory, e.g., {`(* ^operator *)`}. The syntax of a *pattern* is exactly the (id ^attribute value) triplet that represents working memory elements, with the addition that wildcards can be used in place of any or all of the components of the triplet. It is important to note that the whole triplet, including the parenthesis, must be enclosed in curly braces for it to be parsed properly. If wildcards are included, an object will be printed for each pattern match, even if this results in the same object being printed multiple times. See Example 2.

Use of the optional `-internal` flag will result in individual WMEs and their timetags being printed, rather than objects. Except for the addition of the timetags, this is just a change in format of the same information. Compare the output of the two commands in Example 1.

The optional `-depth` flag causes the `print` command to follow identifier links in working memory to the depth specified. By default, Soar uses a depth of 1, which causes the print command to follow each link of the identifier exactly one level. A `-depth` of 2 will result in the object being printed and also all objects whose identifiers are values in the first object. The default depth for Soar can be changed through the `default-wme-depth` command (described on page 114), but specifying `-depth` *`n`* in the `print` command will always override the value set by `default-wme-depth`. In what seems like a special case (but really isn't), the output of `print` for `-depth 0` depends on whether or not the `-internal` flag is also specified. If `-internal` is *not* specified, Soar prints the object, and the `-depth` flag has no effect on the output. However, if `-internal` and `-depth 0` are both specified, then Soar will print only the matched WME. Compare the output of the commands in Example 3.

To print individual WMEs for a given argument (identifier or timetag or pattern), use `print -depth 0 -internal` *`arg`*. For convenience, the `wmes` alias described on page 115 provides exactly that notation. To print a single copy of *all* elements in working memory, in their internal form, i.e., as WMEs with timetags, use:

```
soar> print -depth 0 -internal {(* ^* *)}
```

OR

```
soar> wmes {(* ^* *)}
```

In the above `print` statement, omitting `-depth 0` will cause each WME to be printed for every object they are part of (if the default depth is 1); omitting `-internal` will cause the entire object to be printed for each WME.

Example 1 The default behavior of `print` is to print an object. Specifying `-internal` causes the individual WMEs and their timetags to be printed.

```
soar> print s1
(S1 ^io I1 ^ontop O3 ^ontop O2 ^ontop O1
    ^operator O4 + ^operator O7 ^operator O5 + ^operator O6 +
    ^operator O7 + ^operator O8 + ^operator O9 +
    ^problem-space blocks ^superstate nil
    ^thing T1 ^thing B1 ^thing B3 ^thing B2 ^type state)

soar> print -internal s1
```

```
(3: S1 ^io I1)
(9: S1 ^ontop O3)
(10: S1 ^ontop O2)
(11: S1 ^ontop O1)
(48: S1 ^operator O4 +)
(54: S1 ^operator O6)
(49: S1 ^operator O5 +)
(50: S1 ^operator O6 +)
(51: S1 ^operator O7 +)
(52: S1 ^operator O8 +)
(53: S1 ^operator O9 +)
(4: S1 ^problem-space blocks)
(2: S1 ^superstate nil)
(5: S1 ^thing T1)
(8: S1 ^thing B1)
(6: S1 ^thing B3)
(7: S1 ^thing B2)
(1: S1 ^type state)
```

Example 2 The acceptable preferences for operators that may appear in working memory may be printed by following the identifier-attribute-value pattern with a plus sign, +. When wildcards are used, Soar prints the object for each item matched; in this example, six operator preferences are matched in the same object, and no other operator preferences exist.

```
soar> print {(* ^* * +)}
(S1 ^io I1 ^ontop O3 ^ontop O2 ^ontop O1 ^operator O4 + ^operator O6
       ^operator O5 + ^operator O6 + ^operator O7 + ^operator O8 +
       ^operator O9 + ^problem-space blocks ^superstate nil ^thing T1
       ^thing B1 ^thing B3 ^thing B2 ^type state)
(S1 ^io I1 ^ontop O3 ^ontop O2 ^ontop O1 ^operator O4 + ^operator O6
       ^operator O5 + ^operator O6 + ^operator O7 + ^operator O8 +
       ^operator O9 + ^problem-space blocks ^superstate nil ^thing T1
       ^thing B1 ^thing B3 ^thing B2 ^type state)
(S1 ^io I1 ^ontop O3 ^ontop O2 ^ontop O1 ^operator O4 + ^operator O6
       ^operator O5 + ^operator O6 + ^operator O7 + ^operator O8 +
       ^operator O9 + ^problem-space blocks ^superstate nil ^thing T1
       ^thing B1 ^thing B3 ^thing B2 ^type state)
(S1 ^io I1 ^ontop O3 ^ontop O2 ^ontop O1 ^operator O4 + ^operator O6
       ^operator O5 + ^operator O6 + ^operator O7 + ^operator O8 +
       ^operator O9 + ^problem-space blocks ^superstate nil ^thing T1
       ^thing B1 ^thing B3 ^thing B2 ^type state)
(S1 ^io I1 ^ontop O3 ^ontop O2 ^ontop O1 ^operator O4 + ^operator O6
       ^operator O5 + ^operator O6 + ^operator O7 + ^operator O8 +
       ^operator O9 + ^problem-space blocks ^superstate nil ^thing T1
       ^thing B1 ^thing B3 ^thing B2 ^type state)
(S1 ^io I1 ^ontop O3 ^ontop O2 ^ontop O1 ^operator O4 + ^operator O6
       ^operator O5 + ^operator O6 + ^operator O7 + ^operator O8 +
       ^operator O9 + ^problem-space blocks ^superstate nil ^thing T1
       ^thing B1 ^thing B3 ^thing B2 ^type state)
soar>
```

Example 3 When a timetag is specified as the argument to `print`, Soar prints the object that includes the WME with that timetag.

```
soar> print 30
(O4 ^destination B1 ^moving-block B2 ^name move-block)
```

The output when `-depth 0` is specified still prints the object, unless the `-internal` is also specified.

```
soar> print -depth 0 30
(O4 ^destination B1 ^moving-block B2 ^name move-block)

soar> print -depth 0 -internal 30
(30: O4 ^name move-block)
```

6.2.6.3 Printing the current subgoal stack.

`print` can be used to print the current subgoal stack by specifying the `-stack` argument. By default this includes both states and operators. The stack listing can be restricted by adding the `-states` and `-operator` restrictions. Specifying both options is equivalent to the default. The predefined alias 'ps' is the same as print -stack.

Usage: **`print -stack`** `[-operators|-states]`

argument	prints
`-stack`	prints the current subgoal stack
`-operators`	print only the operators
`-states`	print only the states

Examples

```
soar> print -stack
      : ==>S: S1
      :    O: O7 (move-block)
```

6.2.7 **`production-find`** `{[-lhs|-rhs] [-chunks|-nochunks] \ [-show-bindings] {`*`pattern`*`} }`

The `production-find` command is used to find productions in production memory that include conditions or actions that match a given pattern. The pattern given specifies one or more condition elements on the lefthand side of productions (or negated conditions), or one or more actions on the righthand side of productions. Any *pattern* that can appear in productions can be used in the `production-find` command. In addition, the asterisk symbol, `*`, can be used as a wildcard for an attribute or value. It is important to note that the whole *pattern*, including the parenthesis, must be enclosed in curly braces for it to be parsed properly.

The variable names used in a call to `production-find` do not have to match the variable names used in the productions being retrieved.

The `production-find` command can also be restricted to apply to only certain types of productions, or to look only at the conditions or only at the actions of productions, by using the flags shown in the following table:

flag	Effect on productions found
`-lhs`	Match pattern only against the conditions (lefthand side) of productions (default)
`-rhs`	Match pattern only against the actions (righthand side) of productions
`-chunks`	Look only for chunks that match the pattern
`-nochunks`	Disregard chunks when looking for the pattern
`-showbindings`	Show the bindings associated with a wildcard pattern

Examples

```
soar> production-find {(<s> ^operator <o>)(<o> ^name move-block)}
blocks-world*monitor*move-block
blocks-world*terminate*move-block
blocks-world*apply*move-block*remove-old-clear
blocks-world*apply*move-block*add-new-ontop
blocks-world*apply*move-block*add-new-clear
blocks-world*apply*move-block*remove-old-ontop

soar> production-find -showbindings -rhs \
      {(<state> ^operator <op>)(<op> ^name move-block)}
blocks-world*propose*move-block, with bindings:
   (<state> -> <s>)
   (<op> -> <o>)

soar> production-find -rhs -lhs \
      {(<s> ^operator <o> +)(<o> ^name move-block)}
blocks-world*select*move-block*indifferent
blocks-world*propose*move-block
```

6.2.8 `default-wme-depth` [*n*]

The `default-wme-depth` command reflects the default depth used when working memory elements are printed (using the `print` command or `wmes` alias). The default value is 1. When the command is issued with no arguments, `default-wme-depth` returns the current value of the default depth. When followed by an integer value, `default-wme-depth` sets the default depth to the specified value. This default depth can be overridden on any particular call to the `print` or `wmes` command by explicitly using the `-depth` flag, e.g.,`print -depth 10` *`args`*.

Recall that by default, the `print` command prints *objects* in working memory, not just the individual working memory element. To limit the output to individual working memory elements, the `-internal` flag must also be specified in the `print` command. Thus when the print depth is `0`, by default Soar prints the entire object, which is the same behavior as when the print depth is `1`. But if `-internal` is also specified, the a depth of `0` prints just the individual WME, while a depth of `1` prints all WMEs which share that same identifier. This is true when printing timetags, identifiers or WME patterns.

When the depth is greater than `1`, the identifier links from the specified WME's will be followed, so that additional substructure is printed. For example, a depth of `2` means that the object specified by the identifier, wme-pattern, or timetag will be printed, along with all other objects whose identifiers appear as values of the first object. This may result in multiple copies of the same object being printed out. If `-internal` is also specified, then individuals WMEs and their timetags will be printed instead of the full objects.

Example

```
soar> default-wme-depth 2
soar> default-wme-depth
2
```

Notes

See the `print` command on page 107 for more information and examples on default print depth.

6.2.9 **wmes** {*identifier* | *timetag* | *pattern*}

The `wmes` alias is a shorthand for `print -depth 0 -internal`, used to print working memory elements. The `wmes` command always prints WME's in their internal form, i.e., as separate working memory elements with timetags, rather than as objects.

As with the `print` command, the depth can be set to any level by using the optional `-depth` flag, but then WMEs may be printed multiple times, once for each object they are part of.

Examples The first example prints all WMEs for the object, `a1`:

```
soar> wmes a1
(20: A1 ^name a)
(19: A1 ^type block)
```

This example prints the WME with the timetag, `20`:

```
soar> wmes 20
(20: A1 ^name a)
```

This example will print one copy of each element of working memory:

```
soar> wmes {(* ^* *)}
```

6.3 Configuring Trace Information and Debugging

This section describes the commands used primarily for debugging or to configure the trace output printed by Soar as it runs. Users may: specify the format and content of the runtime trace output; ask that they be alerted when specific productions fire and retract; specify certain actions be taken at various points in the execution cycle; or request details on Soar's performance.

The specific commands described in this section are:

Summary

chunk-name-format - Specify format of the name to use for new chunks.

firing-counts - Print the number of times productions have fired.

format-watch - Change the trace output that's printed as Soar runs.

monitor - Manage attachment of Tcl scripts to Soar events.

pwatch - Trace firings and retractions of specific productions.

stats - Print information on Soar's runtime statistics.

warnings - Toggle whether or not warnings are printed.

watch - Control the information printed as Soar runs.

Of these commands, `watch` is the most often used (and the most complex). `pwatch` is related to `watch`, but applies only to specific, named productions. `firing-counts` and `stats` are useful for understanding how much work Soar is doing. Both `format-watch` and `chunk-name-format` are less-frequently used, but they allow for detailed control of Soar's output. `monitor` is especially useful when interacting with an external environment, but can be used anytime it is useful to always take a specific action at a specific point in Soar's execution cycle. The `monitor` command is discussed in more depth in the *Soar Advanced Applications Manual.*

6.3.1 `chunk-name-format [-short|-long] \ [-prefix [prefix]] [-count [n]]`

The chunk-name-format command specifies the format to be used when naming newly created chunks.

flag	Effect on chunk name
`-short`	Use the short format for naming chunks
`-long`	Use the long format for naming chunks (default)
`-prefix [`*string*`]`	If *string* is given, use *string* as the prefix for naming chunks. Otherwise return the current *prefix.* (defaults to "`chunk`")
`-count` *n*	If *n* is given, set the chunk counter for naming chunks to *n.* Otherwise return the current value of the chunk counter.

The short format for naming newly-created chunks is:

prefixChunknum

The long (default) format for naming chunks is:

*prefix-Chunknum**d*dc*impassetype*dcChunknum*

where:

prefix	is a user-definable prefix string; *prefix* defaults to "`chunk`" when unspecified by the user. It may *not* contain the character `*`.
Chunknum	is *n* for the first chunk created, *n+1* for the second chunk created, etc.
dc	is the number of the decision cycle in which the chunk was formed.
impassetype	is one of `[tie	conflict	cfailure	snochange	opnochange]`.
dcChunknum	is the number of the chunk within that specific decision cycle.

Notes

The `chunk-name-format` command enforces the constraint that the *prefix* string may not contain the “`*`” character.

The `chunk-name-format` command enforces the constraint that the Chunknum cannot be reset to a number lower than the smallest chunk number of any loaded chunk.

The '`*`' character marks the ending of the number of the chunk, and chunk numbers are examined during production loading to allow a starting chunk number to be identified automatically.

`impasse-type` is the impasse of the lowest goal that whose result generated this chunk.

Examples Default naming of chunks uses the long form:

```
soar> print -c
chunk-1*d4*opnochange*1
chunk-2*d5*tie*1
chunk-4*d8*opnochange*1
chunk-5*d8*tie*2
chunk-7*d13*opnochange*1
```

To name chunks according to the “old” (pre-Soar8) scheme:

```
soar> chunk-name-format -short -prefix "chunk" -count 0
```

To begin naming a new sequence of chunks of the form `new-chunk`*n*, where *n* starts with 1 and is incremented by 1 for each new chunk:

```
soar> chunk-name-format -short -prefix "new-chunk"
```

To begin numbering a new sequence of chunks at 1000.

```
soar> chunk-name-format -count 1000
```

To retrieve the next *chunk-num* that will be used to form a chunk name.

```
soar> chunk-name-format -count
8
```

6.3.2 `firing-counts` [*n* | *prodname(s)*]

The `firing-counts` command prints the number of times each production has fired; production names are given from most frequently fired to least frequently fired. With no arguments, it lists *all* productions. If an integer argument, `n`, is given, only the top `n` productions are listed.

If `n` is zero (`0`), only the productions that haven't fired at all are listed. If one or more production names are given as arguments, only firing counts for these productions are printed.

Note that firing counts are reset by a call to `init-soar`.

Example

```
soar> firing-counts 10
    34:  default*generic*elaborate*add-attribute-to-duplicate
    17:  blocks-world*propose*operator*move-block
    11:  blocks-world*monitor*state*relation*ontop
     8:  default*generic*elaborate*state*add-duplicate-to-state
     8:  blocks-world*reject*move-block*twice
     8:  default*generic*elaborate*state*duplicate-id-for-attribute
     7:  blocks-world*monitor*tied-operators*move-block
     7:  default*selection*propose*operator*evaluate-object
     7:  default*selection*elaborate*state*with-wait-true-not-all-
             objects-evaluated
     7:  default*selection*select*operator*evaluate-object*indifferent
soar> firing-counts blocks-world*propose*operator*move-block
    17:  blocks-world*propose*operator*move-block
```

6.3.3 **`format-watch`** `[-stack|-object] \` `[[-add|-remove] [o|s|*] [`*`name`*`] {`*`format`*`}]`

The `format-watch` command allows users to customize the appearance of the output printed as Soar runs. Unlike the `watch` command, which controls *what* information is printed, the `format-watch` command controls *how* this information is printed, that is, how it is formatted. Generally, this command is not used by beginners, but it gives great flexibility for customizing the runtime appearance of Soar. Readers may wish to read through the section on the `watch` command on page 132, before reading this section.

The formatting is controlled by a set of formatting rules, each having certain applicability conditions[1] and a *format string*. When Soar wants to print something, it looks for an applicable rule (choosing the most specific one if more than one is applicable) and uses that format string to control the printout.

[1]In general, these conditions could be made arbitrarily complicated, and a whole Rete net could be used to match the rules. However, we have opted for simplicity here instead: the conditions are very restricted, so that the matching (actually, lookup) can be done in constant time.

argument	Effect on format-watch
`-stack`	applies to printing of context stack trace (default)
`-object`	applies to printing of objects
`-add`	add a new format possibly replacing an old format
`-remove`	remove an existing format
`s`	restrict to *states* only
`o`	restrict to *operator* only
`*`	do not restrict by *type*
`name`	for `-stack`, restrict to *name*d problem spaces for `-object`, restrict to objects with ^`name` *name*
`format`	sequence of format masks to apply

All calls to `format-watch` should take either `-stack` or `-object` as an argument, to specify whether the rule applies to stack formats or object formats. Stack formats control how Soar prints its context stack selections in `watch 1` and `print -stack` output. Object formats control how Soar prints an object, e.g., a certain operator, problem-space, etc. If no type is given, then `-stack` is assumed.

With no additional arguments, `format-watch` prints the current formatting settings (for either stacks or objects, as specified).

Any additional arguments are considered an *action*, which add or delete a formatting rule. The optional *action* takes the form:

`operation class [name] format`

The *operation* must be either `-add` or `-remove`. An `-add` operation adds new formats, replacing any existing ones with identical applicability conditions. A `-remove` operation removes formats with the given applicability conditions. The combination of *class* and *name* define the applicability conditions of the format (i.e., which classes of items the format applies to). The class must be either `s` or `o` and indicates that the operation applies to states or operators, respectively. The wildcard symbol `*` may also be used to indicate that the format applies to all classes of items. If an `-object` trace is being manipulated, then an optional *name* may be given indicating the format applies only to objects with that *name*. If a `-stack` trace is being manipulated, then an optional name may be given indicating the format applies only within problem spaces of that name.

The *format* string can be any sequence of characters surrounded by curly braces. Note that curly braces must be used to delimit the *format* otherwise any square brackets in the format string will be interpreted as a command to be evaluated by Tcl. A set of formatting controls can be used within the *format* string; see the subsection "Formatting Controls" below for more information.

Note that there can only be one rule with given applicability conditions in the system at a time. If you try to `-add` a second rule with identical conditions, the first one is removed and replaced by the second.

Also note that there can be two rules with different applicability conditions that happen to apply to the same object. For example, there could be one rule with a name restriction that matches a certain object, and a second rule that has no name restriction (and thus also matches the object). In this case, Soar uses whichever rule has the most specific conditions. (For purposes of this, name restrictions are considered more specific than type restrictions.)

Formatting Controls

The format strings have *escape sequences* embedded in them, starting with a `%` sign. Soar expands the escape sequences into the appropriate pieces of text. For example, the format string `%right[6,%dc]` means print the current decision cycle number, right justified in a field 6 characters wide.

format string	Effect on output
`%%`	print a percent sign.
`%[`	print a left bracket.
`%]`	print a right bracket.
`%nl`	print a newline.
`%left[`*num,pattern*`]`	print the given *pattern*, left justified in a field of *num* spaces.
`%right[`*num,pattern*`]`	print the given *pattern*, right justified in a field of *num* spaces.
`%id`	print the identifier of the current object.
`%v[`*attr*`]`	print the value(s) of attribute ^*attr* on the current object. If there is no ^*attr* on the current object, nothing is printed.
`%v[`*path*`]`	same as the above, only follow the given *path* to get the value(s). A *path* is a series of attribute names separated by periods, such as foo.bar.baz
`%v[*]`	print all values of all attributes for current object.
`%o[`*args*`]`	same as `%v`, except that if the value is an identifier, it is printed using the appropriate object format rather than just as `O37`, for example.
`%av[`*args*`]`	same as `%v`, except the printed value is preceded with `^attr` to indicate the attribute name.
`%ao[`*args*`]`	a combination of the above two.
`%ifdef[`*pattern*`]`	print the given *pattern* if and only if all escape sequences inside it are "meaningful" or "well-defined." For example, "`%ifdef[foo has value:  %v[foo]]`" will print nothing if there is no `^foo` attribute on the current object.

The following escape sequences are valid only for `-stack` formats:

format string	Effect on output
`%%`	print a percent sign.
`%cs`	print the current state using the defined format.
`%co`	print the current operator using the defined format.
`%dc`	print the current decision cycle number. (not meaningful in stack traces produced by the `(print -stack)` command. In these cases, nothing is printed.)
`%ec`	print the current elaboration cycle number. (not meaningful in stack traces produced by the `(print -stack)` command. In these cases, nothing is printed.)
`%sd`	print the current substate depth (where 0 is the top-level state).
`%rsd[`*pattern*`]`	print the given *pattern*, repeating it *substate depth* times. The pattern may contain other escape sequences.

Examples

The following stack formats are the built-in defaults in Soar, and yield the `watch 1` trace seen when Soar is running:

```
format-watch -stack -add s {%right[6,%dc]: %rsd[   ]==>S: %cs}
format-watch -stack -add o {%right[6,%dc]: %rsd[   ]   O: %co}
```

In the first example above, the format string `%right[6,%dc]` means print the current decision cycle number, right justified in a field 6 characters wide. After that, the format string says to print a colon and a space. The `%rsd` escape causes the pattern in the brackets (three spaces) to be printed *current-substate-depth* times. Finally, the characters "`==>S`" should be printed followed by the current state, using whatever object format is appropriate. The second example above, is left as an exercise for the reader.

The following object formats are the built-in defaults for Soar:

```
format-watch -object -add * {%id %ifdef[(%v[name])]}
format-watch -object -add s {%id %ifdef[(%v[attribute] %v[impasse])]}
format-watch -object -add o evaluate-object
                            {%id (evaluate-object %o[object])}
```

The first command adds a rule that affects both states and operators; this rule prints an identifier followed by its name in parentheses (if a name is defined). This rule will always apply unless a more specific rule also holds for the same item.

The second command adds a rule that affects only the printing of state objects; this rule tprints the state identifier followed by, in parentheses, its ^`attribute` and ^`impasse` attributes, if they are defined.

The third command adds a rule that affects only the printing of operator objects; this rule handles specifically the `evaluate-object` operators in the `selection` space. These operators will be printed as an identifier followed by, in parentheses, the string "evaluate-object " and the printed representation of the object being printed.

Notice that the second and third rules are more specific than the first, and either will take precedence over the first. Also, the third rule gives a special format to be used only on operators with a certain name. Users may find this technique useful in their own code, to get certain parameters printed on certain operators but different parameters printed on other operators.

The following formatting rules produce a Soar trace where instead of indenting for states, the level of subgoaling prefaces each state or operator symbol.

```
format-watch -stack -add s {%left[6,%dc] (%sd)   S: %cs}
format-watch -stack -add o {%left[6,%dc] (%sd)   O: %co}
```

The following formatting rule causes both the current state and current operator to be printed whenever an operator is selected. (There is a linefeed in the middle of the format string.)

```
format-watch -stack -add o {%right[6,%dc]: %rsd[   ]   S: %cs
        %rsd[   ]   O: %co}
```

This format can be useful for watching the effects of a series of operator applications. Each time an operator is selected, the current state is also printed, so users can see what modifications the previous operator made to the state.

6.3.4 monitor *action*

The `monitor` command manages the attachment of Tcl scripts to specific Soar events. Soar events are important events that occur in Soar throughout the execution cycle. Scripts can be attached to these Soar events so that they are invoked every time the Soar event occurs. These scripts can also be removed using the `monitor` command. The status of attachments can also be listed and tested. If a new attachment is created, its id is returned as the result of this command. This id is used when deleting a specific script from an event.

The `monitor` command is particularly useful when Soar must interact with an external environment, but this is considered an advanced usage and is not within the scope of this manual. A more detailed description of the use of the `monitor` command, and specific useful examples are given in the *Soar Advanced Applications Manual.*

The `monitor` command must be followed by a specific *action* which must have one of the following forms:

`-add` ***soar-event script*** [***id***] Add the *script* to the list of scripts to be invoked when the *soar-event* occurs during execution. See below for a list of valid Soar event names. If the *id* is given, it is used to name the attachment. If no *id* is given, then a unique id is generated and returned.

`-delete` ***soar-event*** [***id***] Remove scripts attached to *soar-event*. The *soar-event* must match a valid Soar event name (see below). If *id* is given after a soar-event, then only that particular attachment is removed. Otherwise, all attachments under *soar-event* are removed.

`-list` For every Soar event, list the attached scripts.

`-test` Test the script attachment process by attaching a script to print a message to every Soar event.

`-clear` Remove *all* attached scripts from every Soar event.

Soar Events

The `monitor` command utilizes the Soar callback system which can associate Tcl or C functions with Soar events. The permissable Soar event names for use with `monitor` listed in the following table:

Event name	Description
`after-init-agent`	An agent has just been created and initialized.
`before-init-soar`	The agent is about to be initialized. The `init-soar` command will generate this event.
`after-init-soar`	The agent has just been initialized. The `init-soar` command will generate this event.
`after-halt-soar`	The agent has just been halted.
`before-schedule-cycle`	This event is triggered by the `run` command. it occurs just before the agent is run.
`after-schedule-cycle`	This event is triggered by the `run` command. it occurs just after the agent is run
`before-decision-cycle`	A decision cycle is just about to start.
`after-decision-cycle`	A decision cycle has just ended.
`before-input-phase`	An input phase is just about to start.
`after-input-phase`	An input phase has just ended.
`before-preference-phase-cycle`	A preference phase is just about to begin.
`after-preference-phase-cycle`	A preference phase has just ended.
`before-wm-phase-cycle`	A working memory phase is just about to begin.
`after-wm-phase-cycle`	A working memory phase is just about to begin.
`before-decision-phase-cycle`	A decision phase is about to begin.
`after-decision-phase-cycle`	A decision phase has just ended.
`before-output-phase`	An output phase is just about to begin.
`after-output-phase`	An output phase has just ended.
`wm-changes`	Changes to working memory have just completed.
`create-new-context`	A new state has been created on the goal stack.
`pop-context-stack`	A subgoal has finished.
`production-just-added`	A productions has just been added to the agent.
`production-just-about-to-be-excised`	A production is just about to be removed from the agent.
`firing`	A production instantiation has just fired.
`retraction`	A production instantiation is just about to retract.
`system-parameter-changed`	A system parameter has just been changed.
`system-termination`	The Soar system is exiting.

Examples The following command creates an attachment which prints a message after every Soar decision cycle:

```
monitor -add after-decision-cycle {puts "Finished DC!"}
```

The next example creates an attachment which calls the Tcl pro- cedure "DoSomething" (presumably user-defined) after every production firing:

```
monitor -add firing DoSomething
```

The next example creates an attachment which processes pending X events after every decision cycle. This is often needed in long-running Soar agents (agents that run more than a few decision cycles at a time) so that GUIs do not appear frozen:

```
monitor -add after-decision-cycle update
```

The following example removes the single attachment "m1" which asso- ciates a script with the Soar event after-init-agent:

```
monitor -delete after-init-agent m1
```

The next example removes all script attachments to the Soar event before-preference-phase-cycle:

```
monitor -delete before-preference-phase-cycle
```

Finally, this example removes all script attachments that have been added with the monitor command:

```
monitor -clear
```

6.3.5 **pwatch** [-on|-off] [*production-name(s)*]

The `pwatch` command enables and disables the tracing of the firings and retractions of individual productions. This is a companion command to `watch`, which cannot specify individual productions by name.

With no arguments, `pwatch` lists the productions currently being traced. With one or more *production-name* arguments, `pwatch` enables tracing of those productions; `-on` can be explicitly stated, but it is the default action. If `-off` is specified followed by one or more *production-names*, tracing is turned off for those productions.

When no *production-names* are specified, `pwatch -on` lists all productions currently being traced, and `pwatch -off` disables tracing of all productions.

Example

```
soar> pwatch -on blocks-world*terminate*move-block
soar> pwatch
 blocks-world*terminate*move-block
soar> run 3 d

     0: ==>S: S1
Initial state has a, b, and c on the table.
     1:    O: O6 (move-block)
Moving Block: c to: b
Firing blocks-world*terminate*move-block
```

```
     2:    O: O8 (move-block)
Moving Block: c to: a
Retracting blocks-world*terminate*move-block
Firing blocks-world*terminate*move-block

     3:    O: O11 (move-block)
```

6.3.6 stats [-system [*stat*]] | \
[-memory [*resource*]] | \
[-rete [*rtype qualifier*]]

The `stats` command provides statistical information about Soar's internal workings. Users can obtain summary information on the whole system, the Rete net, or memory pools; or can request single values for individual items. Most users are interested in the summary information for the system and might examine more detailed information only when debugging.

6.3.6.1 stats [-system [*stat*]]

With no arguments or if `-system` is specified, the `stats` command lists a summary of run statistics, including the following:

Version — The Soar version number, hostname, and date of the run.

Number of productions — The total number of productions loaded in the system, including all chunks built during problem solving and all default productions.

Timing Information — Might be quite detailed depending on the flags set at compile time. See Appendix **??** on page **??**.

Decision Cycles — The total number of decision cycles in the run and the average time-per-decision-cycle in milliseconds.

Elaboration cycles — The total number of elaboration cycles that were executed during the run, the everage number of elaboration cycles per decision cycle, and the average time-per-elaboration-cycle in milliseconds. This is not the total number of production firings, as productions can fire in parallel.

Production Firings — The total number of productions that were fired.

Working Memory Changes — This is the total number of changes to working memory. This includes all additions and deletions from working memory. Also prints the average match time.

Working Memory Size — This gives the current, mean and maximum number of working memory elements.

The following specific arguments may follow `-system` to limit the output to the desired information:

```
-dc-count                  -firings-count
-ec-count                  -firings/ec
-ecs-dc                    -wme-change-count
-all-productions-count     -wme-addition-count
-chunk-production-count    -wme-removal-count
-default-production-count  -wme-count
-justification-count       -wme-avg-count
-user-production-count     -wme-max-count
```

If Soar has been compiled with the `NO_TIMING_STUFF` *not* set, then the following statistics are also available with the `-system` argument:

```
-ms/dc
-ms/ec
-ms/firing
-total-time
```

If Soar has been compiled with the `DETAILED_TIMING_STATS` set to `TRUE`, then the following statistics are also available in the `-system` module:

```
-chunking-time
-match-time
-ms/wme-change
-ownership-time
```

The standard distributions of Soar have been compiled to include all the above statistics by default.

Example

```
soar> stats
Soar 8.2 on ledoux.ummu.umich.edu at Thu May  6 10:41:53 1999

91 productions (79 default, 12 user, 0 chunks)
   + 1 justifications
                                                         |  Derived
Phases:     Input    DLP    Pref     W/M   Output  Decision|   Totals
===========================================================|=========
Kernel:     0.080   0.100   2.570   0.710   0.030    0.920 |    4.410
===============  Detailed Timing Statistics   =============|=========
   Match:   0.020   0.020   0.000   0.470   0.000    0.040 |    0.550
```

```
Own'ship:  0.000   0.000   0.000   0.000   0.000    0.000 |    0.000
Chunking:  0.000   0.000   0.040   0.000   0.000    0.000 |    0.040
   Other:  0.060   0.080   2.530   0.240   0.030    0.880 |    3.820
Operand2:  0.000   0.060   0.000   0.100   0.000    0.000 |    0.160
==========================================================|=========
Input fn:  0.020                                          |    0.020
==========================================================|=========
Outpt fn:                                  0.020          |    0.020
==========================================================|=========
Callbcks:  0.050   0.000   0.020   0.010   0.020    0.220 |    0.320
==========================================================|=========
Derived---------------------------------------------------+---------
Totals:    0.150   0.100   2.590   0.720   0.070    1.140 |    4.770

Values from single timers:
 Kernel CPU Time:       4.980 sec.
 Total  CPU Time:       5.610 sec.

23 decision cycles (216.522 msec/dc)
18 elaboration cycles (0.783 ec's per dc, 276.667 msec/ec)
4 p-elaboration cycles (0.174 pe's per dc, 1245.000 msec/pe)
87 production firings (4.833 pf's per ec, 57.241 msec/pf)
486 wme changes (355 additions, 131 removals)
    match time: 1.132 msec/wm change
WM size: 224 current, 102.376 mean, 240 maximum

    *** Time/<x> statistics use the total kernel time from a ***
    *** single kernel timer.  Differences between this value ***
    *** and the derived total kernel time  are expected. See ***
    *** help  for the  stats command  for more  information. ***
```

6.3.6.2 **stats [-memory [*resource* | -pool *pool-stat*]]**

The optional `stats` argument `-memory` provides information about memory usage and Soar's memory pools, which are used to allocate space for the various data structures used in Soar. The following specific arguments may follow `stats -memory` to limit the output to the desired information:

```
-hash-table
-misc
-overhead
-pool [pool-statistic]
-strings
-total
```

where *`pool-statistic`* is either `-total` or *`pool-name [aspect]`*. The available *`pool-names`* are:

```
action             {node varnames}
{alpha mem}        not
{chunk condition}  {output link}
{complex test}     preference
condition          production
{cons cell}        {rete node}
{dl cons}          {rete test}
{float constant}   {right mem}
identifier         {saved test}
instantiation      slot
{int constant}     {sym constant}
{io wme}           token
{ms change}        variable
{negative token}   wme
```

Note that the two-word *pool-names* must be enclosed in curly braces.

If no *aspect* is given, then print all statistics about the given pool. If given, *aspect* must have one of the following forms:

```
-used
-free
-item-size
-total-bytes
```

The first two *aspects* are available only if Soar has been compiled with `MEMORY_POOL_STATS` set to `TRUE`, which is the default setting for the Soar distribution.

Examples

```
soar> stats -memory
 1259813 bytes total memory allocated
    2328 bytes statistics overhead
    8149 bytes for strings
  133340 bytes for hash tables
 1112688 bytes for various memory pools
    3308 bytes for miscellaneous other things
Memory pool statistics:

Pool Name        Used Items  Free Items  Item Size  Total Bytes
---------------  ----------  ----------  ---------  -----------
chunk condition           0         818         40        32720
```

```
io wme                  0           0          16            0
output link             0           0          24            0
preference            217         192          80        32720
wme                   228         161          84        32676
slot                  157         184          96        32736
instantiation          84         545          52        32708
ms change               6         676          48        32736
right mem             478         545          32        32736
token                2441        3399          56       327040
node varnames         796        1251          16        32752
rete node             740         624          48        65472
rete test             181        2548          12        32748
alpha mem             106         638          44        32736
saved test              0        2729          12        32748
not                     1        2728          12        32748
action                230        1134          24        32736
production             92         537          52        32708
condition             560         122          48        32736
complex test           12        4082           8        32752
float constant          0           0          24            0
int constant           10        1354          24        32736
sym constant          229         940          28        32732
identifier             64         250         104        32656
variable              173         645          40        32720
dl cons                 0        2729          12        32748
cons cell             245        3849           8        32752
```

6.3.6.3 **`stats [-rete [`*`rtype qualifier`*`]]`**

The optional `stats` argument `-rete` provides information about node usage in the Rete net, the large data structure used for efficient matching in Soar. To restrict information to specific modules, an *rtype* and *qualifier* may be specified. The following *rtypes* are available:

`{unhashed memory}`	`memory`
`{unhashed mem-pos}`	`mem-pos`
`{unhashed negative}`	`negative`
`{unhashed positive}`	`positive`
`{dummy top}`	`{dummy matches}`
`{conj. neg.}`	`{conj. neg. partner}`
`production`	`total`

Note that the two-word *rtypes* must be enclosed in curly braces.

and *qualifier* must be one of the following:

```
-actual
-if-no-merging
-if-no-sharing
```

The `total` statistic gives a total count over all node types. The `-if-no-sharing` option is available only if Soar has been compiled with `SHARING_FACTORS` set to `TRUE`, which is the default for the Soar distribution.

Examples

```
soar> stats -rete
      Node Type          Actual  If no merging
---------------------  ---------  -------------
      unhashed memory          2             29
               memory         28            485
     unhashed mem-pos         27              0
              mem-pos        457              0
    unhashed negative          0              0
             negative         31             31
    unhashed positive         23             50
             positive         73            530
            dummy top          1              1
        dummy matches          0              0
           conj. neg.          3              3
   conj. neg. partner          3              3
           production         92             92
---------------------  ---------  -------------
                Total        740           1224
```

6.3.7 **warnings** [-on|-off]

The `warnings` command controls whether warnings are printed during the loading of productions. The default value for `warnings` is `-on`; it may also be set to `-off`. With no argument, the `warnings` command returns the current setting. If warnings are disabled using this command, then *most* warnings are supressed, but some aren't because they are too important to be ignored.

The warnings that are printed apply to the syntax of the productions, to notify the user when they are not in the correct syntax. When a lefthand side error is discovered (such as conditions that are not linked to a common state or impasse object), the production is generally loaded into production memory anyway, although this production may never match or may seriously slow down the matching process. Righthand side errors, such as preferences that are not linked to the state, usually result in the production not being loaded.

Example

```
soar> warnings -off

soar> warnings
off
```

6.3.8 watch [0 | 1 | 2 | 3 | 4 | 5]

```
watch [ 0 | 1 | 2 | 3 | 4 | 5 ]
watch [none | decisions | phases | productions |  \
        wmes | preferences] [-on|-off|-inc[lusive]]
watch productions [-all | -default | -user |  \
                   -chunks | -justifications  \
                  {-print|-noprint}]
                  [-nowmes|-timetags|-fullwmes]
watch wmes filter-options
watch learning  [-print | -noprint | -fullprint]
watch aliases  [-on | -off]
watch loading  [-on | -off]
watch backtracing  [-on | -off]
```

The `watch` command controls the amount of information that is printed out as Soar runs. The information controlled by this setting pertains to Soar's "internal state": state and operator decisions, the productions that fire and retract, and changes to working memory and preference memory. The `watch` setting has no effect on output produced by RHS `write` actions or RHS calls to Tcl functions.

With no arguments, `watch` will print the information about the current `watch` "settings," i.e., the values of each parameter. The functionality of the `watch` command is quite overloaded, and the syntax is somewhat complex; therefore the description will be presented in the following subsections: basic watch settings, watching productions, watching wmes, watching learning, and watching other functions.

6.3.8.1 Basic Watch Settings

The basic functionality of the `watch` command is to trace various *levels* of information about Soar's internal workings. The higher the *level*, the more information is printed as Soar runs. At the lowest setting, `0 | none`, nothing is printed. The levels are cumulative, so that each successive level prints the information from the previous level as well as some additional information. The default setting for the `watch` *`level`* is `1`, (or `decisions`). The levels are described in the following table:

watch	effect on the Soar trace
`0 | none`	print nothing about Soar's internals
`1 | decisions`	print the state and operator selected for each decision cycle (the default watch level)
`2 | phases`	also print out the *phases* of the decision cycle as Soar executes
`3 | productions`	also print the names of productions as they fire and retract (including chunks and justifications)
`4 | wmes`	also trace the working memory elements that are added and deleted as productions fire and retract
`5 | preferences`	also print the preferences asserted and retracted when productions fire and retract.

The numerical arguments (`0 - 5`) do not take any arguments or modifiers. They inclusively turn on all levels up to the number specified. To use numerical arguments to turn off a level, specify a number which is less than the level to be turned off. For instance, to turn off watching of produc- tions, specify "`watch 2`" (or `1` or `0`). Numerical arguments are provided for shorthand convenience. For more detailed control over the watch settings, the named arguments should be used.

The named arguments can have one of the additional switches, `[-on | -off | -inc]`. Specifying the `-inc` switch (which fully specified is `-inclusive`), or setting no flag at all, has the effect of setting all levels up to, and including, the level specified. This is the same behavior as when the equivalent numeric argument is used. Setting either the `-on` or `-off` switch selectively turns on or off *only* that level. For example, `watch productions -on` selectively turns on the tracing of production firings/retractions; `watch productions -off` selectively turns it off again. `watch productions [-inc]` turns on the tracing of productions and also turns on tracing of all levels below productions: decisions and phases, too.

The numeric and named flags may also be used in combinations. For example, you might want to say something like "`watch 3 phases -off`" to watch state and operator decisions and to see the names of productions that fire and retract, but to omit the printing of phases.

Examples

```
soar> watch 1
Current watch settings:
  Decisions:  on
  Phases:  off
  Production firings/retractions
    default productions:  off
    user productions:  off
    chunks:  off
    justifications:  off
    WME detail level:  1
  Working memory changes:  off
  Preferences generated by firings/retractions:  off

  Learning:  -noprint  (watch creation of chunks/just.)
  Backtracing:  off
  Alias printing:  (null)
  Loading:  on

soar> watch 3 phases -off
soar> watch
Current watch settings:
  Decisions:  on
  Phases:  off
  Production firings/retractions
    default productions:  on
    user productions:  on
    chunks:  on
    justifications:  on
    WME detail level:  1
  Working memory changes:  off
  Preferences generated by firings/retractions:  off

  Learning:  -noprint  (watch creation of chunks/just.)
  Backtracing:  off
  Alias printing:  (null)
  Loading:  on

soar>
```

Notes

In order to watch `preferences` or `wmes`, users must also watch `productions`. Preferences and WMEs can be `watch`ed only if the productions that create them are `watch`ed. So if only `-user` productions are being watched, then any preference or wme activity resulting from `-defaults` and `-chunks` will not be printed.

Only when a `[-on | -off]` switch is specified, are the other current settings preserved. When using an inclusive or numeric setting, all levels are turned off, and only the appropriate settings are turned back on.

6.3.8.2 Watching Productions

By default, the names of the productions are printed as each production fires and retracts (at `watch` levels `3` and higher). However, it may be more helpful to watch only a specific *type* of production. The tracing of firings and retractions of productions can be limited to only certain types by modifying the `watch productions [-on|-off|-inc]` command using the following flags:

flag	**subset of productions**
`-all`	applies to all productions
`-defaults`	applies to default productions only
`-user`	applies to user productions only
`-chunks`	applies to chunks only
`-justifications`	applies to justifications only

followed by either `-print` or `-noprint`. And in the future users should be able to specify `-fullprint` to get the full production printed instead just the production name. Each *type* may also be specified with its unique first letter; for example, `-d` is equivalent to `-defaults`. Combinations may be formed by combining letters, such as `-cj` to specify chunks and justifications. Note that substrings longer than the unique first letter will not work in this context; either the first letter only, or the full string must be specified.

Finally, when watching productions, users may set the level of detail to be displayed for WMEs that are added or retracted as productions fire and retract.

flag	**effect on WME information**
`-nowmes`	don't print any info about WMEs
`-timetags`	print only the timetags for WMEs
`-fullwmes`	print the full WMEs added or retracted as watched productions fire and retract.

Detailed information about WMEs will be printed only for productions that are being watched. For instance, if no `-chunks` are being watched, then no WME details for chunks will be printed.

So the full syntax for fine-tuning the watching of productions is:

Usage:
```
watch productions [-on|-off|-inc] \
          {-all|-chunks|-defaults|-justifications|-user} \
          -print | -noprint  \
          [-nowmes | -timetags | -fullwmes ]
```

Examples

To watch all productions, phases and decisions:

```
soar> watch productions
```

To watch all productions, phases and decisions, but not default productions:

```
soar> watch productions -inc -defaults -noprint
```

To watch only `-user` productions fire, and not affect other `watch` settings:

```
soar> watch productions -off -u -print
```

To do the same as above, but also see the full WMEs: `watch` settings:

```
soar> watch productions -off -u -print -fullwmes
```

Notes

To watch specific *named* productions, use the `pwatch` command.

In the future the `-print | -noprint` arguments may be modified so that `-on | -off` will also work, and perhaps the `-fullprint` option will be implemented as well. So always check the Soar on-line help pages for the latest information.

6.3.8.3 Watching working memory elements

There are two different ways to control the printing of working memory changes as Soar runs: one is to set the level of detail printed for WMEs when watching productions, as described in the previous section; the other is to watch WMEs that match a given identifier-attribute-value triplet. The latter is done by `watch`ing `wmes`, optionally followed by `-on|-inc` (since `-off` wouldn't make sense here), and specifying an *action* to add, remove, list or reset the filters; a *type* which specifies whether to apply the filter to wmes as they are either added or removed or both; and a *pattern* which describes the identifier-attribute-value triplet to watch.

Usage: **`watch wmes`** *`action type pattern`*

The four *actions* supported are:

action	effect on the Soar trace
`-add-filter`	add a filter to print wmes that meet the *type* and *pattern* criteria
`-remove-filter`	delete filters for printing wmes that match the *type* and *pattern* criteria
`-list-filter`	list the filters of this *type* currently in use does not use *pattern* arg
`-reset-filter`	delete all filters of this *type* does not use *pattern* arg

The *type* argument is one of:

type	effect on the Soar trace
`-adds`	print info when WME is *added*
`-removes`	print info when WME is *retracted*
`-both`	print info when WME is added or retracted

The `-add-filter` and `-remove-filter` arguments also require a *pattern* to specify which WMEs are of interest. The *pattern* takes the form:

$\{id \mid *\}\ \{attribute \mid *\}\ \{value \mid *\}$

where `*` is a wildcard that matches any string. For any given *pattern*, a working memory element must match exactly for the `watch` command to print any information. Only constants and pre-existing identifiers are may be specified in the *pattern*. This command is somewhat fragile.

Examples

Users can `watch` an *attribute* of a particular object (as long as that object already exists):

```
soar> watch wmes -add-filter -both D1 speed *
```

or print WMEs that retract in a specific state (provided the `state` already exists):

```
soar> watch wmes -add-filter -removes S3 * *
```

or watch any relationship between objects:

```
soar> watch wmes -add-filter -both * ontop *
```

To list all specific WME filtering:

```
soar> watch wmes -list-filter -both
```

To remove all specific WME filtering:

```
soar> watch wmes -reset-filter -both
```

6.3.8.4 Watching learning

Usage: **`watch learning`** `[-print|-noprint|-fullprint]`

As Soar is running, it may create justifications and chunks which are added to production memory. The `watch` command allows users to monitor when chunks and justifications are *created* by specifying one of the following arguments to the `watch learning` command:

argument	effect on the Soar trace
`-print`	print the names of new chunks and justifications when created
`-noprint`	print nothing about new chunks or justifications (default)
`-fullprint`	print entire chunks and justifications when created

Currently these arguments affect both chunks and justifications equally. There is no way to watch the creation of chunks but not justifications, but that may change in the future.

The `watch learning` arguments affect the amount of information printed when a chunk or justification is *created*, but not when they are fired or retracted.

For printing additional information about chunks and justifications after they are created, see the `explain-backtraces` command on page 139.

6.3.8.5 Watching other functions

Three additional arguments to the `watch` command control the tracing of other Soar events, as shown in the following table:

watch		effect on the Soar trace
`backtracing`	`-on`	print backtracing information when a chunk is created
	`-off`	don't print backtracing information (default)
`aliases`	`-on`	echo aliases when they are defined
	`-off`	don't echo aliases when they are defined (default)
`loading`	`-on`	print an asterisk, `*`, for each production loaded and a pound sign, `#`, for each production excised (default)
	`-off`	don't print anything when productions are loaded or excised

6.4 Configuring Soar's Runtime Parameters

This section describes the commands that control Soar's Runtime Parameters. Many of these commands provide options that simplify or restrict runtime behavior to enable easier and more localized debugging. Others allow users to select alternative algorithms or methodologies. Users can configure Soar's learning mechanism; examine the backtracing information that supports chunks and justifications; provide hints that could improve the efficiency of the Rete matcher; limit runaway chunking and production firing; choose an alternative algorithm for determining whether a working memory element receives O-support; and configure options for selecting between mutually indifferent operators. There is also a mode for allowing users to revert to Soar 7 methodology.

The specific commands described in this section are:

Summary

explain-backtraces - Print information about chunk and justification backtraces.

indifferent-selection - Controls how indifferent selections are made.

learn - Set the parameters for chunking, Soar's learning mechanism.

max-chunks - Limit the number of chunks created during a decision cycle.

max-elaborations - Limit the maximum number of elaboration cycles.

multi-attributes - Declare multi-attributes so as to increase Rete matching efficiency.

o-support-mode - Choose experimental variations of o-support.

save_backtraces - Save trace information to explain chunks and justifications.

soar8 - Toggle between Soar 8 methodology and Soar 7 methodology.

waitsnc - Generate a wait state rather than a state-no-change impasse.

6.4.1 `explain-backtraces` [*name*] [-full] [*cond-num*]

The `explain-backtraces` command provides some interpretation of the backtraces generated during the formation of chunks and justifications when impasses are resolved. This command is only meaningful for a particular chunk or justification if the `save-backtraces` variable has been set to `-on` *before* the impasse is resolved (see Section 6.4.8). If no argument is specified, then `explain-backtraces` prints a list of all chunks and justifications for which backtracing information is available.

There are four different ways to call `explain-backtraces`: with no arguments, with a production name, with a production name and an integer condition number, and with a production name and the argument `-full`:

arguments	action
(no args)	print a listing of all productions that can be "explained"
`prodname`	print the conditions and grounds for the named production
`prodname n`	print the grounds for the nth condition of the named production
`prodname` `-full`	print the full backtrace for the named production

The two most useful variants are `explain-backtraces` *`name`* and `explain-backtraces` *`name n`*. The first variant prints a numbered list of all of the conditions for the named chunk or justification, and the *ground* which resulted in its inclusion in the chunk/justification. A *ground* is a working memory element which was tested in the supergoal. Often just knowing which WME was tested in the supergoal is enough to explain why the chunk/justification exists. If not, the second variant, `explain-backtraces` *`name n`*, where n is the number of the condition of interest, can be used to further backtrace through that particular condition to find out why it is included in the chunk/justification. Additionally, the user may wish to call `explain-backtraces` *`chunkname`* `-full` to see the full backtrace of the chunk.

Example

> *comment:* this example needs to be updated for explain-backtraces and save-backtraces. and to Soar7

Note that the `save-backtraces` variable must be set to `-on` prior to the impasse being resolved.

```
soar> explain-backtraces
List of all explained chunks/justifications:
Have explanation for chunk-3
Have explanation for chunk-2
Have explanation for chunk-1
Have explanation for justification-5
Have explanation for justification-4
Have explanation for justification-3
Have explanation for justification-2
Have explanation for justification-1

soar> explain-backtraces chunk-1
sp {chunk-1
   (state <s1> ^object-dynamic <n3> ^problem-space <p1> ^desired <d4>
         ^operator <o1> + ^object-dynamic <n2>)
   (<n3> ^clear yes ^object-static <b3>)
   (<s1> ^object-dynamic <n1>)
   (<n1> ^clear yes ^object-static <b2>)
   (<p1> -^default-state-copy no ^two-level-attributes object-dynamic
         ^name blocks-world ^default-operator-copy no)
   (<d4> ^ontop-count 3 ^object-dynamic <d3>
         ^object-dynamic { <> <d3> <d1> }
         ^object-dynamic { <> <d1> <> <d3> <d2> })
   (<b3> ^type block)
   (<d3> ^object-static <b3> ^ontop <b2>)
   (<d1> ^object-static <b2>)
   (<o1> ^moving-block <b2> ^destination <b1>)
   (<d1> ^ontop <b1>)
   (<n2> ^object-static <b1> ^ontop <t1>)
   (<d2> ^object-static <b1>)
   (<d2> ^ontop <t1>)
-->
   (<s2> ^operator <o2> >)
}

  1 :  (state <s1> ^object-dynamic <n3>) Ground : (S4 ^object-dynamic N2)
  2 :  (<n3> ^clear yes)                Ground : (N2 ^clear yes)
  3 :  (<s1> ^object-dynamic <n1>)      Ground : (S4 ^object-dynamic N3)
  4 :  (<n1> ^clear yes)                Ground : (N3 ^clear yes)
  5 :  (<s1> ^problem-space <p1>)       Ground : (S4 ^problem-space P2)
  6 :  (<p1> -^default-state-copy no)   Ground :
                                          (P2 -^default-state-copy no)
  7 :  (<p1> ^two-level-attributes object-dynamic)
                                        Ground : (P2 ^two-level-attributes
                                          object-dynamic)
  8 :  (<p1> ^name blocks-world)        Ground : (P2 ^name blocks-world)
  9 :  (<p1> ^default-operator-copy no) Ground :
                                          (P2 ^default-operator-copy no)
 10 :  (<s1> ^desired <d4>)             Ground : (S4 ^desired D2)
 11 :  (<d4> ^ontop-count 3)            Ground : (D2 ^ontop-count 3)
 12 :  (<n3> ^object-static <b3>)       Ground : (N2 ^object-static B4)
 13 :  (<b3> ^type block)               Ground : (B4 ^type block)
 14 :  (<d4> ^object-dynamic <d3>)      Ground : (D2 ^object-dynamic D3)
 15 :  (<d3> ^object-static <b3>)       Ground : (D3 ^object-static B4)
 16 :  (<d3> ^ontop <b2>)               Ground : (D3 ^ontop B5)
 17 :  (<n1> ^object-static <b2>)       Ground : (N3 ^object-static B5)
 18 :  (<d4> ^object-dynamic { <> <d3> <d1> })
                                        Ground : (D2 ^object-dynamic D4)
 19 :  (<d1> ^object-static <b2>)       Ground : (D4 ^object-static B5)
 20 :  (<s1> ^operator <o1> +)          Ground : (S4 ^operator O15 +)
 21 :  (<o1> ^moving-block <b2>)        Ground : (O15 ^moving-block B5)
 22 :  (<o1> ^destination <b1>)         Ground : (O15 ^destination B6)
 23 :  (<d1> ^ontop <b1>)               Ground : (D4 ^ontop B6)
 24 :  (<s1> ^object-dynamic <n2>)      Ground : (S4 ^object-dynamic N4)
 25 :  (<n2> ^object-static <b1>)       Ground : (N4 ^object-static B6)
 26 :  (<d4> ^object-dynamic { <> <d1> <> <d3> <d2> }) Ground : (D2
       ^object-dynamic D5)
 27 :  (<d2> ^object-static <b1>)       Ground : (D5 ^object-static B6)
 28 :  (<n2> ^ontop <t1>)               Ground : (N4 ^ontop T2)
 29 :  (<d2> ^ontop <t1>)              Ground : (D5 ^ontop T2)

soar> explain-backtraces chunk-1 2
Explanation of why condition  (N2 ^clear yes) was included in chunk-1

Production justification-17 matched
    (N2 ^clear yes) which caused
production
  default*selection*select*success-evaluation-becomes-best-preference
  to match
    (E4 ^symbolic-value success) which caused
A result to be generated.

soar>
```

6.4.2 **indifferent-selection** [-first|-last|-random|-ask]

The `indifferent-selection` command allows the user to set which option should be used to select between operator proposals that are mutually indifferent in preference memory.

By default, Soar will select the `-first` of the mutually indifferent augmentations, and create the corresponding element in working memory. "First" refers to a list internal to Soar; the ordering of the augmentations is arbitrary but deterministic, so that if you run Soar repeatedly, `-first` will always make the same decision. Similarly, `-last` chooses the last of the tied objects from the internal list. The options, `-first` and `-last` are in contrast to `-random`, which also makes an arbitrary decision, but this will not generally be the same decision for each repeated run.

A fourth method for deciding among indifferent operator proposals currently works only on Unix systems when running without the Tcl-Soar Interface. This is because a Wish or Tcl Console window is currently required. The `-ask` method prompts the user to make the decision. This option should be restored on all Soar platforms with the next release, so it is documented here, but users are alerted that it doesn't work as advertised for Soar 8.2.

If no argument is provided, `indifferent-selection` will display the current setting. With an argument, it sets `indifferent-selection` to the given value:

argument	effect on selection
`-first`	select the first indifferent object from Soar's internal list (default)
`-last`	select the last indifferent object from Soar's internal list
`-random`	select randomly
`-ask`	ask the user to choose

Example

```
soar> indifferent-selection
-first
soar> indifferent-selection -random
```

6.4.3 `learn` [-on|-off|-except|-only] [-list] \ [-all-levels | -bottom-up]

The `learn` command controls the parameters for chunking (Soar's learning mechanism).

With no arguments, this command prints out the current learning environment status. If arguments are provided, they will alter the learning environment as described in the table below.

learn	effect on chunking behavior
`-on`	turn chunking on (default)
`-except`	chunking is on, *except* as specified by RHS `dont-learn` actions
`-only`	chunking is on *only* as specified by RHS `force-learn` actions
`-off`	turn chunking off
`-list`	prints listings of dont-learn and force-learn states
`-all-levels`	build productions whenever a subgoal returns a result (default)
`-bottom-up`	build productions only for subgoals that have not yet had any subgoals with chunks built

The `-except` and `-only` flags interact with `dont-learn` and `force-learn` production actions, as described in Section 3.3.6.12 on page 67.

With the `-on` flag, chunking is on all the time. With the `-except` flag, chunking is on, but Soar will not create chunks for states that have had RHS `dont-learn` actions executed in them. With the `-only` flag, chunking is off, but Soar will create chunks for only those states that have had RHS `force-learn` actions executed in them. With the `-off` flag, chunking is off all the time.

The `-only` flag and its companion `force-learn` RHS action allow Soar developers to turn learning on in a particular problem space, so that they can focus on debugging the learning problems in that particular problem space without having to address the problems elsewhere in their programs at the same time. Similarly, the `-except` flag and its companion `dont-learn` RHS action allow developers to temporarily turn learning off for debugging purposes. These facilities are provided as debugging tools, and do not correspond to any theory of learning in Soar.

The `-list` flag produces a listing of the identifiers of all states that have been set to "don't learn" and all states that have been set to "force learn".

The `-all-levels` and `-bottom-up` flags are orthogonal to the `-on`, `-except`, `-only`, and `-off` flags, and so, may be used in combination with them. Recall from Chapter

4 that with bottom-up learning, chunks are learned only in states in which no subgoal has yet generated a chunk. In this mode, chunks are learned only for the "bottom" of the subgoal hierarchy and not the intermediate levels. With experience, the subgoals at the bottom will be replaced by the chunks, allowing higher level subgoals to be chunked.

Learning can be turned on or off at any point during a run.

Example

```
soar> learn -on

soar> learn
Current learn settings:
-on
-all-levels
```

6.4.4 `max-chunks [n]`

The `max-chunks` command is used to limit the maximum number of chunks that may be created during a decision cycle. The initial value of this variable is 50; allowable settings are any integer greater than 0.

The chunking process will end after `max-chunks` chunks have been created, *even if there are more results that have not been backtraced through to create chunks.*, and Soar will proceed to the next phase. A warning message is printed to notify the user that the limit has been reached.

This limit is included in Soar to prevent getting stuck in an infinite loop during the chunking process. This could conceivably happen because newly-built chunks may match immediately and are fired immediately when this happens; this can in turn lead to additional chunks being formed, etc. If you see this warning, something is seriously wrong; Soar is unable to guarantee consistency of its internal structures. You should not continue execution of the Soar program in this situation; stop and determine whether your program needs to build more chunks or whether you've discovered a bug (in your program or in Soar itself).

Example

```
soar> max-chunks
50
soar> max-chunks 100
soar> max-chunks
100
```

6.4.5 `max-elaborations` [*n*]

> *comment:* somewhere, maybe appendix C, i'd like to see an example of hitting the limit

The `max-elaborations` command is used to limit the maximum number of elaboration cycles allowed during an elaboration phase. The inital value of this variable is 100; allowable settings are any integer greater than or equal to 1.

The elaboration phase will end after `max-elaboration` cycles have completed, *even if there are more productions eligible to fire or retract*; and Soar will proceed to the next phase. A warning message is printed to notify the user that the limit has been reached and Soar has moved on to the next phase. This limits the total number of cycles of parallel production firing but does not limit the total number of productions that can fire during elaboration.

This limit is included in Soar to prevent getting stuck in infinite loops (such as a production that repeatedly fires in one elaboration cycle and retracts in the next); if you see the warning message, it may be a signal that you have a bug your code. However some Soar programs are designed to require a large number of elaboration cycles, so rather than a bug, you may need to increase the value of `max_elaborations`.

Regardless of the reason that your program has hit the maximum number of elaboration cycles, if you see the warning message it means that your program is not working as intended. You should either fix a bug or increase the limit.

Example

```
soar> max-elaborations
100
soar> max-elaborations 20
soar> max-elaborations
20
```

6.4.6 `multi-attributes` [*attribute* [*n*]]

The `multi-attributes` command is used before productions are loaded to declare attributes that are multi-attributes, i.e., they will routinely have multiple values. The information provided by this command is used by the Rete matcher, which reorders the conditions of productions heuristically in an attempt to increase the speed of the matching process.

The specified *attribute* is the name of an attribute that may take on multiple values; the optional *n* is an integer (greater than 0), indicating an upper limit on the number of expected values that will appear for the attribute. If `n` is not specified, the value 10 is used for each declared multi-attribute. More informed values will tend to result in greater efficiency.

This declaration is not required. It also has *no effect* on the contents of working memory. Instead, the `multi-attributes` command is used only to increase the efficiency of the matcher so that Soar can have heuristic information about multi-attributes available when it internally reorders production conditions in the Rete net.

With no arguments, the `multi-attributes` command prints a listing of the currently declared multi-attributes.

Example

```
soar> multi-attributes thing 4
soar> multi-attributes ontop 3
```

Notes Note that `multi-attributes` declarations must be made *before* productions are loaded into production memory.

6.4.7 `o-support-mode` [0 | 1 | 2]

The `o-support-mode` command is used to control the way that O-support is determined for preferences. See Section ?? for a discussion of the default scheme.

The `o-support-mode` value must be one of three values:

value	effect
`0`	O-support works as described earlier in this manual
`1`	O-support works as described in this manual, *but* a message is printed whenever the alternative scheme would have made a difference in the Soar program.
`2`	The alternative O-support scheme is used.

In the alternative scheme, O-support is determined according to the following rules:

```
IF the preference is for an operator augmentation of a state,
THEN the preference gets I-support
ELSE IF the production creating the preference tests an ^operator
        augmentation on a state, OR tests an acceptable preference
        for an ^operator augmentation on a state, OR creates an
        additional acceptable preference for an ^operator augmentation
        on a state,
     THEN the preference gets O-support
     ELSE the preference gets I-support
```

The main difference in the alternative O-support scheme is that O-support determination is based entirely on the structure of the production that creates the preference, and is not based on working memory. (In the default scheme, O-support is based on whether the preference is linked to the operator or the state, but this linkage might be created *after* the production has fired.)

A few specific details:

1. Operator proposal preferences always get I-support.
2. Search control preferences (desirability preferences for operators) always get I-support.
3. Changes to working memory resulting from operator elaboration get O-support. (This is the most contentious change under the alternative scheme.)
4. Structures shared between the operator and the state are no longer "special" (With the default scheme, there is some ambiguity over whether shared structure is considered to "test the operator".)

Differences and similarities between the two schemes:

- For most "normal" O-support calculations, things don't change, the calculation will just be simpler (for both user and machine) under the alternative scheme.
- Operator proposal, operator application, state elaboration, operator selection all will come out the same, if there are no usual shared structures involved.
- Operator elaboration is definitely different.

6.4.8 save_backtraces

comment: this could use a pointer to an explanation of what a backtrace is

The `save_backtraces` variable is a toggle that controls whether or not backtracing information (from chunks and justifications) is saved. The initial value of this variable is `off`; it may also be set to `on`.

When `save_backtraces` is set to "`off`", backtracing information is not saved and explanations of the chunks and justifications that are formed can not be retrieved. When `save_backtraces` is set to "`on`", backtracing information can be retrieved by using the `explain-backtraces` command (described on page 139). Saving backtracing information may slow down the execution of your Soar program, but it can be a very useful tool in understanding how chunks are formed.

Example

```
soar> set save_backtraces
off
soar> set save_backtraces on
soar> set save_backtraces
on
```

6.4.9 **soar8** `[-on | -off]`

Significant architectural changes were made to Soar between version 7 and version 8. The `soar8` command allows users to revert to the Soar 7 methodology in order to run older Soar programs. Both production memory and working memory must be empty to toggle between Soar 7 mode and Soar 8 mode. The `soar8` command with no arguments returns the current mode; the default is that `soar8` is `-on`. This command may not be available in future versions of Soar, as maintaining both architectures in the same source code is cumbersome and may lead to problems as more changes are introduced.

soar8	effect on Soar
`-on`	use the Soar 8 methodology (default)
`-off`	use the Soar 7 methodology

Users are referred to the *Release Notes for Soar 8.2* for detailed information on the differences between the two architectures.

6.4.10 **waitsnc** `[-on | -off]`

In some systems, esepcially those that model expert (fully chunked) knowledge, a state-no-change may represent a *wait state* rather than an impasse. The `waitsnc` command allows users to anticipate this situation and cause Soar to generate a wait state whenever a `state-no-change` impasse would otherwise occur. When `waitsnc` is set to `-on`, Soar will automatically generate a wait state rather than a `state-no-change` impasse. The decision cycle will repeat (and the decision cycle count is incremented) but no `state-no-change` impasse (and therefore no substate) will be generated.

waitsnc	effect on Soar
`-on`	turns state-no-change impasse into "wait" mode
`-off`	state-no-change generates impasse (default)

6.5 File System I/O Commands

This section describes commands which interact in one way or another with operating system input and output, or file I/O. Users can save/retrieve information to/from files, redirect the information printed by Soar as it runs, and save and load the binary representation of productions. The specific commands described in this section are:

Summary

command-to-file - Evaluate a command and print its results to a file.

directory functions - `chdir, cd, dirs, popd, pushd, pwd, topd`

echo - Print a string to the current `output-strings-destination`.

log - Record all user-interface input and output to a file.

output-strings-destination - Redirect the Soar output stream.

rete-net - Save the current Rete net, or restore a previous one.

source - Load and evaluate the contents of a file.

The `source` command is used for nearly every Soar program. The directory functions are important to understand so that users can navigate directories/folders to load/save the files of interest. Any Soar application that includes a graphical interface or other simulation environment will require the use of `echo` and `output-strings-destination`.

6.5.1 `command-to-file` {*cmd*} *filename* [-new|-existing]

The `command-to-file` command evaluates a specified Soar command and redirects the output of the command to the specified file. The file specified may be a new file, an existing file to be overwritten, or an existing file to be added to; if neither `-new` or `-existing` is specified, the file will be opened as a "new" file, overwriting the file if it already exists.

argument	action
cmd	The command to be evaluated
filename	The name of the file to open
`-new`	Open the named file, overwriting it if it already exists (default)
`-existing`	Open the named file, appending to it if it already exists

To process a multiple-word Soar command using `command-to-file`, the multiple-word command and all its arguments must be enclosed in either curly braces, { and }, or double quotes. If double quotes are used, the string will first be scanned for Tcl variable references (indicated by a $ sign) and embedded command evaluations (anything enclosed in square brackets, `[]`), before the command is evaluated.

Examples

```
soar> command-to-file {print -all} my-task.soar
soar> command-to-file {print -chunk} my-chunks.soar
soar> command-to-file firing-counts -existing my-firing-counts.save
```

The first example will save all productions (including chunks, justifications, and default productions) to the file `my-task.soar`; the second example will save only chunks to the file `mychunks.soar`. (Both files are formatted such that they could be read back in to reload the productions.) The third example appends the firing-counts data to the file `my-firing-counts.save`; note that the curly braces were not required to group the command string, but including them would work fine too.

6.5.2 Directory/Folder functions: `chdir`, `cd`, `dirs`, `popd`, `pushd`, `pwd`, `topd`

There are several commands for changing and displaying the current *directory* or *folder*: `chdir`, `cd`, `pwd`. The commands `chdir` and `cd` are synonyms. They take as an optional argument the relative or absolute path of the directory that should become the current working directory; if no argument is given, the `$HOME` directory is used if it exists. The command `pwd` prints the full pathname of the current working directory.

There is an alternative approach to managing directories, which involves maintaining a *directory stack*. Using this approach, the "top" of the stack is the current directory, returned by the `topd` command; new directories are "pushed" onto the stack with the `pushd` ***dirname*** command; and when a directory is "popped" off the stack with `popd`, the next directory in the stack becomes the new current directory. The `dirs` command lists the entire stack.

command	action
`cd [`***dirname***`]`	changes the process's current working directory
`chdir [`***dirname***`]`	changes the process's current working directory
`pwd`	prints the current working directory. This is the directory from which files will be loaded.
`dirs`	lists the agent's directory stack.
`popd`	pop a directory off of the directory stack and change to the directory now at the top of the stack.
`pushd` ***dirname***	push the current directory onto the directory stack and change to the given directory.
`topd`	lists the top directory on the directory stack.

A process can have at most, one current directory location. This has implications for running Soar with more than one agent. If the current directory is changed for one agent, it is changed for *all* agents. This is a situation where using a directory stack is particularly useful. Each agent has its own directory stack which is not modified when other agents `pushd` ***dirname***, `popd`, or `cd` ***dirname*** to other locations. The *current directory* will be modified, but the stack will not be. So as long as agents maintain their own directory stack using `pushd`, `popd`, `topd` and `dirs`, it will always be possible to ensure that the agents can set the proper current directory before invoking any commands that read or write files.

6.5.3 echo `[-nonewline] [`***args***`]`

The `echo` command prints its arguments back to the current output stream, which is normally *stdout* or the user interface, but can be set to a variety of channels. By default, a newline is printed after the echoed arguments, but if the optional `-nonewline` argument is specified first, no newline character is printed.

Users wanting to print variables and data to the screen, should use `echo` rather than the Tcl "`puts`" command. The `echo` command gets redirected to the appropriate channel according to `output-strings-destination`; the `puts` command does not. (see Section 6.5.5).

The arguments to be echoed back may be enclosed in curly braces, { and }, to control interword spacing, for example, but in most cases, these delimiters are not needed.

Examples

```
soar> echo Test load number 5
Test load number 5

soar> echo {I  want   space}
I  want   space
```

6.5.4 log [-new *filename* | -existing *filename* | -off | -query | -add *string*]

The `log` command allows users to save all user-interface input and output to a file. When Soar is logging to a file, everything typed by the user and everything printed by Soar is written to the file (in addition to the screen).

With no arguments, the `log` commands prints the current logging status ("open" or "closed" and the name of the file that is open if logging is on). The optional arguments are described in the following table:

log	**Action**
`-new` *filename*	Begin logging to the named file, overwriting it if it already exists
`-existing` *filename*	Begin logging to the named file, appending to it if it already exists
`-off`	Close the current log file
`-query`	Print the current log status
`-add` *string*	Add the given *string* to the open file

Examples

This initiates logging and places the record in foo.log:

```
soar> log -new foo.log
```

This appends log data to an existing foo.log file:

```
soar> log -existing foo.log
```

This terminates logging and closes the open log file:

```
soar> log -off
```

6.5.5 **`output-strings-destination`** `[-pop | -push` ***`action`***`]`

The `output-strings-destination` command redirects the Soar text output stream. This is useful for applications which need to change where printed results are placed. Printed output is normally sent to standard output. However, if a Graphical User Interface (GUI) is being used in place of the command line inter- preter, as is the case with the Tcl-Soar Interface, then printed output should appear in the GUI. GUIs are composed of elements called *widgets* and printed output would be directed to a *text widget*, or its *command procedure*.

Printed output is normally sent to standard output. Printed output can be sent to any other open file descriptor as well, such as an open file or pipe. It is also possible that the user is not interested in an agent's printed output. In that case, the printed output can be discarded – which results in faster processing for the agent as well.

Some Soar commands *return* results and some *print* results. If the user wishes to have printed results returned so that they can be saved for later use (i.e., saved to a variable), then this command can cause the printed output to be appended to the returned result.

The printing facility is implemented as a stack, so newly installed print redirections are in force until popped off the stack. This is done to allow easy transient redirection, supporting the restoration of prior printing contexts after completing a printing task. Hence, there are two primary functions: `-push` *`action`* and `-pop`. The `-pop` function takes no additional arguments as it serves only to pop the print-redirection stack to re-establish the prior printing context. The `-push` *`action`* takes one of the following additional arguments:

`-push` arguments	effect
`-append-to-result`	Instead of simply printing command output, append the output string to the result returned by the command.
`-channel` *`open-fid`*	Redirect printing to the open channel (file or pipe) denoted by *open-fid*
`-discard`	Ignore prints (similar to redirecting to /dev/null in the Bourne shell, sh)
`-procedure` *`proc-name`*	Redirect printing to the Tcl procedure named *proc-name*. (Tk widgets are also procedures, so this action can send output to widget procedures, or any other Tcl procedure).
`-text-widget` *`widname`* *`interp-name`*	Redirect printing to the text widget named *widname* in the current interpreter. If *interp-name* is specified, then printing is redirected to the text widget named *widname* in the interpreter named *interp-name*.

Example This example redirects prints to the text widget ".text":

```
soar> output-strings-destination -push -text-widget .text
```

This example causes all printing to be supressed:

```
soar> output-strings-destination -push -discard
```

This example removes the most recently added print destina- tion:

```
soar> output-strings-destination -pop
```

Notes

This command only affects the printing generated by Soar commands such as `write` and `echo`. It does *not* affect printing done by the Tcl commands such as `puts`.

6.5.6 **rete-net** {-save | -load} *filename*

The `rete-net` command saves the current Rete net to a file or restores a Rete net previously saved. The Rete net is Soar's internal representation of production memory; the conditions of productions are reordered and common substructures are shared across different productions. This command provides a fast method of saving and loading productions since a binary format is used. `rete-net` files are portable across platforms that support Soar. The action must be one of the following:

argument	action
`-save` *filename*	Save the Rete net in the named *filename*.
`-load` *filename*	Load the named *filename* into the Rete network.

The Rete network cannot be saved while there are justifications present. These can be eliminated by using the `init-soar` command.

In order to load productions stored in binary form, working memory and production memory must both be empty. Working memory can be emptied by using the `init-soar` command. Production memory can be emptied by using the `excise -all` command. If working memory or production memory is not empty, an error message is issued.

Example

```
soar> rete-net -save my-program-with-chunks.rete
soar> init-soar
soar> excise -all
soar> rete-net -load my-program-with-chunks.rete
```

Notes

If the filename contains a suffix of ".Z", then the file is compressed automatically when it is saved and uncompressed when it is loaded. Compressed files may not be portable to another platform if that platform does not support the same uncompress utility. File compression in the `rete-net` command is not available on Macintoshes.

6.5.7 **source** ''*filename*''

The `source` command is a core Tcl command which *loads* the file, *`filename`*: it opens the file and sequentially evaluates any commands included in the file as if they had been directly entered by the user. The *`filename`* can be a simple filename, in which case it is loaded from the current working directory; or it can specify the full pathname to access files located in another directory. Users should refer to Tcl documentation for the format for specifying a full pathname.

The `source` command is typically used to load a file that contains productions for a Soar program, but files may contain any of the commands in this chapter, including the `source` command itself, and often include core Tcl commands.

Example

```
soar> source "blocks.soar"
**********
soar>
```

```
soar> source "blocks.soar"
#*#*#*#*#*#*#*#*#*#*
soar>
```

Notes

If *`filename`* is a simple string with no special characters, the double quotes may be omitted.

The printing of asterisks and pound signs for each production loaded (as defined by the `sp` command) and excised (redefined using `sp` or removed from production memory using `excise`) may be turned off by using the `watch loading` command; this is described in Section 6.3.8.

If the file being `source`'d includes commands to `source` other files, then users should either `pushd` to the directory, or specify full pathnames whenever the `source` command is issued.

6.6 Soar I/O Commands

This section describes the commands used to manage Soar's Input/Output (I/O) system, which provides a mechanism for allowing Soar to interact with external systems, such as a computer game environment or a robot. Soar I/O is accomplished via *input functions* and *output functions* which are managed using the `io` command. These functions make calls to `add-wme` and `remove-wme` to add and remove elements to the `io` structure of Soar's working memory. See section 6.6 for a functional description of Soar I/O and *The Soar Advanced Applications Manual* for more information on creating input and output functions. The demo file `demos/soar-io-using-tcl.tcl` gives examples for doing I/O in Soar using Tcl scripts.

The specific commands described in this section are:

Summary

add-wme - Manually add an element to working memory.

io - Register or cancel routines for managing Soar's input and output links.

remove-wme - Manually remove an element from working memory.

These commands are used only when Soar needs to interact with an external environment.

6.6.1 **`add-wme`** *`id`* `[^]`*`attribute value`*

The `add-wme` command adds a new working memory element directly to working memory, bypassing the usual evaluation processes of Soar. This command is provided for use in Soar input functions; although there is no programming enforcement, `add-wme` should only be called from registered input functions to create working memory elements on Soar's input link. `add-wme` returns the timetag of the new WME, which can be used later by `remove-wme` to remove the WME when it is no longer needed.

The `id` must be an identifier that already exists in working memory. The `attribute` and `value` may be any symbols; an asterisk (`*`) indicates that Soar should create a new identifier for the attribute or value (see page 36). Any symbol generated by Soar has the form of a single letter followed by an integer which will make the symbol unique. The carat symbol that precedes the attribute is optional.

Note that because the `id` must already exist in working memory, the WME that you are adding will be attached (directly or indirectly) to the top-level state. Input functions should only add working memory elements to the ^`input-link`, which is a structure on the top-level `io` attribute. As with other WME's, any WME added via a call to `add-wme` will automatically be removed from working memory once it is no longer attached to the top-level state.

Example

Typically, `add-wme` will not be invoked from Soar's command line, except perhaps as a debugging aid, but the syntax is the same whether it is typed at the user prompt or as part of an input function. Note that prior to issuing this command, the user must have a valid `identifier`, such as the symbol I2 for the ^`input-link`. The method for obtaining the Soar `input-link` and `output-link` identifiers is described in the demo file `demos/soar-io-using-tcl.tcl` and in *The Soar Advanced Applications Manual.*

```
soar> add-wme I2 block1 on-table
```

Warning

The `add-wme` command may have weird side effects (possibly even including system crashes) when used outside its intended role. It should only be invoked during the Input Phase of Soar's execution cycle. Also note that Soar's chunking mechanism can't backtrace through working memory elements created via `add-wme`.

6.6.2 **io** -add -input *script* [*id*]
io -add -output *script* *id*
io -delete {-input|-output} *id*
io -list {-input | -output}

Recall from section 6.6 that Soar I/O is accomplished via *input functions* and *output functions* which manipulate augmentations of the top-level ^`io` attribute. Input functions are called at the start of every execution cycle to add and remove working memory elements on the ^`io.input-link`. Output functions are processed at the end of every execution cycle in response to changes made to the ^`io.output-link`. Input and output functions are *registered* with Soar using the `io` command. The `io` command allows users to add and delete input and output functions, and list all currently registered functions.

6.6.2.1 Adding I/O functions.

6.6.2.2 Deleting I/O functions.

6.6.2.3 Listing I/O functions.

The method for obtaining the Soar `input-link` and `output-link` identifiers is described in the demo file `demos/soar-io-using-tcl.tcl` and *The Soar Advanced Applications Manual.*

6.6.3 `remove-wme` *n*

The `remove-wme` command removes the working memory element with the given timetag, *n*, which must be a positive integer matching the timetag of an existing working memory element. This command is provided primarily for use in Soar input functions; although there is no programming enforcement, `add-wme` should only be called from registered input functions to create working memory elements on Soar's input link.

Although this command is able to remove any working memory element or preference — including those automatically created by the Soar architecture and via the evaluation of preferences — recklessly removing working memory elements is likely to have weird side effects, including system crashes.

Example

Typically, `remove-wme` will not be invoked from Soar's command line, except perhaps as a debugging aid, but the syntax is the same whether the command is typed at the user prompt or is part of an input function. Note that prior to issuing this command, the user must have determined the timetag of the WME which is to be removed, usually a WME that was added with `add-wme`. The method for managing the Soar `io` structure is described more fully in the demo file `demos/soar-io-using-tcl.tcl` and in *The Soar Advanced Applications Manual.*

```
soar> remove-wme 45
```

Warning

The `remove-wme` command may have weird side effects (possibly even including system crashes) when used outside its intended role. It should only be invoked during the Input Phase or Output Phase of Soar's execution cycle. It should *never* be invoked from the action side of productions. `remove-wme` may have adverse affects on chunking. Removing input working memory elements or state, operator, or impasse working memory elements may have weird side effects, including system crashes.

6.7 Miscellaneous

comment: this section still needs to be rewritten...

The specific commands described in this section are:

Summary

alias

default rules

predefined aliases

soarnews

soar.tcl file

unalias

variables

version

6.7.1 `alias [ name [definition | -off ]]`

The `alias` command displays and defines Soar aliases. When called with no arguments, it displays a listing of the currently defined aliases; when called with a specific name, it displays the alias currently defined for that name; when called with a name and a definition, it defines a new alias. When called with a name and `-off`, it removes the named alias.

The `name` can be one or more alphanumeric characters, but must begin with a letter.

The `definition` can be any Soar command, including another alias, and including any number of arguments to that command. The definition may be an arbitrarily complex single command. If more complex (multi-command) aliases are desired, the the Tcl `proc` command can be used to define a new procedure (see Chapter 5.)

Version 7.0.3 comment: probably has the same curly braces/double quotes tradeoffs as other commands, but I'm not certain.

KJC: yep

Aliasing of a command may be turned off by using `alias name -off` or `unalias name`. (The latter is provided for similarity to the Unix aliasing capability.)

> *Version 7.0.3 comment:* I'm confused by how this command is implemented; probably due to Tclisms.
>
> Seems to me that "alias" with no args should print a listing of all the aliases and their definitions, just as in Unix. But instead, it just lists the names of aliases that have been defined.
>
> When I look at the actual definitions, I suddenly understand – It's way more complicated than what I typed. But I don't understand why this is necessary. (Why can't it just be saved as a string, the way (I thought) Unix does it?)

Example

```
soar> alias r3 run 3 d
alias r3 run 3 d
soar> r3
r3

Moving Block: b to: c
     9:    O: O39 (move-block)
Moving Block: b to: a
    10:    O: O43 (move-block)
Moving Block: c to: b
    11:    O: O46 (move-block)

soar> alias
alias
? ea ec et exit fc i l m p ps q r r3 s w wmes
soar> alias r3
alias r3
r3:
      if {$args == ""} {
  run 3 d
      } else {
  eval run 3 d $args
      }
```

> *Version 7.0.3 comment:* I'm fairly certain that this change is what broke the ability to use multiple commands with an alias – e.g., you used to be able to define an alias like: "run 3 d; print s1". I can "eval run 3 d" and "eval run 3 d; print s1". I'm guessing this other approach has to do with some sort of efficiency concern, but I wonder if the tradeoff is worth it. (Not being able to define multi-command aliases, and not being able to print a simple definition of an alias.
>
> KJC: I don't think it's on the list for 7.1 either. I'd like to leave it and see how it plays with users. Since alias is implemented in TCl and not C, it will be simple to have users update it later.

6.7.2 Defining command aliases

The user may define his or her own aliases for any of the Soar commands provided here; see the `alias` command on page 161 for more information. Frequently used aliases may be defined in a file that is loaded using the `source` command at start-up time; see Appendix ?? on page ?? for more information.

6.7.3 `source $default`

The command `source $default` provides a simple means of loading Soar's default knowledge (described in Section ?? and in Appendix ??) without knowing the path-name to the file of default rules.

Example

```
soar> source $default
***************************************************************
************************************

soar>
```

Notes

The dollar sign indicates that `default` is a Tcl variable. If you have any problems with this command, it is likely that this variable is not being resolved properly. If this should happen, you'll have to use the full pathname to load the default productions; report the problem to your local Soar administrator.

If you can run Soar, you should always be able to source the default productions with the longer command: `source $soar_library/default.soar`.

6.7.4 `predefined aliases`

6.7.5 `soarnews`

> *Version 7.0.3 comment:* this command should probably be changed to print even more info, for example, where to get documentation and how to join soar-group.
>
> update example output for final release.

The `soarnews` command prints information about the current release.

Example

```
soar> soarnews
News for Soar version 7.0.2. TCL TK


Bugs and questions should be sent to soar-bugs@cs.cmu.edu
The current bug-list may be obtained by sending mail to
soarhack@cs.cmu.edu with the Subject: line "bug list".

This software is in the public domain, and is made available AS IS.
Carnegie Mellon University, The University of Michigan, and
The University of Southern California/Information Sciences Institute
make no warranties about the software or its performance, implied
or otherwise.

Type "help" for information on various topics.
Type "quit" to exit.  Use ctrl-c to stop a Soar run.
Type "soarnews" for news.
Type "version" for complete version information.

soar>
```

6.7.6 The **`$soar_library/soar.tcl`** file

The `$soar_library/soar.tcl` file is loaded for all users at a local site, and can be reconfigured by the local Soar administrator. This file is used to load local aliases; it also contains the Tcl code that implements many of the user-interface functions. It is also the appropriate place for platform-dependent code.

Individual Soar users have no control over this file.

6.7.7 **`unalias`** `[ name [definition | -off ]]`

6.7.8 **`version`**

The `version` command prints the version number of Soar. This is useful information to know when asking for help or reporting a bug.

Example

```
soar> version
7.0.2. TCL TK
```