

# **The Soar 8 Tutorial**

John E. Laird  
University of Michigan  
February 23, 2001

This tutorial is the culmination of work by many people. Eaters was developed by Randy Jones and then refined and updated by Scott Wallace. Mazin As-Sanie developed TankSoar based on Tag-Soar, which was originally developed by Mike van Lent. Soar 8 is based changes to Soar 7 suggested by Bob Wray's thesis work, but also includes changes to the decision cycle suggested by Randy Jones, and refined by Bob Wray, Karen Coulter, and Mike van Lent. Karen Coulter brought all of the pieces of Soar 7 and Soar 8 together and is responsible for the final integration and release of the Soar 8 code. Brad Jones developed visual-Soar. Jessica Laird tested the tutorial and pointed out ambiguities and complexities in earlier versions.

## Introduction

This is a guide for learning to create software entities in Soar, version 8. It assumes no prior knowledge of Soar or computer programming.

The goals of this document are:

- Introduce you to the basic operating principles of Soar.
- Teach you how to run Soar programs and understand what they do.
- Teach you how to write your own Soar programs.

This tutorial takes the form of a sequence of lessons. Each lesson introduces concepts one by one and gives you a chance to use them creating Soar entities. Each lesson builds on the previous ones, so it is important to go through them in order. To make the best use of this tutorial, we recommend that you read the tutorial, do the exercises, run the programs, and write your own Soar entities. The programs are available at <http://ai.eecs.umich.edu/soar/tutorial.html>. Please use the most recent versions. Although the tutorial is long, you should be able to work through it quickly.

What is Soar? We call Soar a unified architecture for developing intelligent systems. That is, Soar provides the fixed computational structures in which knowledge can be encoded and used to produce action in pursuit of goals. In many ways, it is like a programming language, albeit a specialized one. It differs from other programming languages in that it has embedded in it a specific theory of the appropriate primitives underlying symbolic reasoning, learning, planning, and other capabilities that we hypothesize are necessary for intelligent behavior. Soar is not an attempt to create a general purpose programming language. You will undoubtedly discover that some computations are more appropriately encoded in a programming language such as C, C++, or Java. Our hypothesis is that Soar is appropriate for building autonomous entities that use large bodies of knowledge to generate action in pursuit of goals.

This tutorial is specific to Soar 8, which has significant changes from earlier versions of Soar. These changes improve Soar's ability to maintain the consistency of its reasoning while interacting with dynamic environments.

The tutorial comes in five parts. Part I uses a Pacman like game called Eaters to introduce the basic concepts of Soar. After working through Part I, you should be able to write simple Soar programs. Part II uses a grid-based tank game called Tank-Soar. Part II concentrates on Soar's subgoal mechanism as it is used for task decomposition. Part III uses the Water Jug and Missionaries and Cannibals problems to introduce internal problem solving and search. Part IV uses the same tasks as Part III to introduce look-ahead planning and learning. Part V covers the Soar system that plays Quake II. The software for Part V will be available soon.

Soar has its own editor, called Visual Soar, which is highly recommended for used in developing Soar programs. Visual Soar is available from the Soar homepage.

To get a more abstract overview of Soar and the associated research issues, you should read "A Gentle Introduction to Soar." To get more details on the exact operation of Soar, you should read "The Soar Manual." The manual is available on our web site at <http://ai.eecs.umich.edu/~soar/docs.html>, and we would be happy to send the other paper to you.

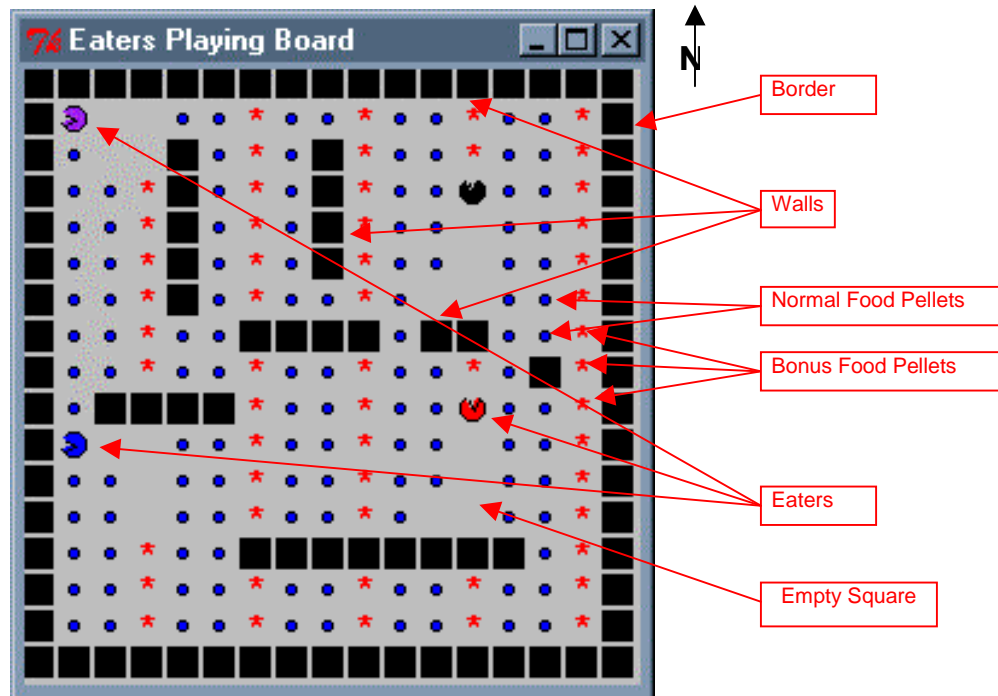


## Part I: Simple Soar Programs

Throughout this tutorial, you will be writing and running Soar programs for simple games. In this part, the programs are simple and do not use Soar subgoaling or learning mechanisms.

### 1. The Eaters Game

The game used in this part is called “Eaters.” In Eaters, PACMAN-like eaters compete to consume food in a simple grid world. Below is the Eaters Playing Board near the beginning of a game with four eaters.



The Eaters world consists of a rectangular grid, 15 squares wide by 15 squares high. Walls bound all four sides. Interior wall sections are randomly generated for each new game. No two walls will touch, so there are no corners, except for exterior walls and no “dead ends” anywhere on the board. Each eater starts at a random location. Food pellets are in all other squares of the grid. There are two kinds of food: normal food (blue circles and worth 5 points) and bonus food (red stars and worth 10 points). An eater consumes food by moving into a square. When an eater moves out of a square it will be empty (unless another eater moves into it).

An eater can sense the contents of cells (eater, normal food, bonus food, and empty) up to 2 squares away in all directions. On each turn, an eater can move one square north, south, east, or west. An eater can also jump two squares north, south, east, or west. An eater can jump over a wall or another eater. An eater does not consume any food in the space it jumps over. A jump costs the eater 5 points.

Whenever two eaters try to occupy the same cell at the same time, they collide. As a result of the collision, their scores are averaged and they are teleported to new, random locations on the board.

## 1.1 Installing Eaters and Soar

Before reading the tutorial, you should install Soar and its associated software on your computer so that you can run the examples and exercises. Using your favorite browser, visit the Soar Tutorial page: <http://ai.eecs.umich.edu/~soar/tutorial.html>. Follow the link to the Eater tutorial and follow the directions on that page for installation.

## 1.2 Creating an Eater

Start the Eaters game by double clicking on the Eaters icon in the folder where you installed Eaters. After Eaters has started, you will have two new windows on your computer screen. One of the windows is the Eaters Playing Board that was shown on page 5. The second window is the Eaters Control Panel, which is shown below. You will use this window to create and destroy eaters and run the game.

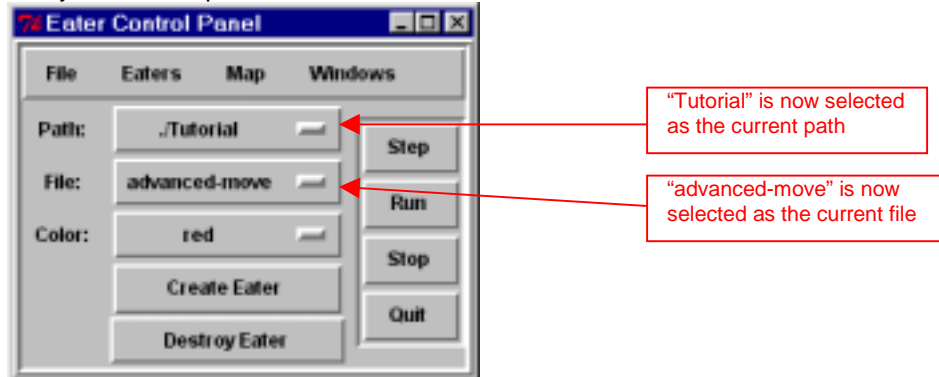


Eaters are controlled by Soar programs, which are automatically loaded when the eater is created. The first step in creating an eater is to select the path to a folder (directory) on your computers that contains files of Soar programs for controlling eaters. To select the path, click and hold your mouse on the button to the right of the word "Path:", which is highlighted in the figure below (If you have a two-button mouse, always click with the left-mouse button unless explicitly told to click with the right).

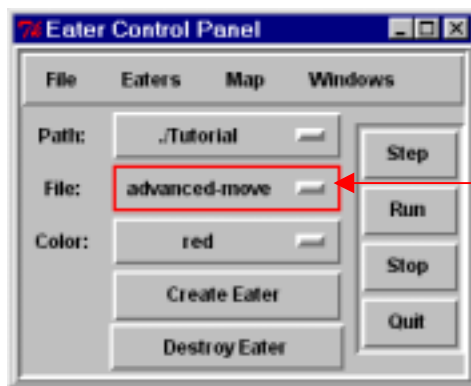


Click on this to select the folder.

When you click on the button, a menu of possible paths will come up. If this is the first time that the tutorial has been run on your machine, the only name that should come up is "Tutorial". Select that item and your control panel should look as follows:



The "Tutorial" folder is now selected as the current path. Below the path, you will see that advanced-move is selected as the current file. Advanced-move is selected only because it is the file in the folder that is first in alphabetic order. Your next step is to select a file. Different files have different programs, which generate different behavior. To select a file, click and hold your mouse on the File button (highlighted below). A menu of all of the possible files will appear. Select the file named "move-to-food".



After you have selected the move-to-food eater, the control panel should look as follows:



You can also pick a color for your eater by clicking on the button to the right of "Color:" labeled "red". The color does not affect on the eater's behavior, so pick your favorite color.

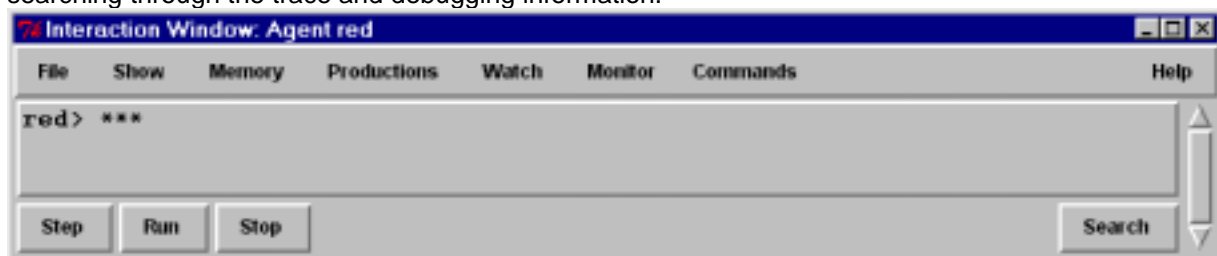


Once you have selected the path, the file, and a color, you are ready to create your eater by clicking on the Create Eater button.



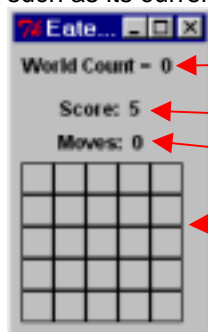
After clicking on Create Eater, two new windows will appear. Arrange the Eater windows so that they do not overlap.

One of the new windows is the Interaction Window. The Interaction Window will look as shown below, although it will be larger, and have additional panes. Across the top are a series of pull down menus, which are described later. The main body of the window is where you can type in commands and where trace and debugging information will be displayed. The bottom of the window contains buttons for single stepping, running, or stopping the eater. There is also a search button that creates a pop-up window for searching through the trace and debugging information.



The eater is referred to by its color, which is listed in the window title (Agent red), and in the prompt in the interaction window (red>). The \*'s in the window following the red> prompt are printed for the individual rules that are successfully loaded into the eater. In Soar, a program consists of rules, and move-to-food has only three simple rules that move the eater to a neighboring cell that contains food. As your eaters get more sophisticated, more rules will be loaded.

The Eaters Info window is the other window created. This window contains information about your eater, such as its current score, the number of moves it has taken, and a graphical depiction of its sensory data.



The number of steps taken by eaters since the map was initialized

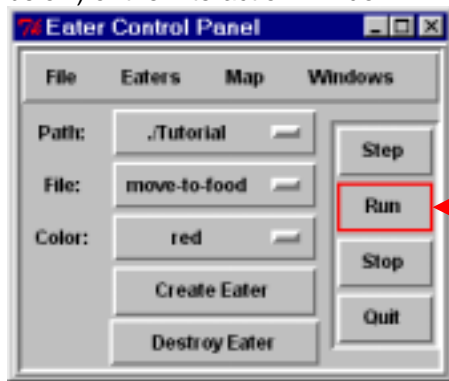
The eater's current score

The number of moves made by the eater

The current sensor data for the eater (empty because the eater hasn't started).

### 1.3 Running an Eater

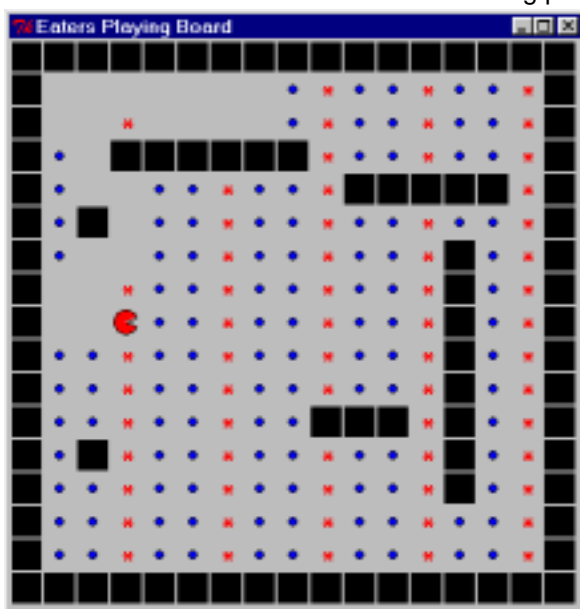
Your eater is now ready to start eating. Click the Run button on either the Eater Control Panel (highlighted below) or the Interaction Window.



The eater will start to move and consume food. After a few moves, click the Stop button on either the Eater Control Panel (highlighted below) or the Interaction Window.



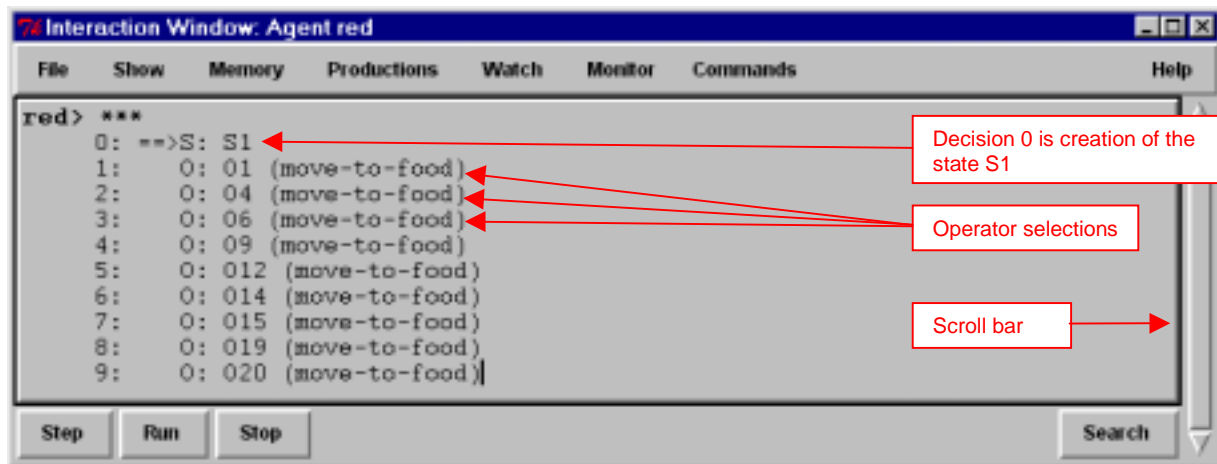
At this point, the Eaters Playing Board will look something like the one below. Your board will look different because the walls and eater starting position are different each time the game is played.



During the run, the Eaters Info window will change with each move, showing the food pellets, walls, and empty squares that the eater senses. As shown below, an eater can sense a 5x5 grid of cells, although the move-to-food eater only looks at the four neighboring cells.



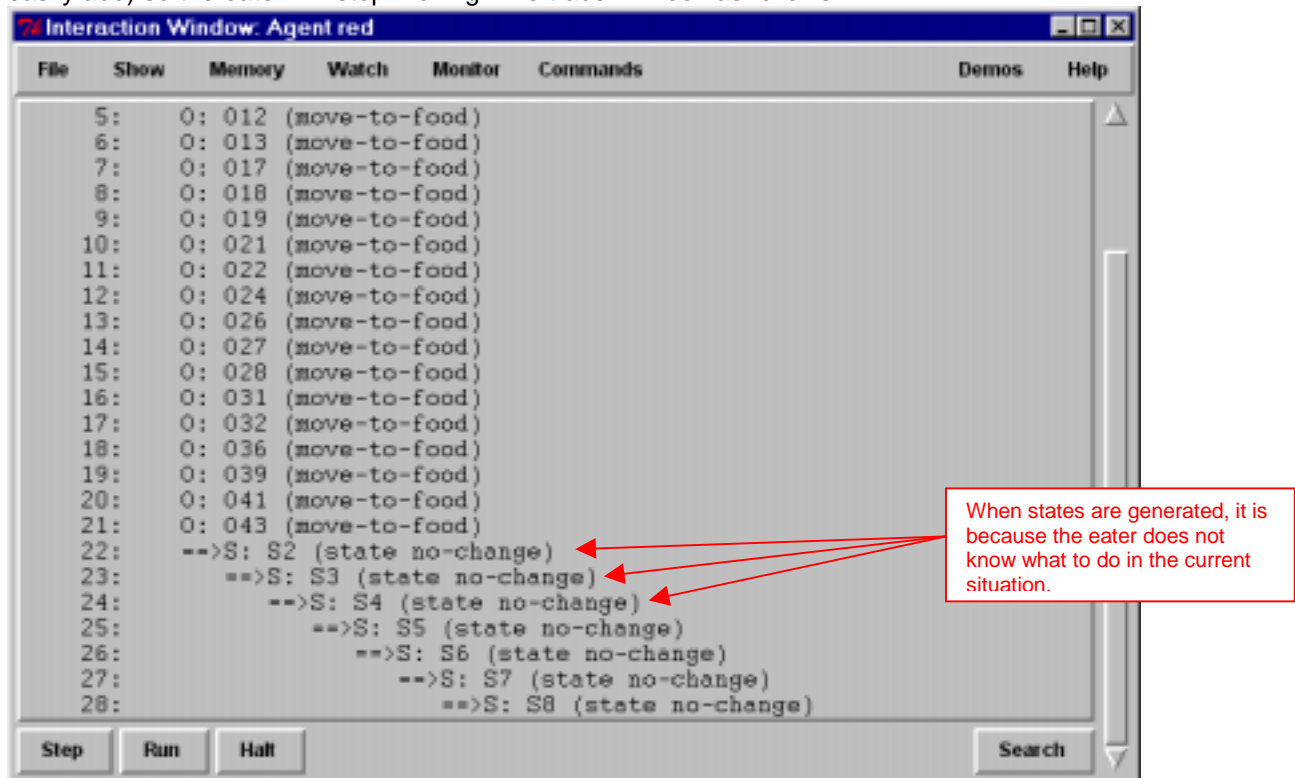
The Interaction window (shown below) displays a trace of the eater's decisions.



The line starting with 0: ==> shows the creation of the first *state*. This is followed by successive selections of move-to-food *operators*. The move-to-food operator causes the eater to move to a new cell. Some of the eaters will also jump; others will halt when they cannot sense any food.

If the trace gets longer than the window can hold, the printing will scroll up. You can look at decisions that scroll off the window by using the scroll bar at the far right of the window. You can search for information in the trace by using the search function at the lower right-hand corner of the Interaction Window.

If you let the eater run for a long time, it will eventually get to a place where there is no food directly next to it. The move-to-food eater does not have any rules to respond to this situation (although you could easily add) so the eater will stop moving. The trace will look as follows:



When an eater does not know how to respond to a situation, it will start generating new states, such as S2, S3, S4, ... in the above trace above. In later sections of the tutorial you will learn how to write rules that take advantage of these new states, but for now you just need to know that when they arise, the eater does not know what to do.

At anytime you can create more eaters. You can create additional eaters from the same or different files, so that you can create a set of identical eaters (except for color), or sets of different eaters. Each time a new eater is created, a new Interaction Window is created, and the Eaters Info window is expanded to include information on the new eaters.

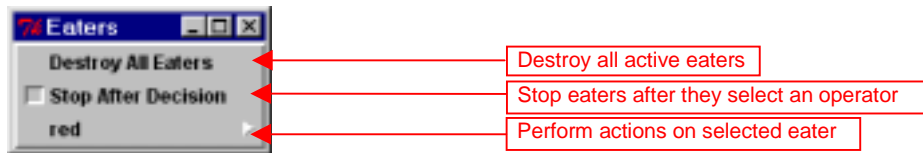
To destroy an eater, click and hold the Destroy Eater button. A menu will appear with the colors of all of the currently active eaters. Select the one you wish to destroy and it will be removed from the game.



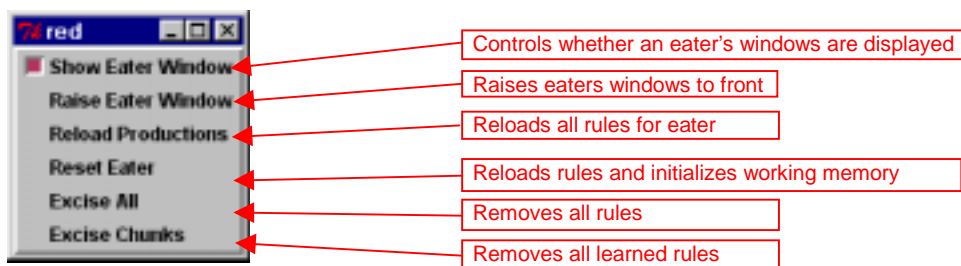
Additional menus are available via the labels at the top of the control window (Eaters, Map, Windows).



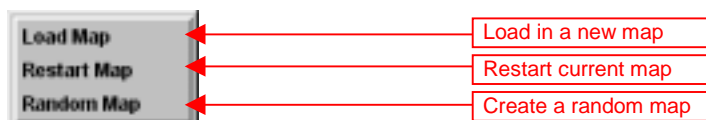
If you click on “Eaters,” a menu will appear that allows you to destroy all active eaters, modify when Soar stops (this will be explained in detail later), or select some actions for a specific eater based on its color. In the example, there is only one eater (red), so only its color shows up as a selection in the menu.



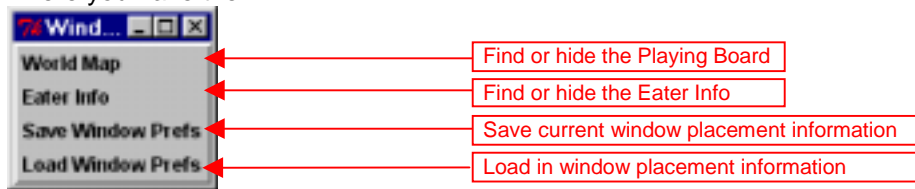
If you click on red, a new menu appears. The most important action allows you to reload the production rules for the eater. This is useful if you have found a bug, modified the rules in the file, and then wish to immediately test them on the current situation. Soar will load in the rules, replacing original versions with new versions (a “#” is printed in the Interaction Window whenever a rule is replaced), and adding new rules. Reset Eater reloads the production rules and restarts the eater as if it were just created. Excise All removes all rules, while Excise Chunks removes all learned rules (more on learning much later...).



If you click on the “Map” button, a menu with three options will appear. You can load in a stored map by clicking on Load Map. You can restart the eaters on the current map by clicking on the “Restart Map” button. The eaters will be placed randomly on the board and all of the food will be regenerated. You can also generate a completely new map by clicking on the “Random Map” button.



The Windows button lets you find or hide the Playing Board and Eater Info windows. It also lets you save away the current configuration of windows, so that the next time Eaters starts up, your windows will be where you have them.



To quit the Eaters game, click the Quit button on the Eaters Control Panel.



Spend some time creating, running and destroying different eaters. Notice how the eaters move around the board differently, some being much more efficient at consuming food.

## 2. Building A Simple Eater Using Rules

All of the knowledge in an eater is represented as if-then *rules*. In Soar, rules are called *productions* and we will use the terms interchangeably. Rules are used to select and apply things called *operators* and much of this tutorial will be describing how rules and operators fit together. But before we get to operators, we are going to learn about writing rules.

### 2.1 Hello-World Rule: English

The first eater we will look at just prints "Hello World" in the Interaction Window. In general, an operator should perform this type of activity, but in order to ease you slowly into Soar, we start with a single rule. Below is a simple rule written in English.

```
hello-world:
  If I exist, then write "Hello World" and halt.
```

Soar cannot directly interpret English, so you must write rules in Soar's very stylized and precise language. A major component of the rest of this tutorial is teaching you this language.

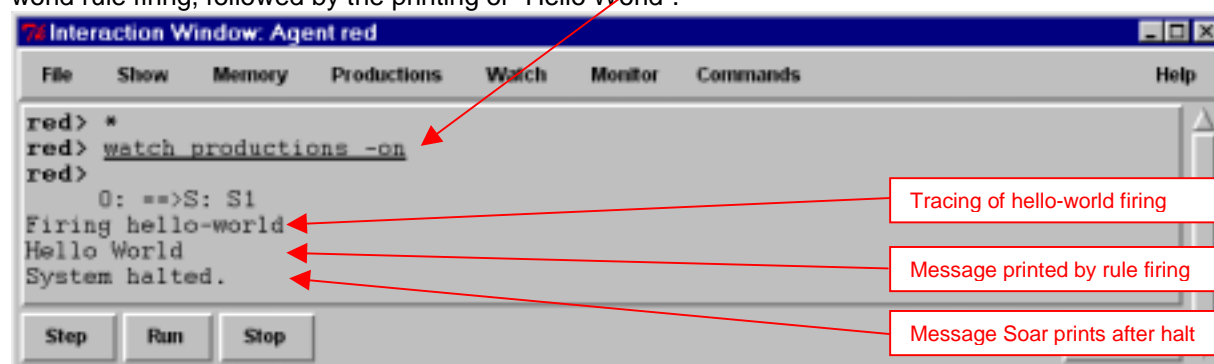
Soar works by testing the "if parts" of rules ("If I exist"). These "if parts" are called *conditions*. If the conditions of a rule are true in the current situation, the "then parts", or *actions*, of the rule ("write 'Hello World' and halt") are performed. This is called *firing* the rule. To determine if the conditions are true, Soar compares them to statements held in *working memory*. Working memory defines the current situation, which for an eater consists of its perception of its world, results of intermediate calculations, active goals, and operators.

To see what the hello-world rule does, start up Eaters, select the hello-world-rule file, and create an eater. Don't immediately click on run, because you will not see a trace of the rule firing. You must modify the level of tracing to see rule firings, by typing the following into the Interaction Window (followed by the "Enter" key):

```
red> watch productions -on
```

Only type in the underlined part

Now click on the step button. What happened? The Interaction Window contains a trace of the hello-world rule firing, followed by the printing of "Hello World".

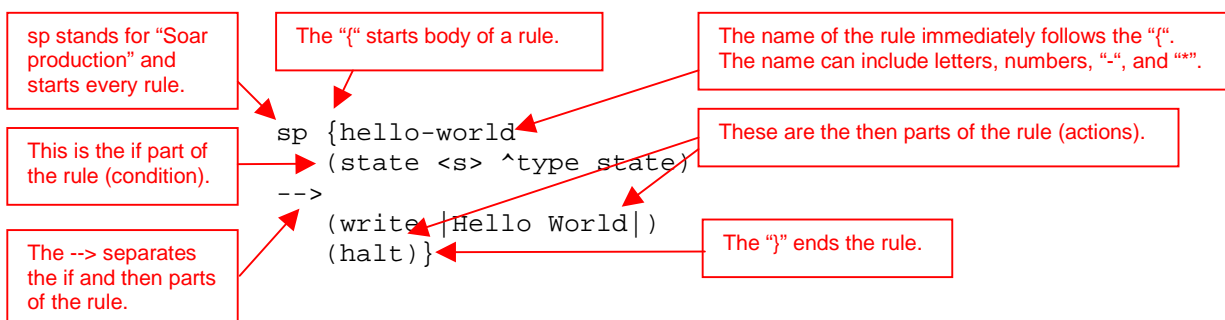


Based on hello-world, you should be able to come up with English descriptions of rules that write out messages.

Note: If you click on the run button, Eaters will try to keep running even after Soar halts. Eaters keeps running because there might be other eaters and it shouldn't stop just because one of them halts. Click on Stop to halt Eaters.

## 2.2 Hello-World Rule: Soar

As stated earlier, Soar has a very stylized and precise language you have to use. Here is the Soar rule for hello-world. This is about the simplest Soar program you can create.



Every rule starts with the symbol "sp", which stands for "Soar production." The remainder of the rule body is enclosed in curly braces: "{" and "}". The body consists of the rule name, followed by one or more conditions (the if part), then the symbol "-->", and then one or more actions (the then part). Below is a template for rules:

```
sp {rule*name
  (condition)
  (condition)
  ...
  -->
  (action)
  (action)
  ...}
```

... means additional conditions can follow.

... means additional actions can follow.

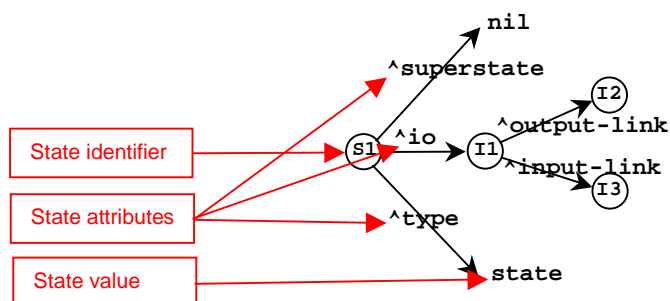
The name of the rule can be any arbitrary combination of letters, numbers, dashes ("-"), and asterisks ("\*"). The one exception is that a name cannot be a single letter followed by a number. Soar reserves those names for its own purposes. The specific name is irrelevant to the operation of the rule, but you should always pick a name that is meaningful. Following the name, there must be at least one condition. A condition tests for the existence (or absence) of some data in working memory. If all of the conditions match, then the rule will fire and all of the actions are performed. Most actions create new structures in working memory. Some actions remove structures from working memory, and some actions perform other functions, such as writing text to the screen or halting as in hello-world.

In the next several sections we will explore the structure of conditions and actions and explain what all of the special symbols, such as <s> and ^type, mean. Before doing that, we need to first look at the structure of working memory in more detail.



## 2.3 Working Memory

Working memory contains all of a Soar entity's dynamic information about its world and its internal reasoning. It contains data from sensors, intermediate calculations, current operators and goals. Soar organizes this information as *states* with all data being associated with a state. For the eaters you will build, there will be a single state, and all information will be connected, directly or indirectly, to the state. Below is a graphic picture of the part of working memory every Soar entity starts with.



Working memory is a graph structure with nodes, such as S1, I1, I2, I3, nil, and state, connected by links, such as superstate, io, type, output-link, and input-link. Soar has two kinds of nodes: *identifiers* and *constants*. The nodes that can have links emanating from them, such as S1, are called *identifiers*, while the others, such as state and nil are constants. In the example above, S1 is the identifier for the state. Identifiers are always created automatically by Soar and consist of a single letter followed by a number. The case of the letter does not matter, so S1 or s1 mean the same thing. Although I2 and I3 do not have any links emanating from them, they can in the future and are identifiers. In contrast, the symbol state is not an identifier and cannot have links emanating from it.

The links are called *attributes* and are always prefaced by an “^”. Only identifiers can have attributes. S1 has three attributes: superstate, io, and type. I1 has two: output-link and input-link.

The graph can be thought of as consisting of a set of triples, where each triple is an identifier, an attribute, and *value*, and where the value is that node pointed to by the attribute. In the example above, the symbols nil, I1, state, I2, and I3 are all values of various attributes.

Each of these identifier-attribute-value triples is a *working memory element*. There are five working memory elements in the figure above:

```
S1 ^superstate nil
S1 ^io I1
S1 ^type state
I1 ^output-link I2
I1 ^input-link I3
```

This is the minimal contents of working memory, and as your programs get larger and more complex, they will contain many more working memory elements.

A collection of working memory elements that share the same first identifier is called an *object*. For example, the three working memory elements that have S1 as their identifier are all part of the state object. The working memory elements that make up an object are called *augmentations*. We usually write all of the augmentations of an object together in a list surrounded by parentheses. The first item in the list is the identifier that is shared by all of the augmentation, followed by the pairs of attributes and values for each augmentation. The objects for the above working memory elements are:

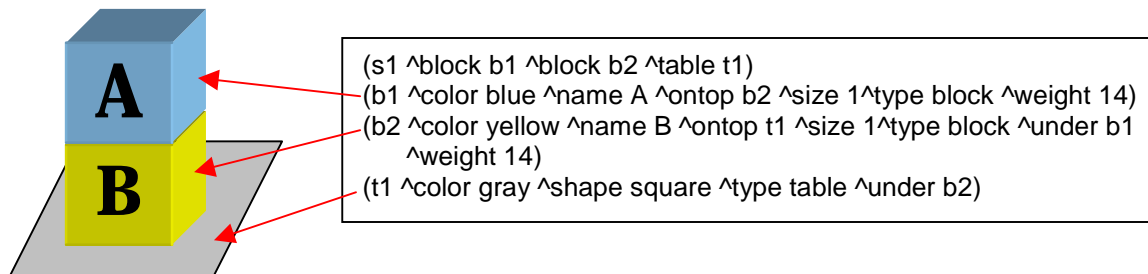
```
(S1 ^io I1 ^superstate nil ^type state)
(I1 ^input-link I3 ^output-link I2)
```

An individual augmentation can also be written in this form:

```
(S1 ^type state)
```

A working memory object usually represents something about the world, such as an eater, a piece of food, or a cell on the board. The individual augmentations represent properties (such as color, size, or weight), or relations with other objects (such as on top of, behind, or inside).

Soar can be used for many different tasks, and for each different task, working memory will contain descriptions of the task objects. If you were writing a program in Soar to control a robot that stacked blocks, you might represent the task objects, such as a blue block on top of a yellow block that is on a table, using the representation below.



Working memory usually also contains objects that are only conceptual things and do not have an identifiable physical existence, such as state `s1`, which organizes other objects, relations, and properties. The exact representation of objects is up to you, the designer of a Soar program.

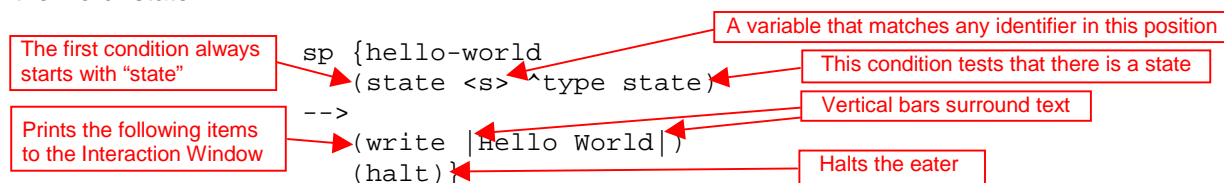
Soar does not require any declarations of the possible attributes and constants. In fact, some Soar programs generate new attributes and constant values as they execute. The Soar editing and development tool, Visual Soar, does require declarations for the structure of working memory elements, but those declarations are used only to check for errors in your rules and they are not used by Soar when it is executing a program.

## 2.4 Hello-World Rule: Soar Details

You now know enough to return to the first example rule. The original English version included the test “If I exist”

Remember that every eater has `(s1 ^type state)` in working memory, which signifies in some way that the eater does exist. Thus, you can test for the presence of that structure in working memory to determine if the eater exists. The obvious condition to write would be `(s1 ^type state)`. However, `s1` is just an arbitrary symbol and might not be the identifier of that state every time the eater is run.<sup>1</sup> Thus, we need a test that there is an identifier, but without testing a specific value. That is exactly what a *variable* does – it matches any symbol in working memory with the only constraint being that all occurrences of it in a rule match the same symbol. A variable is a symbol surrounded by “<” and “>”, such as `<s>`. The exact symbol used in the variable (such as “s”) is irrelevant but should be picked to be meaningful to its use.

Putting these pieces together gives: `(<s> ^type state)`. That is almost correct, but in Soar, every rule must start by matching a state, and to remind you of this, the first condition of every rule must start with the word “state”.



The original text for the rule's actions was:

```
then write "Hello World" and halt.
```

The rule's first action prints “Hello World” in the Interaction Window. Vertical bars, “|”, mark constants with special characters. These constants can contain any characters and allow you to print spaces, upper and lower-case letters. The second action is a special command in Soar that halts the eater. From this simple example, you should be able to write your own rules that print out different messages for Soar.

If you wish to try this out yourself, you have two choices. The first is to use Visual-Soar. Visual-Soar is the Soar development environment and it lets you develop and manage Soar programs. It has its own documentation and tutorial and can be found at the Soar web pages. Although it will initially slow you down to learn Visual-Soar, it will be worth it in the long run. If you choose to use Visual-Soar, first create a folder/directory for all of the eaters you will be writing in the “agents” folder of the Eaters program (“Eaters/agents”). The “Tutorial” folder is already there. For example, if I were doing the tutorial on a PC, I might call it “Johns-Eaters”. Then install Visual-Soar in that folder/directory and use it to create a new project, called move-north.

If you want to use a text editor, almost any will do, such as Word, WordPerfect, Wordpad, Notepad, or Emacs. In this editor, create a new file named move-north.soar in your new folder/directory. Make sure the file ends in “.soar”. This may be difficult to do in Wordpad or Notepad, which automatically add “.txt” to the end of a file, but is necessary for the file to show up in the Eater file menu. If you are having trouble with this in Windows, go to the Settings item off the Start menu. Select Folder Options and then pick the File Type tab. Select “New Type...” and in the pop-up window define the soar type. You need to fill in “soar” (without quotes) under the “Associated extension” and select “text/plain” under “Content Type.” Whenever you save the file, make sure you save it as a text file with linefeeds. Soar cannot handle formatted text and text without linefeeds.

<sup>1</sup> s1 will always be the identifier of the first state, but that is an artifact of the way Soar is implemented.

### 3. Building Simple Eaters Using Operators

In this section, you will learn how to use rules to select and apply *operators*. Operators perform actions, either in the world, or internally in the “mind” of an eater. In Eaters, there will be operators to move the eater around the board. There could also be operators to remember uneaten food, count food, or remember where walls are. The operators in a TicTacToe program would mark X’s or O’s on the board. The operators for a soccer program would probably include move, turn, kick the ball, send messages to teammates, interpret messages from teammates, choose a tactic or strategy, and so on.

Operators are created in working memory by one or more *proposal* rules, which test that the operator can apply. The actions of an operator are performed by one or more *application* rules, which test that the operator is selected and then make the appropriate changes to working memory.

#### 3.1 Hello World Operator: English Version

Instead of using a single rule to print “Hello World”, you will use an operator. Operators allow you to have an action considered in multiple situations (rules that propose the operator), allow multiple reasons for selecting an action (rules for selecting operators), and allow multiple ways for doing it (rules that apply the operator). For this simple, isolated action, an operator may not be necessary, but as soon as other actions are added, operators become extremely useful. Here are the rules for the hello-world operator.

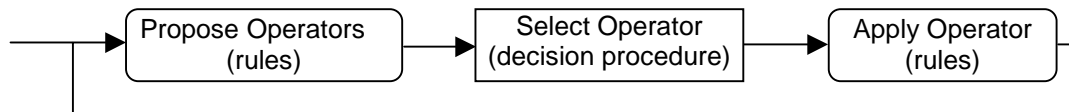
Propose\*hello-world:

If I exist, propose the hello-world operator.

Apply\*hello-world:

If the hello-world operator is selected, write “Hello World” and halt.

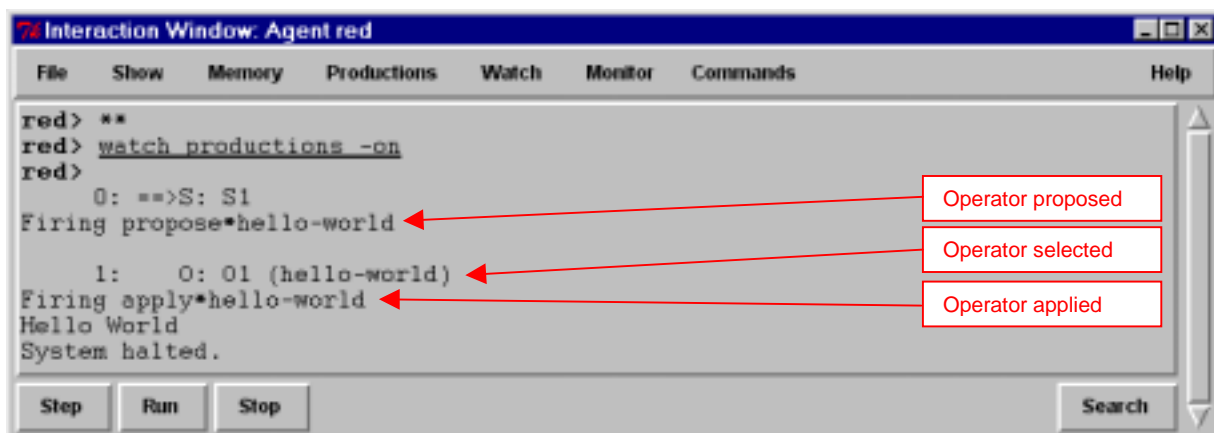
The first rule proposes the hello-world operator, and the second performs its actions after the operator has been selected. Notice that the first rule only proposes the hello-world operator. An operator is selected by Soar’s *decision procedure*, which collects together proposed operators and selects one.



Soar’s basic operation is a cycle in which operators are continually proposed, selected, and applied. Rules fire to propose and apply operators, while the decision procedure selects the current operator.

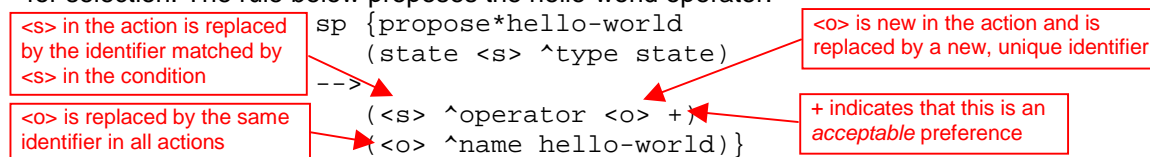
To see this operator run, destroy any existing eaters (such as hello-world-rule), and then select hello-world-operator as the current eater’s file and create an eater. Once again, change the tracing by typing in watch productions -on. Click on the run button.

The trace shows that propose\*hello-world fires first to propose the operator. The operator hello-world (O1) is then selected (by the decision procedure). After hello-world is selected, apply\*hello-world fires and performs the actions of the operator by printing out “Hello World” and halting.



### 3.2 Hello-World Operator: Soar Version

There is a one-to-one mapping of the English rules to Soar rules. The condition of `propose*hello-world` is the same as the condition for the hello-world rule. The difference is in the action, where `propose*hello-world` proposes the hello-world operator. A rule proposes an operator by creating an *acceptable preference* for the operator. An acceptable preference is a statement that an operator is as a candidate for selection. The rule below proposes the hello-world operator:

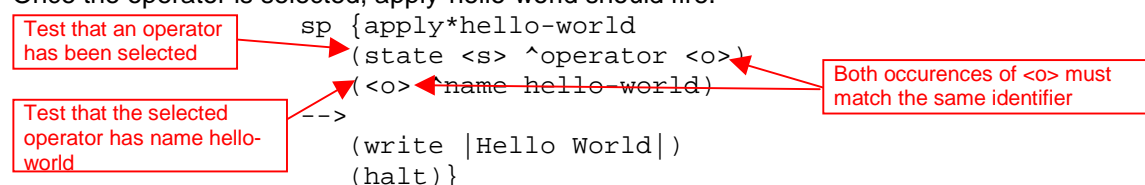


The first action creates an acceptable preference for a new operator (which is added to working memory) and the second action creates a working memory element that augments the operator with its name.

A preference looks just like other working memory elements except that it has a fourth item: the type of preference, which in this case is `+`. The identifier of the preference is `<s>`, which means that the identifier matched to `<s>` in the condition is used in creating the action. For example, if working memory contained `(s1 ^type state)`, then when this rule fired the preference would begin with `(s1 ^operator`. The value of the preference, `<o>`, is a new variable that did not occur in the condition. When new variables appear in actions, Soar automatically creates a new identifier and uses it for all occurrences of that variable in the action. For example, if `o1` is the identifier created for `<o>`, then `(s1 ^operator o1 +)` and `(o1 ^name hello-world)` are added to working memory. For action variables such as `<o>`, a different identifier is created each time a rule fires.

Additional rules may create preferences to compare operators. The decision procedure selects an operator based on all the created preferences. If a single operator is proposed, that operator is selected. In this case, `propose*hello-world` fires creating a single acceptable preference for `o1`, and then the decision procedure selects `o1` to be the current operator and adds `(s1 ^operator o1)` to working memory. Note that this augmentation *does not* have the `+` following the value. Only the decision procedure can add such an operator augmentation for a state to working memory (while proposal rules can add acceptable preferences for operators that differ by ending with the `+`).

Once the operator is selected, `apply*hello-world` should fire.



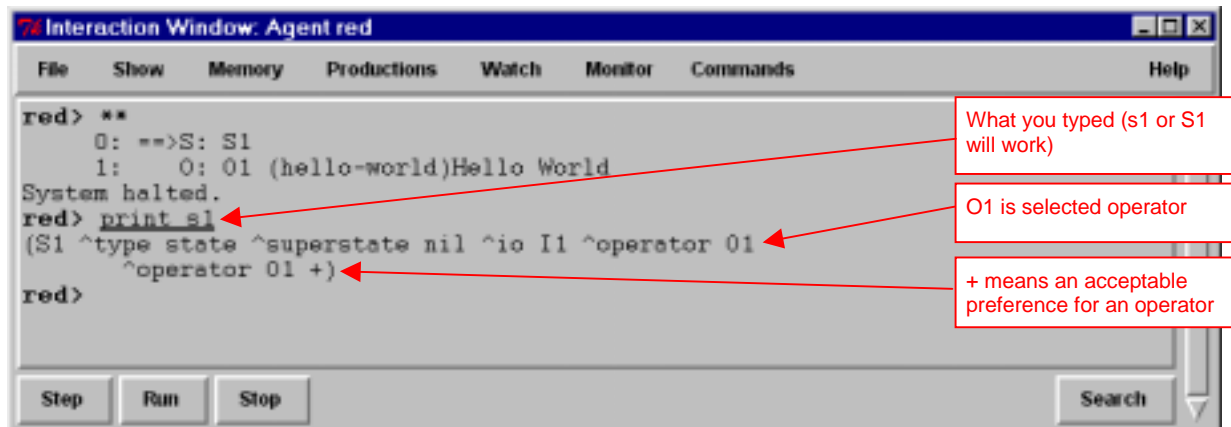
This rule has exactly the same actions as `hello*world`, but with conditions that test that the operator hello-world has been selected. The first condition tests that some operator has been selected. We cannot include a specific identifier for the value (such as `o1`) because the exact value of the identifier can be different each time we run the program, so a variable, in this case `<o>`, is used.

The second condition tests that some object in working memory has name hello-world. The rule will only match if both `<o>`'s match the same identifier (such as `o1`). This is true for all variables in Soar; if the same variable shows up multiple times in the conditions of a rule, the rule will not match (and fire) unless all occurrences of the variable in the rule match the same symbol in working memory. In this case they both match `o1`, and this rule fires.

If the same variable is used in other rules, it can match completely different identifiers or constants – that is, the identity of variables only matters within a rule. The exact symbols used in a variable are irrelevant, although as a general convention, `<s>` is used to match state identifiers and `<o>` is used to match operator identifiers. However, this is only a convention to make it easier to understand the rules.

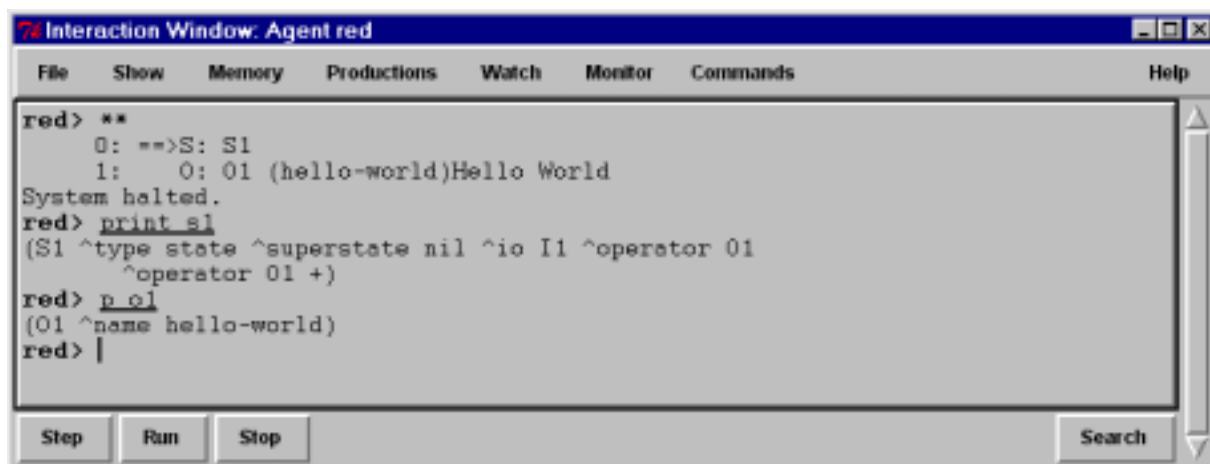
### 3.3 Examining Working Memory

We can get an even more detailed look at Soar's processing by examining the data structures in working memory. The Interaction Window allows you to print out the contents of working memory using the print command. To print out all of the attributes and values that have `s1` as the identifier, first click in the interaction window, and type `print s1` followed by the "Enter" key (the prompt will appear as soon as you start typing). The Interaction Window should now look like:

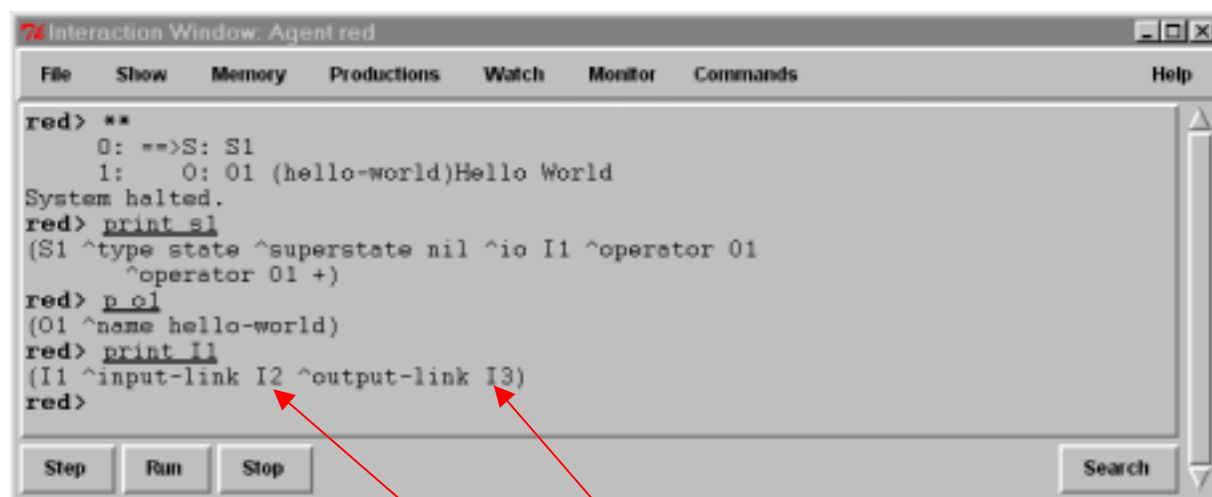


You now see all of the attributes and values that have `S1` as their identifier. The attributes `io`, `superstate`, and `type` are created automatically for the state when the program starts to run. The `operator` attribute is created when the `hello-world` operator (`o1`) is selected. Only the selected operator has a working memory element such as this *without* the following "+". In addition to the selected operator, the acceptable preference for the operator is there, marked with the "+". If there were additional operators proposed with acceptable preferences, there would be additional working memory elements with the operator attribute with the ids of the operators followed by the "+", but for a given state there is only one working memory element with the operator attribute without the "+", which signifies the *selected* operator. The attributes are ordered alphabetically when they are printed, but that is just to make it easier to find a specific attribute.

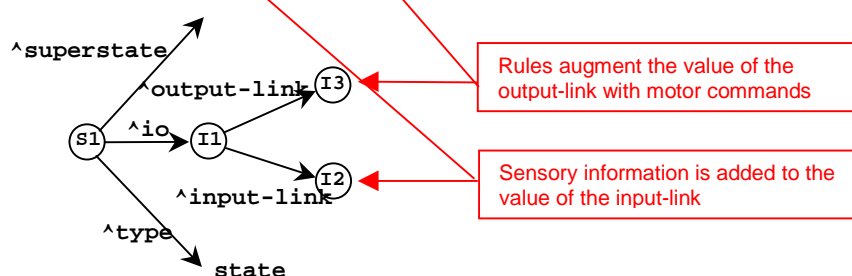
We can examine more of the structure of `o1` using the print command. This time, type `p o1`. The letter `p` means "print." Almost all commands in Soar also have one or two-character shortcuts.



Operator `o1` has an augmentation with attribute `name` and value `hello-world`. We can also examine the augmentations of some of the other values. `I1` is the value of the `^io` attribute, which stands for input-output. To see more about input and output, print `I1`. However, instead of using the `print` or `p` command, you can use the mouse to print information about `I1`. (If you don't have a mouse, just use `print`.) To do this, move the cursor over `I1` in the Interaction Window, then click and hold the right mouse button. A menu will appear that has a list of commands: `print`, `preferences`, `WMEs`, `productions`, and `run`. Move the mouse over the `print` command, still holding down the button, and slide it to the right. A new menu will appear with different print options. Slide the mouse over the top one and then release the button. Your interaction window should look like:



There are two attributes of `io`: `input-link` and `output-link`. The `input-link` is where an eater's sensory information is available in working memory. The `output-link` is where action commands must be created for the eater to move in its world. Using the mouse, explore the structure of the `input-link` (`I3`). You can also try out some of the other commands available through the mouse, although we will work our way through them later in the tutorial.



You now know the basic structure of rules and operators in Soar. You should be able to create your own eaters that print out simple messages with operators. Although you haven't gotten the eater to move, you have come a long way.

## 4. Move-North Operator

The first “real” operator you will create moves an eater north one step. (It might not even do that if there is a wall to its north when it is created.) The first step in creating an operator is to write down the proposal and application rules in English. Not only will this force you to think through many of the details of the operator before you worry about how to write it in Soar, but the English rules can be used as documentation for the Soar rules. As mentioned on page 19, you can use either Visual-Soar or a text editor to write your rules.

Type the English versions of the proposal and application rules for the operator move-north. We want these versions to be documentation, so we have to tell Soar to ignore them. You can do this by starting each line with the pound-sign (“#”). Soar ignores lines that start with a pound sign.

In Soar, the name of the rule can be almost any string of characters, but it is a good idea for it to be a meaningful summary of the purpose of the rule. A useful convention is to split the name into parts separated by asterisks (“\*”). The first part is the task, the second part is the function (propose, apply, elaborate), and the third is the name of the operator. There may be additional parts for more details of what the rule does. In the tutorial so far, there is really no task, so you should name your rules just with function\*operator-name.

Even if you don’t think you know what you are doing, it is worthwhile to take the time to write something before you turn the page and look at our example. A good idea is to copy as much as you can from what we’ve already done, so it would be a good time to look back at the hello-world operator. To save you some time, here is how the English version of hello-world might appear in a file.

```
##### Hello-world operator #####
# Propose*hello-world:
# If I exist, propose the hello-world operator.
#
# Apply*hello-world:
# If the hello-world operator is selected, write “Hello World” and halt.
```

You should now try to write an operator that moves the eater north one step.

This is easy and the operator you come up with will be similar to hello-world except that the action of the operator will be different. You can assume that there is an action command (move north) that moves an eater to the north.

(Don’t turn the page until you’ve written move-north.)



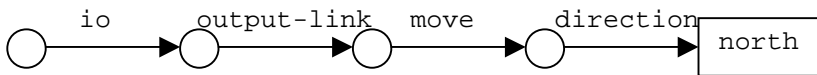
## 4.1 English Version

Here is one possible version of the move-north operator.

```
##### Move-north operator #####
# Propose*move-north:
# If I exist, then propose the move-north operator.
#
# Apply*move-north:
# If the move-north operator is selected, then generate an output command to
# move north.
```

This operator has only two changes from hello-world. First, the name is move-north instead of hello-world. Second the action of the second operator is to generate an output command to move north.

To write the Soar rule for apply\*move-north, you have to know how to get an eater to move. All external actions are issued by creating working memory elements that are augmentations of the output-link. The output-link is an augmentation of the io object, which in turn is an augmentation of the state. In Eaters, a move command is issued by creating a “move” augmentation on the output-link object, which in turn has an augmentation called “direction” with a value of the direction to move: “north”, “south”, “east”, or “west”. For each task in Soar, a set of output commands is defined, and in Eaters there are two commands: move and jump.



You should now try to write the Soar rules for the move-north operator. Here are the rules for Hello-World that you can use to create your own version of the move-north operator.

```
sp {propose*hello-world
    (state <s> ^type state)
-->
    (<s> ^operator <o> +)
    (<o> ^name hello-world)}

sp {apply*hello-world
    (state <s> ^operator <o>)
    (<o> ^name hello-world)
-->
    (write |Hello World|)
    (halt)}
```

Once you have modified these rules for move-north, make sure you save move-north.soar as a text file with linefeeds. Turn the page and compare your rules to the ones that I wrote.

## 4.2 Soar Version

Below is the proposal rule I wrote. The proposal rule is exactly the same as the proposal rule for hello-world, except that the names of the rule and the operator are changed.

```
sp {propose*move-north
    (state <s> ^type state)
-->
    (<s> ^operator <o> +)
    (<o> ^name move-north)}
```

Replaced hello-world with move-north

The rule that applies the operator has more differences. Its action is to add a move command to the output-link. This command will cause the eater to move one cell to the north. In order to add the command to the output-link, the rule must match the output-link identifier in its conditions with a variable. Therefore, conditions are included that match the io (<io>) object, and then the output-link (<ol>).

```
sp {apply*move-north
    (state <s> ^operator <o>
        ^io <io>)
    (<io> ^output-link <ol>)
    (<o> ^name move-north)
-->
    (<ol> ^move <move>)
    (<move> ^direction north)}
```

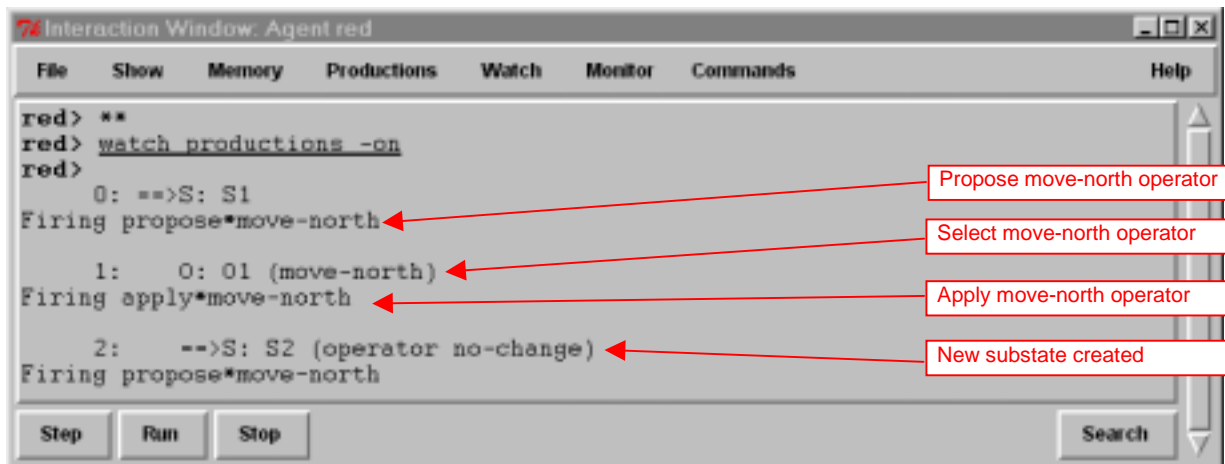
^io must be matched to get to the output-link

^output-link gives path to identifier for action

Add action to output-link to move north

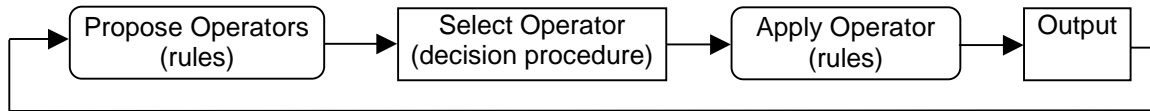
The exact order of the conditions (and actions) does not matter, except that the first condition must match the state identifier.

You should change your rules if they were significantly different than these. Then create a move-north eater and run it by clicking on the step button a few times until your eater moves north. If your eater doesn't seem to be working, skip ahead to the Debugging Your Eater section. Once your eater has moved north, your interaction window should look like the following:



But what happened to your eater? After selecting move-north and taking one step, the eater had nothing more to do and a new state is created (s2 in the above Figure). Unfortunately, the rules you wrote are not sufficient to get the eater to continually move north. In the next section, you will modify your eater so that it continues to move north.

One alternative you might have tried was to include a halt in the action of apply\*move-north. If you tried that, your eater did not move because the halt is executed before the output is processed.



In Soar, after all of the application rules have fired, the output system picks up new working memory elements on the output-link and sends them to the Eaters' world, causing the eater to move. If you had included a halt as an action in apply\*move-north, Soar would have stopped before the eater moved.

### 4.3 Shortcuts

The apply\*move-north rule has lots of variables in it whose sole purpose is to make connections between attributes. For example, <o> is used to later match the name of the operator and <io> is used to later match the output-link. Similarly, <move> is used in the action to connect the output-link to the final direction. None of these variables is used in both conditions and the actions.

```

sp {apply*move-north
  (state <s> ^operator <o>
    ^io <io>)
  (<io> ^output-link <ol>)
  (<o> ^name move-north)
-->
  (<ol> ^move <move>)
  (<move> ^direction north)}
  
```

To simplify the writing and reading of rules, Soar allows you to combine conditions that are linked by variables. To do this, you can just string together attributes, replacing the intermediate variables with a period, ".", to separate the attributes. In Soar, we call this "dot" or "path" notation. Below is the same rule as before using dot notation.

```

sp {apply*move-north
  (state <s> ^operator.name move-north
    ^io.output-link <ol>)
-->
  (<ol> ^move.direction north)}
  
```

. replaces <o>  
 . replaces <io>  
 . replaces <move>

This rule is exactly the same as the original from Soar's perspective.

One mistake you want to avoid making is to use dot notation in the action when you are creating multiple sub-attributes of a new object. This will create multiple objects, each with a single sub-attribute. For example, if you want create a second augmentation of the move object called speed (which is irrelevant in Eaters), you *do not* want to do the following:

```

-->
  (<ol> ^move.direction north
    ^move.speed fast)
  
```

This is equivalent to:

```

-->
  (<ol> ^move <move1>
    ^move <move2>)
  (<move1> ^speed fast)
  (<move2> ^direction north)
  
```

These actions will create *two* move augmentations on the output-link, each with a single attribute. The correct action is:

```

-->
  (<ol> ^move <move>)
  (<move> ^speed fast
    ^direction north)
  
```

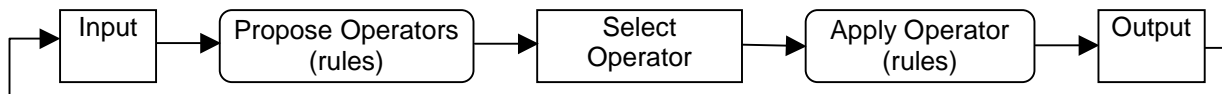
## 5. Move-North Operator: Multiple Moves

Although the move-north operator did move one step north, the eater never takes a second step. In this section, you are going to find out why it takes only one step and modify the eater so that it can take multiple steps. This is one of the most complex sections of the tutorial, so study it carefully.

### 5.1 Operator Selection: Multiple Instances

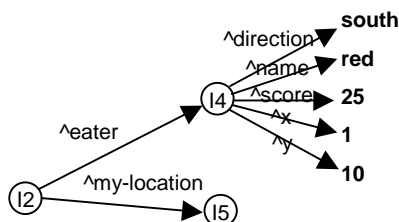
In Soar, each action in the world, such as moving an eater, should be performed by a separate operator *instance*. An operator instance is a separate operator in working memory element created by the firing of an operator proposal rule. Thus, each action should test for the creation of a new operator object in working memory. Different instances of the same operator will have the same name, and sometimes will have the same augmentations, but a given instance will be used for only one move. Some operator instances may include many actions, but they will be selected and applied only once.

Therefore, new instances of the move-north operator should be created in working memory for each new move. You should not attempt to have the move-north operator selected once, and have it move an eater multiple times. Instead, you should design your eater so that a new instance of the move-north operator is created for each move. You can do this by having the operator proposal rule fire each time the eater is to move. How can you change propose\*move-north so that it fires after each move? As the rule is currently written, it will fire only once because it only tests `^type state`, which stays in working memory forever. Your rule needs to test working memory elements that change each time the eater moves – those working memory elements that correspond to the eater's senses on the input-link. Soar is designed so that changes to the input-link are made following output, just in time to affect operator proposals.



The information that comes in on the input-link consists of objects with attributes and values.

```
(I2 ^eater I4 ^my-location I5)
(I4 ^direction south ^name red ^score 25 ^x 1 ^y 10)
```



The input-link object, I2, has two augmentations. The first, `^eater`, has information about the eater: its current direction, its name, its current score, and its x, y coordinates. The second, `^my-location`, has additional substructure (not shown) that includes the eater's sensing of nearby cells. Some of the `^eater` information changes during the game:

- The x location will change when the eater moves east or west.
- The y location will change when the eater moves north or south.

Thus, whenever the eater moves, either the x or y location will change. We can modify the conditions of our proposal rule to test both of these working memory elements, and eliminate the test for `^type state`.

```
sp {propose*move-north
  (state <s> ^io.input-link.eater <e>)
  (<e> ^x <x> ^y <y>)
-->
  (<s> ^operator <o> +)
  (<o> ^name move-north)}
```

Matches current position of eater,  
which changes after each move.

What will happen? First, when the original working memory elements for x and y are removed from working memory, the original move-north operator will be removed from working memory because the rule instantiation that created it no longer matches – some of the working memory elements responsible for the match are gone. We call this *retracting* the rule firing. Rules such as propose\*move-north maintain their actions in working memory only as long as they match exactly the same working memory elements.

Second, a new instance of the move-north operator will be created because propose\*move-north will match the new values of x and y and fire. To see these changes, modify your move-north operator as above and then rerun it. This time, we are going to increase the tracing so that we can see the individual changes to working memory. To do this, either type “watch wmes –on” in the Interaction Window, or click on the watch menu at the top of the interaction window and select “4. WMEs.” You should also trace the production firings.

The figure below shows the end of the trace. The trace is long because it includes the addition and removal of all of the sensory data that changes during a move (not shown below). In the trace, an addition to working memory is shown preceded by an “=>”, while a removal from working memory is preceded by an “<=”. Each working memory element also contains a number, such as 153 for the top element in the trace below. This is the *timetag* of the working memory element – a unique number generated when a working element is created. Soar sometimes displays only timetags instead of the full working memory element, and the timetag can be used in the print and wmes commands for displaying the full working memory element (more details later).

```

Interaction Window: Agent red
File Show Memory Productions Watch Monitor Commands Help

=>WM: (153: M1 ^status complete)
Firing propose*move-north
Retracting propose*move-north
=>WM: (155: S1 ^operator O2 +)
=>WM: (154: O2 ^name move-north)
<=WM: (121: S1 ^operator O1 +)
<=WM: (122: S1 ^operator O1)
<=WM: (120: O1 ^name move-north)
=>WM: (156: S1 ^operator O2)

2: 0: O2 (move-north)
Firing apply*move-north
=>WM: (158: M2 ^direction north)
=>WM: (157: I3 ^move M2)
=>WM: (164: S2 ^quiescence t)
=>WM: (163: S2 ^choices none)
=>WM: (162: S2 ^impasse no-change)
=>WM: (161: S2 ^attribute operator)
=>WM: (160: S2 ^superstate S1)
=>WM: (159: S2 ^type state)

3: --S: S2 (operator no-change)
red> p i3
(I3 ^move M1 ^move M2)
red>

```

You will notice that your eater still doesn't move more than once even though a second move-north operator is selected and a second move command is created. The reason is that the original move command (i3 ^move m1) is not removed and Eaters must have only a single move command if it is to move an eater. The move command is not removed because it is the action of an operator application rule. Such actions are not automatically removed when the rule no longer matches.

## 5.2 Operator Application: Persistence

As noted above, proposing and selecting a second operator does not make the eater move a second time because the first move command is not removed from working memory. Even if Eaters was more flexible, the accumulation of more and more move commands is a problem. Why aren't these commands automatically removed like the preferences for operators? The reason is that the rule that creates them is part of the application of an operator, and operator applications create *persistent* working memory elements.

Persistence is necessary for creating memories of prior events, such as the memory of something sensed in the environment. For example, if an eater needed to remember the location of uneaten food so that it could return to it later, the eater would have to create a persistent structure in its working memory. Otherwise, as soon as it stopped sensing the food, it would forget about it. In Soar, all working memory elements created by an operator application rule are persistent. A rule is an operator application rule if it tests a selected operator and modifies the state. Persistent working memory elements are said to be *operator-supported*, or *o-supported*, because they are created by operators. For example, `apply*move-north` is an operator application rule and creates o-supported working memory elements.

```
sp {apply*move-north
  (state <s> ^operator.name move-north
    ^io.output-link <ol>)
-->
  (<ol> ^move.direction north)}
```

Tests selected operator

Actions are o-supported

Just as it is important to have persistent memories, it is also important to have structures that are removed automatically when the reason for creating them goes away. Actions of rules that test only the state, or that test the operator and elaborate only the operator create non-persistent preferences or working memory elements. Obvious examples of non-persistent structures are the preferences for operators. These are created by operator proposal rules, which retract the operators when the proposal conditions change. Other examples include state elaboration rules and operator elaboration rules. Non-persistent working memory elements and preferences are called *instantiation-supported*, or *i-supported*, because they only persist as long as the rule instantiation that created them. For example, `propose*move-north` is not an operator application rule because it does not test a selected operator. All of its actions are i-supported and they will be removed from working memory when `propose*move-north` no longer matches.

```
sp {propose*move-north
  (state <s> ^io.input-link.eater <e>)
  (<e> ^x <x> ^y <y>)
-->
  (<s> ^operator <o> +)
  (<o> ^name move-north)}
```

i-supported actions

Returning to the problem of moving the eater, you need to add a rule that removes old move commands from the output-link after the move is finished. In Eaters, the output system creates an augmentation on the move object after the action has executed: `^status complete`. To remove a structure from working memory, a rule “rejects” a working memory element in its action. A reject must be part of an operator application rule because it is a persistent change to working memory. Therefore, the removal must test the move-north operator so that it is an operator application rule. In English, the rule is:

```
# Apply*move-north*remove-move:
# If the move-north operator is selected,
#   and there is a completed move command on the output link,
#   then remove that command.
```

In Soar, the action of a rule can remove a working memory element by following it by a reject preference, which is designated with a minus sign (“-”). The above rule is translated in to Soar as follows:

```
sp {apply*move-north*remove-move
  (state <s> ^operator.name move-north
    ^io.output-link <o1>)}
  (<o1> ^move <move>)
  (<move> ^status complete)
-->
  (<o1> ^move <move> -)}
```

Cannot use dot notation for <o1> and <move> because they are needed in the action

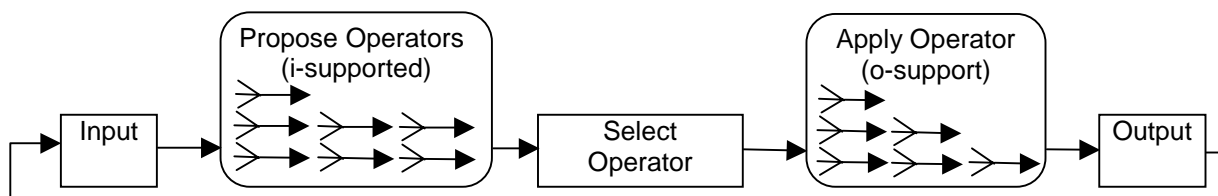
Created by the output system when the move action is completed

- means remove from working memory

The action of this rule removes the working memory element specified by `(<o1> ^move <move>)`, which would be something like `(i3 ^move m1)`. When this working memory element is removed, all of the augmentations of `m1` are automatically removed because they are no longer *linked* to the rest of working memory. In this case, `(m1 ^direction north)` and `(m1 ^status complete)` are removed. These would not be removed if they remained linked via another working memory element that had `m1` as a value.

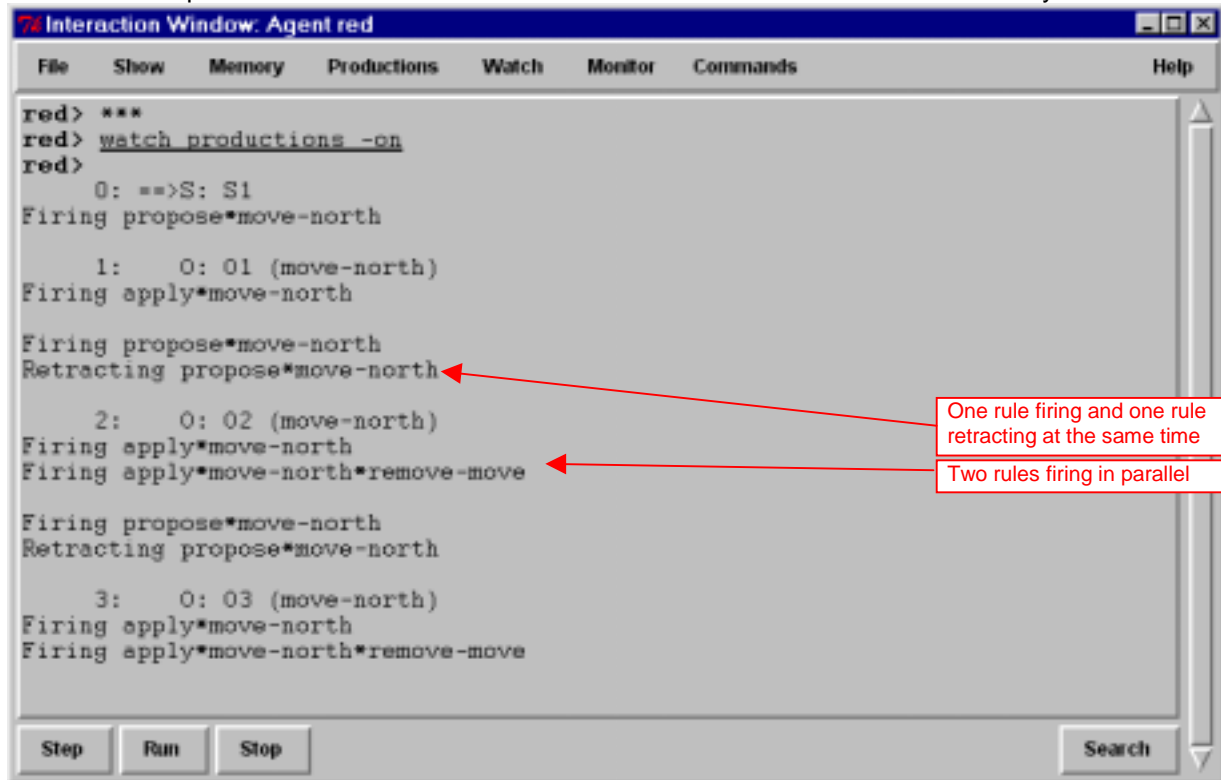
This is an operator application rule, so it will not fire during the following operator proposal phase. Instead, it will fire during the next operator application phase, after a new move-north operator has been selected. It will not interfere with other operator application rules because rules fire (and retract) in *parallel* as a wave. The parallelism is only simulated, but the effect is as if all the rules fire or retract at the same time. Once a wave of rules has fired, their actions may cause additional rules to match or retract. Soar will continue to fire or retract additional waves of rules until no more rules match. Thus, within the phases of proposing and applying operators, Soar fires all rules that match until *quiescence*, that is, until no more rules match. During the propose phase, only rules that have i-supported actions will fire. During the apply phase, both o-supported and i-supported rules fire.

Returning to our diagram of the phases of Soar, both the propose operator and apply operator phases can be expanded to show that multiple rules fire and retract in parallel until quiescence.



### 5.3 Running and Tracing

When running your new eater, it will go north until it hits a wall or the border. When it can no longer move, it will start generating states because it cannot successfully apply and terminate the move-north operator. Below is a sample trace of the interaction window for the first three moves to the north by an eater.



If you look closely at this trace, you see that two rules are firing at the same time: `apply*move-north`, and `apply*move-north*remove-move`. One is adding the new move command to the output-link at the same time that the other is removing the previous move command. There are also examples of rules firing and retracting at the same time where an operator is being removed from working memory at the same time that a new operator is being created.

You can see how the rules fire in different phases by turning on the watch for phases. The easiest way to do this is to click on the watch button on the menu bar and then select phases. You can also type in `watch phases -on`.



Instead of letting Soar run free, you can step Soar through each decision. The default will stop each step after the application phase, when output is being processed. This is just after the output commands have been added to the output-link and just before the eaters are about to move. In the following trace, the step button has been used, and then the output-link is examined. This is repeated twice.

```

blue> ***
      0: ==>S: S1
      1: 0: 01 (move-north)
blue> p i3
(I3 ^move M1)
blue> p -depth 2 i3
(I3 ^move M1)
      (M1 ^direction north)
blue> step
      2: 0: 02 (move-north)0
blue> p -depth 2 i3
(I3 ^move M2)
      (M2 ^direction north)
blue>

```

From prior runs, you know that `I3` is the identifier of the output-link. Although Soar does not guarantee that `I3` will always be the identifier of the output-link, it always is, just as `I1` is the identifier of the io object, `I2` is the identifier of the input-link, and `S1` is the identifier of the first state.

When you print `I3`, you see that the move command is correctly added to the output-link. You can see more than one level of augmentation by using the `-depth` flag on the print command. Here a depth of 2 was used, which is sufficient for printing out the augmentations of `I3` and `M1`.

You can also step through one phase at a time by using the `run -p` command.

## 6. Move-To-Food Operator

In this section you will create an eater that greedily moves to any food it senses. You will use the lessons you learned from the move-north operator, and learn more about the structure of the input-link and operator preferences.

Each eater can sense the food and walls that surround it, two steps in each direction. The Eaters Info window shows what an eater can sense. For this eater, you will write an operator that moves the eater to one of the neighboring cells (north, east, south, or west) that contains normalfood or bonusfood. Since food can be in more than one of the neighboring cells, more than one operator may be proposed. Soar does not automatically select randomly from among a set of proposed operators – it will get a *tie impasse* if there are multiple operators with only acceptable preferences. To avoid ties, Soar has additional preferences. For this exercise, it does not matter which food an eater consumes first, so you can use a preference that makes the decision procedure arbitrarily select one of the proposed operators.

Based on the above discussion, you will need four rules for the move-to-food operator.

- You need one rule to propose the operator when there is normal-food in a neighboring cell and a second rule to propose the operator when bonus-food is in a neighboring cell. Unlike the move-north operator, these proposal rules do not have to test the coordinates of the eater because the contents of the neighboring cells will change when the eater moves. The contents will change (the working memory elements for contents of all of the sensed cells are removed and re-added to working memory) even if the sensed object is the same type. The operator proposal rules can also create the indifferent preferences that lead to a random selection.
- You need a third rule to apply the operator and move the eater in the correct direction.
- You need a fourth rule to remove the move command from the output-link.

Now try to write English versions of the move-to-food operator. Use the English versions of the move-north operator as a guide:

```
##### Move-north operator #####
# Propose*move-north:
# If I am at a specific location, then propose the move-north operator.
#
# Apply*move-north:
# If the move-north operator is selected,
#   generate an output command to move north.
#
# Apply*move-north*remove-move:
# If the move-north operator is selected,
#   and there is a completed move command on the output link,
#   then remove that command.
```

## 6.1 English Version

Here is one possible set of rules for the move-to-food operator.

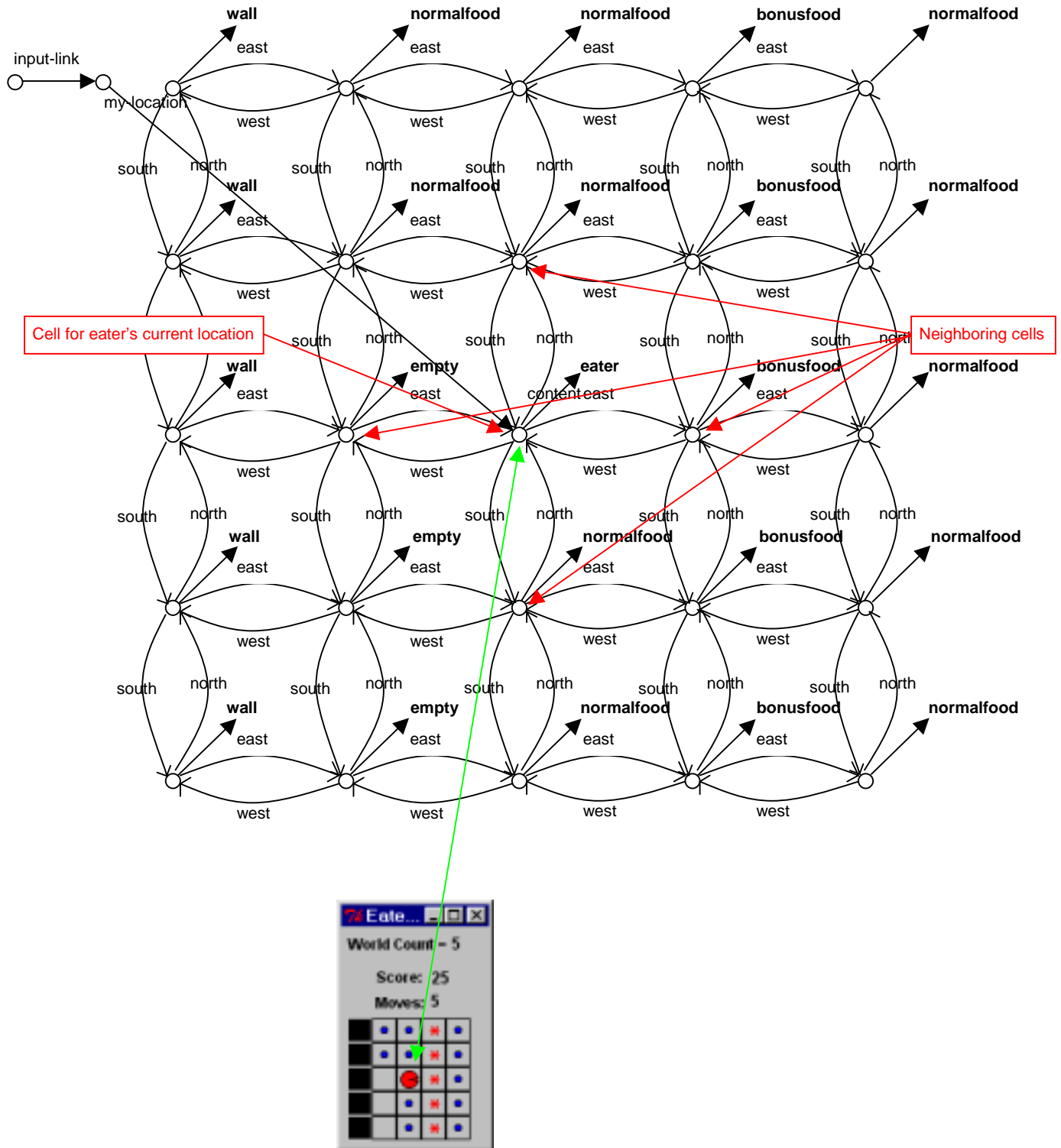
```
##### Move-to-food operator #####
# Propose*move-to-food*normalfood
# If there is normalfood in an adjacent cell,
#   propose move-to-food in the direction of that cell
#   and indicate that this operator can be selected randomly.
#
# Propose*move-to-food*bonusfood
# If there is bonusfood in an adjacent cell,
#   propose move-to-food in the direction of that cell
#   and indicate that this operator can be selected randomly.
#
# Apply*move-to-food
# If the move-to-food operator for a direction is selected,
#   generate an output command to move in that direction.
#
# Apply*move-to-food*remove-move:
# If the move-to-food operator is selected,
#   and there is a completed move command on the output link,
#   then remove that command.
```

Converting these rules to Soar requires a more detailed examination of preferences and the input-link.

- **Preferences.** To select randomly among the proposed operators, Soar has a preference called *indifferent*. The equal sign, “=”, is used to signify an indifferent preference for an operator, just as “+” signifies acceptable. The indifferent preference means that the decision procedure can randomly select among all operators with indifferent preferences, so it is important that all proposed operators have indifferent preferences. Even though we will create an indifferent preference for these operators, the acceptable preference is still necessary and an operator will not be selected if it does not have an acceptable preference.
- **Input-link representation.** The input-link has two augmentations: eater and my-location. Remember that the value of the my-location augmentation is the eater’s current position and is a cell in middle of the 5x5 Eaters Info sensory field. On the next page there is a graphic depiction of this cell and all of the other cells in the sensory field. Adjacent cells are augmentations labeled north, east, south, and west. Every cell also has a ^content augmentation, whose value can be wall, empty, eater, normalfood, or bonusfood. If the cell has an eater in it, there will be additional augmentations (not shown in the figure) for the color of the eater (^eater-color), and the eater’s current score (^eater-score). Below the figure for the input-link structure is the corresponding Eaters Info window.

Based on the information in those figures, you should try to write a Soar rule for propose\*move-to-normalfood. Two hints:

1. Write down the sequence of attributes that go from the state to the value of the content of a cell, separating each attribute with a “.”.
2. Use a variable to match the direction augmentations to a neighboring cell and then test that the content is normalfood. That variable in the attribute position will match any direction augmentation between cell: north, east, south, or west. You can write it: ^<variable>



## 6.2 Soar Version

Below is the proposal rule I wrote without dot notation so that it is clear which working memory elements are being matched. The proposal rule tests a sequence of linked identifier-attribute-values, and includes a variable for the attribute that leads to an adjacent cell. The variable will match north, east, south, or west if the final condition matches an adjacent cell containing (has “^content”) normalfood.

```

sp {propose*move-to-normalfood
  (state <s> ^io <io>)
  (<io> ^input-link <input-link>)
  (<input-link> ^my-location <my-loc>)
  (<my-loc> ^<direction> <cell>)
  (<cell> ^content normalfood)
-->
  (<s> ^operator <o> +)
  (<s> ^operator <o> =)
  (<o> ^name move-to-food
    ^direction <direction>)}

```

Matches any attribute

Matches an adjacent cell

Matches cell with normalfood

= signifies indifferent/random

Use matched direction in action

The indifferent preference tells Soar that a random selection can be made between proposed operators. The <direction> in the action augments the operator with the direction of an adjacent cell containing normalfood. When an eater is surrounded by food, the <direction> variable will match all directions, leading to four matches of the rule. In Soar, all new matches fire in parallel, creating new operators, each with a different ^direction augmentation. For example, if there was normalfood to the south and west, two operators would be created, one with ^direction south and one with ^direction west.

This rule can be written much more concisely using dot notation, and a short cut for the preferences.

```

sp {propose*move-to-normalfood
  (state <s> ^io.input-link.my-location.<dir>.content normalfood)
-->
  (<s> ^operator <o> + =)
  (<o> ^name move-to-food
    ^direction <dir>)}

```

+ = means two preferences are created: acceptable and indifferent

The rule that proposes the operator to move to consume bonusfood is very similar.

```

sp {propose*move-to-bonusfood
  (state <s> ^io.input-link.my-location.<dir>.content bonusfood)
-->
  (<s> ^operator <o> + =)
  (<o> ^name move-to-food
    ^direction <dir>)}

```

The rule to create the move command on the output link is very similar to the one used in move-north.

The only difference is that instead of always using “north” as the direction, it uses the direction created by the operator proposal rule and matched by variable <dir>.

```

sp {apply*move-to-food
  (state <s> ^io.output-link <ol>
    ^operator <o>)
  (<o> ^name move-to-food
    ^direction <dir>)
-->
  (<ol> ^move.direction <dir>)}

```

The final rule removes the move command from the output-link when it has completed.

```

sp {apply*move-to-food*remove-move
  (state <s> ^io.output-link <ol>
    ^operator.name move-to-food)
  (<ol> ^move <move>)
  (<move> ^status complete)
-->
  (<ol> ^move <move> -)}

```

### 6.3 Shortcuts and Extensions

The rules `propose*move-to-normalfood` and `propose*move-to-bonusfood` differ only in the tests for `normalfood` and `bonusfood`. Instead of writing an individual rule for each of those values, it is possible to write a single rule that tests for any one of a set of alternative values. The alternative values are written in the same position as a single value, but are surrounded by double angle brackets: `<< normalfood bonusfood >>`. Any number of different values can be included, but none of them can be a variable. Using this notation, the two rules can be rewritten as the following rule:

```
sp {propose*move-to-food
  (state <s> ^io.input-link.my-location.<dir>.content
    << normalfood bonusfood >>))
-->
  (<s> ^operator <o> + =)
  (<o> ^name move
    ^direction <dir>)}
```

Make sure there is a space between the <<, >>, and the values

This notation can also be used for testing alternative values of attributes.

Soar supports another shortcut, similar in spirit to dot notation that allows you to eliminate variables in conditions whose sole purpose is to link conditions. Dot notation works if there is a single augmentation. However, if there are multiple augmentations, dot notation doesn't help. Grouping, using parentheses, handles this case. In place of where a variable would go as an action, you can insert the augmentations of the value. For example, the original `apply*move-to-food` was:

```
sp {apply*move-to-food
  (state <s> ^io.output-link <ol>
    ^operator <o>)
  (<o> ^name move-to-food
    ^direction <dir>))
-->
  (<ol> ^move.direction <dir>))
```

This can be replaced by

```
sp {apply*move-to-food
  (state <s> ^io.output-link <ol>
    ^operator (^name move-to-food
      ^direction <dir>))
-->
  (<ol> ^move.direction <dir>))
```

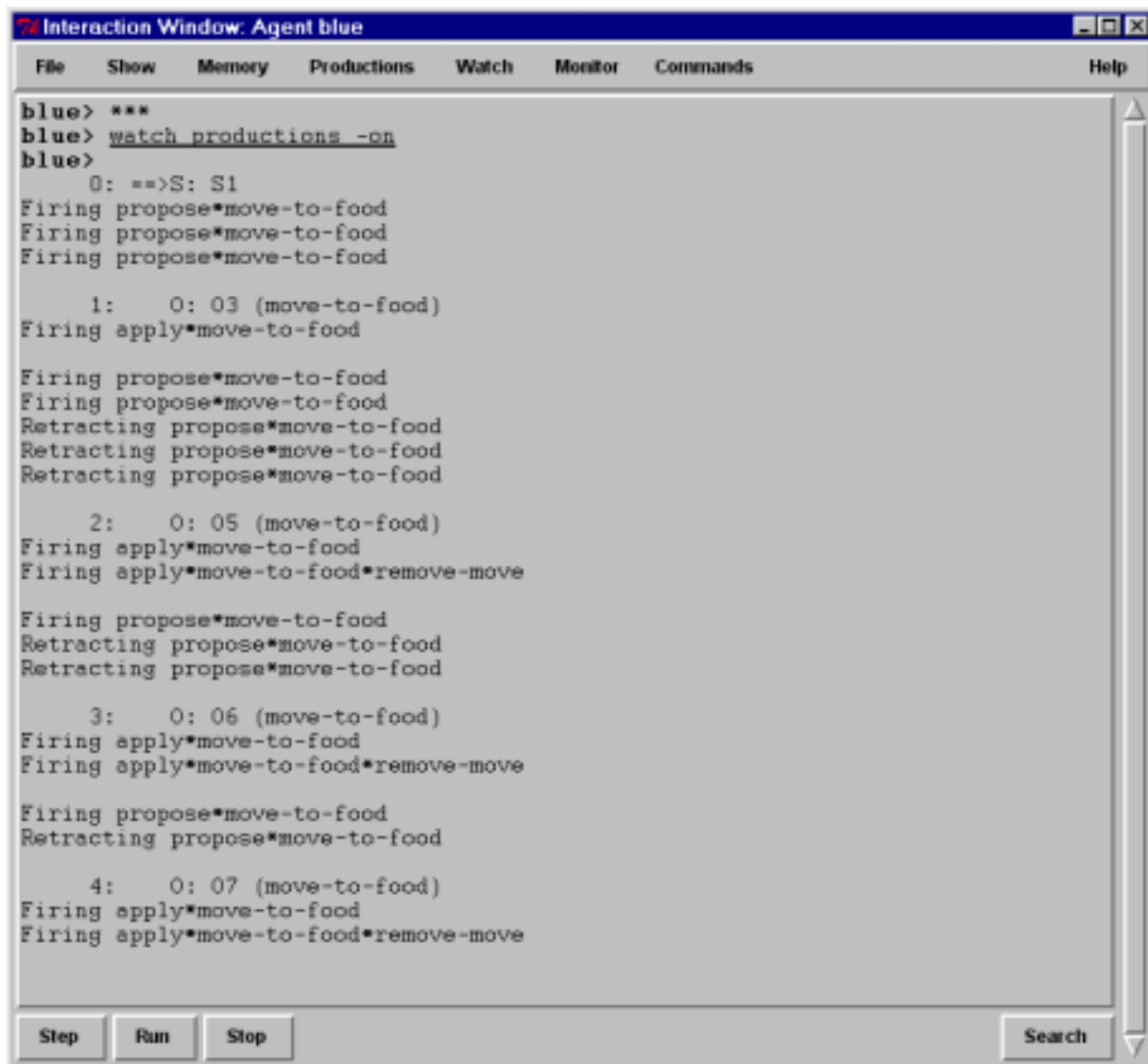
Although this does eliminate unnecessary variables, it often makes the rules more difficult to read, so this notation will not be used in the rest of the tutorial.

One more thing you might want to add is the ability to trace the direction of the selected operator. You can do this with a rule that tests that the operator is selected and uses the write action.

```
sp {monitor*move-to-food
  (state <s> ^operator <o>)
  (<o> ^name move-to-food
    ^direction <direction>))
-->
  (write |Direction: | <direction>))
```

This rule will fire in parallel with `apply*move-to-food` because it matches a selected operator.

## 6.4 Example Trace



The screenshot shows a window titled "Interaction Window: Agent blue". It has a menu bar with "File", "Show", "Memory", "Productions", "Watch", "Monitor", "Commands", and "Help". The main area displays a trace of the agent's internal state and production firing. The trace starts with "blue> \*\*\*" and "blue> watch productions -on". It then shows a series of production firing and retracting events, grouped by a counter (0, 1, 2, 3, 4) and a state identifier (S1). The events include "Firing propose\*move-to-food", "Firing apply\*move-to-food", and "Retracting propose\*move-to-food". The trace ends with "blue>".

```
blue> ***
blue> watch productions -on
blue>
  0: ==>S: S1
Firing propose*move-to-food
Firing propose*move-to-food
Firing propose*move-to-food

  1:    0: 03 (move-to-food)
Firing apply*move-to-food

Firing propose*move-to-food
Firing propose*move-to-food
Retracting propose*move-to-food
Retracting propose*move-to-food
Retracting propose*move-to-food

  2:    0: 05 (move-to-food)
Firing apply*move-to-food
Firing apply*move-to-food*remove-move

Firing propose*move-to-food
Retracting propose*move-to-food
Retracting propose*move-to-food

  3:    0: 06 (move-to-food)
Firing apply*move-to-food
Firing apply*move-to-food*remove-move

Firing propose*move-to-food
Retracting propose*move-to-food

  4:    0: 07 (move-to-food)
Firing apply*move-to-food
Firing apply*move-to-food*remove-move
```

At the bottom of the window, there are buttons for "Step", "Run", "Stop", and "Search".

## 7. Debugging Soar Programs

In trying to write the operators in the previous section, you may have made some mistakes. In this section, you will learn techniques for finding and fixing bugs in Soar programs. The techniques are separated based on the cause of the error:

- When the rules for an eater are read into Soar, Soar checks to make sure that the rules are legal and displays messages when it finds errors. These types of errors are called syntax errors. Visual-Soar will also check for syntax errors in rules. I recommend using Visual-Soar to check for errors first.
- If your rules load into Soar without any error messages, there can still be mistakes in the underlying logic of the rules. These are semantic errors.

This section starts with a subsection on Syntax Errors. Subsections follow this on different techniques for monitoring and examining your program while it is running. The last subsection is a walk through of using these techniques to debug a program.

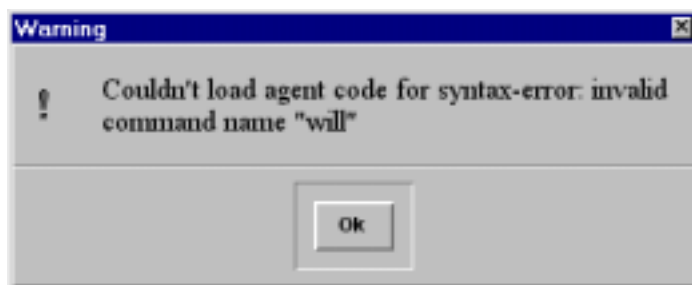
### 7.1 Syntax Errors

There are many different types of syntactic errors that you can have in your programs. In this section, you will get a chance to try to correct the most common types of errors by correcting a file with at least eight different types of errors:

1. Missing comment character
2. Missing {
3. Missing )
4. Extra )
5. Missing }
6. Missing state
7. Missing ^
8. Disconnected rhs action variable.

Unfortunately, Soar does not give very good error messages, so it is often hard to figure out what the problem is. We are working on improving this.

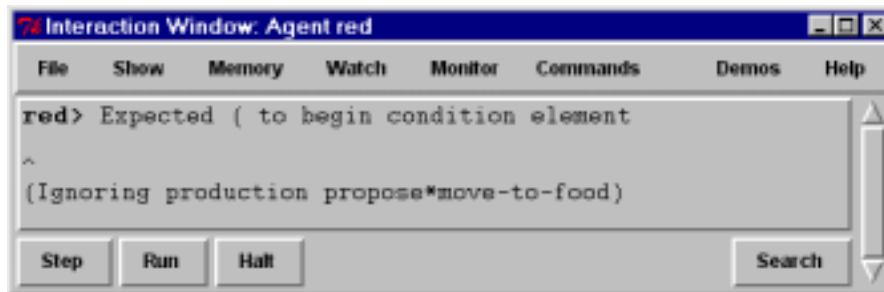
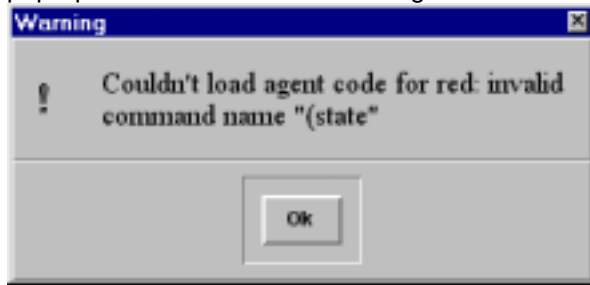
To start, open up the syntax-error.soar file in the tutorial-bugs folder. This has the move operator from the previous section, but with many errors. You will make corrections to the syntax-error.soar file until it loads in with no errors. Next, start up Eaters and select the tutorial-bugs folder. Then try to create a syntax-error eater. When you do, you will immediately get a pop-up error window:



Somewhere, early in the file, Soar ran across the word “will” when it was looking for production rules, starting with “sp”. You will get a similar error whenever there are words in your file that are not preceded by the comment character “#”, or are not legal Soar commands (like “sp” or “watch”). Correct this error by adding # before “will”.

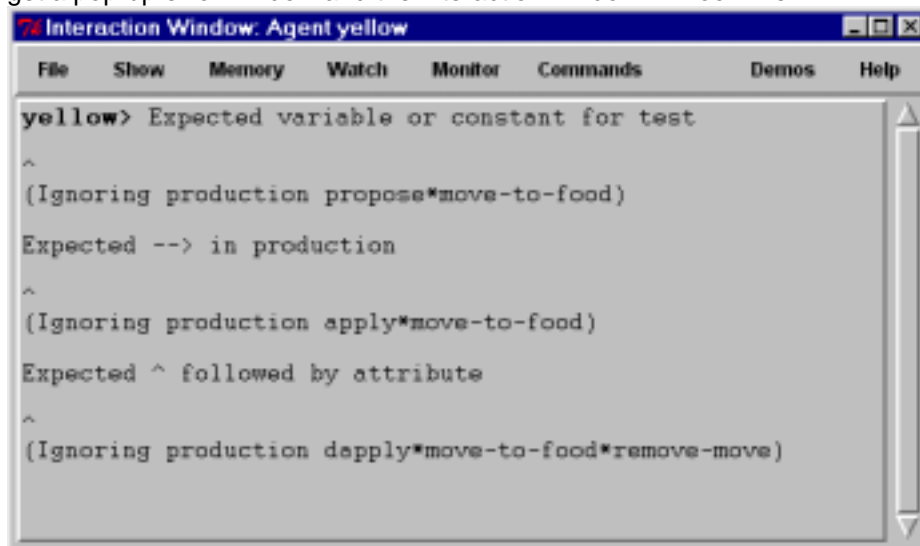


To continue, click ok to the pop-up window, delete the eater, and try to create it again. You will now get a pop-up error window and a message in the interaction window:

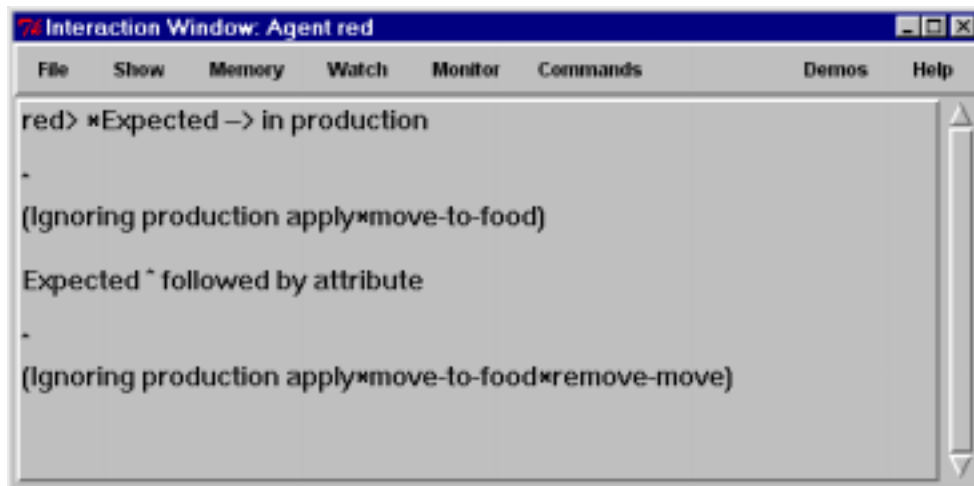


Once again, Soar has been unable to find a legal command, and in this case it can't find "sp {" because the { is missing. Unfortunately, Soar does not provide a helpful error message (the problem has nothing to do with "(state" or a missing "("), so you have to hunt around a bit to find the problem. To fix this problem, add a "{" before the production name.

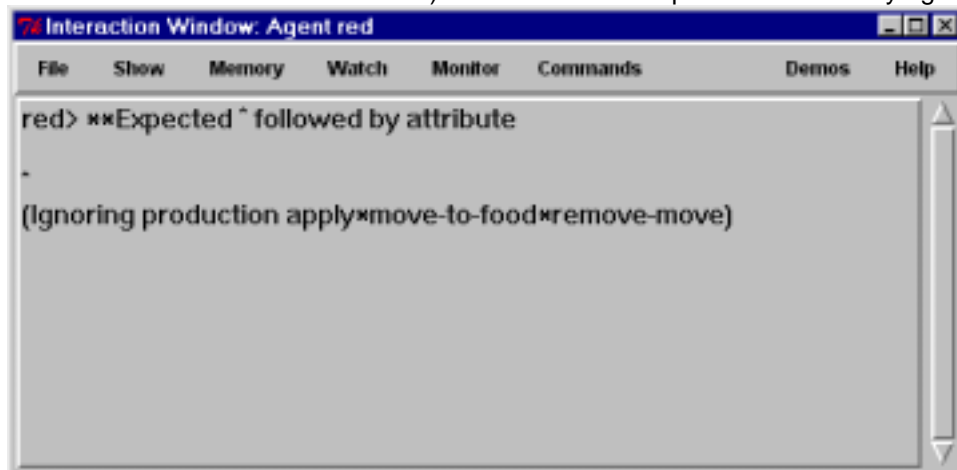
To continue, click ok to the pop-up window, delete the eater, and try to create it again. You will no longer get a pop-up error window and the interaction window will look like:



In this case, Soar did not find an end to the first condition and encountered the -->. Add a ")" and try again.



This time, Soar loaded in the first rule correctly, which you can tell because it printed out a “\*” without any errors. However, it did find an error in `apply*move-to-food`. Soar is confused in the conditions of this rule because there was an extra close parenthesis after `^name move-to-food`, so it thought there should be an arrow instead of the `^direction <dir>`). Remove the extra parenthesis and try again.



Now Soar has loaded in the first two rules and is trying to load in the third. The error message points out that somewhere there is a missing `^` for an attribute. In this case it is the operator. If you add the `^` before the operator in the first condition, the file will now load in correctly.

## 7.2 Write Statements

One of the oldest debugging techniques in traditional programming languages is to add print statements throughout the program. Although not the most effective technique, it can be easily used in Soar by adding write statements in the action of rules. To make the output readable, you want to have a linefeed before writing out any text, using the (crlf) command, which stands for carriage-return and linefeed.

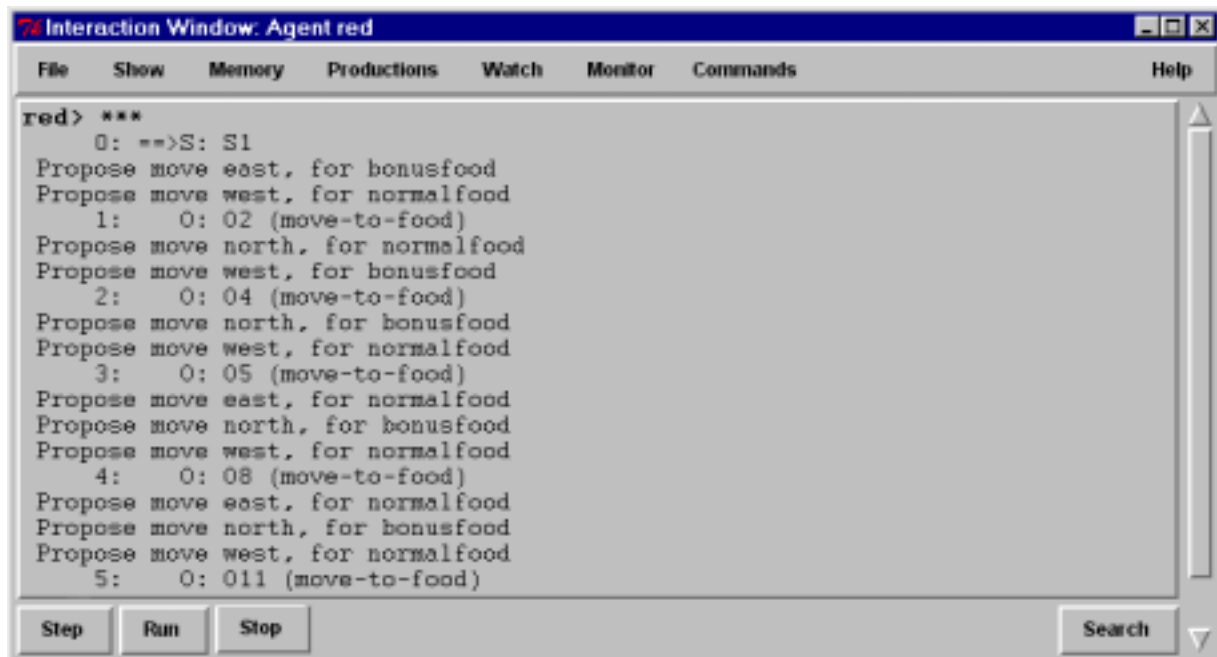
For example, if you want to keep track of all of the move-to-food operators that are proposed, including their direction and expected food, you could take the original rule given below:

```
sp {propose*move-to-food
    (state <s> ^io.input-link.my-location.<dir>.content
        << normalfood bonusfood >>)
-->
    (<s> ^operator <o> + =)
    (<o> ^name move-to-food
        ^direction <dir>)}
```

and modify it by adding a write statement. To get the right values in the action requires a variable for the food type in the condition.

```
sp {propose*move-to-food
    (state <s> ^io.input-link.my-location.<dir>.content
        { <type> << normalfood bonusfood >> })
-->
    (write (crlf) | Propose move | <dir> |, for | <type>)
    (<s> ^operator <o> + =)
    (<o> ^name move-to-food
        ^direction <dir>)}
```

The write command concatenates all of its arguments that can include constants and variables.



### 7.3 Runtime Debugging Commands

While Soar is running, there are many commands you can use to get information on what the current state of the system is, what happened in the past, and what is about to happen. This subsection is organized by the different commands. To learn the details of all of these commands, you should check the Soar User's Manual.

#### ***print***

You have already seen the print command, which is used to print out working memory structures. Print can take a variety of arguments. One of the most useful is to print to a given depth. This prints not only the current identifier and all of its augmentations, but also all of their augmentations recursively to a given depth. For example, you can print out the current state to a depth of 2 by using the following command:

```
red> print -depth 2 s1
```

The order is important and s1 must come at the end of the print statement. The result is:

```
(S1 ^type state ^superstate nil ^io I1 ^operator O2 ^operator O4 +
  ^operator O3 + ^operator O2 + ^operator O1 +)
(I1 ^input-link I2 ^output-link I3)
(O2 ^name move-to-food ^direction south)
(O4 ^name move-to-food ^direction west)
(O3 ^name move-to-food ^direction north)
(O1 ^name move-to-food ^direction east)
```

You can also use the mouse to print working memory objects by right clicking on an identifier. If you select the print option, you will see that there are many other options for printing objects in working memory.

#### ***wmes***

The wmes command can also be used to print out individual working memory elements instead of complete objects. This command also prints out the *timetag* of an individual working memory element. This is a unique number that Soar uses to keep track of each working memory element.

```
red> wmes o1
(120: o1 ^name move-to-food)
(121: o1 ^direction east)
```

Timetags can also be used as arguments in the print and wmes commands. Print will display all of the working memory elements that share the same identifier as the working element with the given timetag, while, wmes will display only the single working memory element.

```
red> print 120
(O1 ^name move-to-food ^direction east)

red> wmes 120
(120: o1 ^name move-to-food)
```

**matches**

One of the questions you will frequently want to ask is, “What rules are about to fire?” The matches command will return a list of all rules that are ready to fire, separated into those that will be operator applications (O Assertions), those that will create I-support augmentations (I Assertions), and those that will remove I-supported augmentations (Retractions).

```
red> matches
O Assertions:
I Assertions:
Retractions:
```

Matches also can be invoked through the “show” item on the menu list, or by right clicking on the interaction window, then selecting the production item on the menu that appears.

Matches can also take as an argument the name of a rule. It will then print out the rule, condition by condition, with information on how many working memory elements match that condition and are consistent with all of the variables in the previous conditions. When Soar loads in a rule, it automatically reorders the conditions of the rule so that it can efficiently match it, so the ordering of the conditions from matches will not be the same as the ordering of the rule when you wrote it.

If a rule completely matches, then it has either already fired, or is about to fire.

```
red> matches propose*move-to-food
1 (state <s> ^io <i*1>)
1 (<i*1> ^input-link <i*2>)
1 (<i*2> ^my-location <m*1>)
7 (<m*1> ^<dir> <d*1>)
3 (<d*1> ^content { << normalfood bonusfood >> <c*1> })
```

3 complete matches.

In this example, the first three conditions have a single match, and then the fourth condition matches seven different working memory elements. These are all of the augmentations of my-location. The final condition then restricts the matches to be only those that have content normalfood or bonusfood.

If a rule does not completely match, the condition that failed to match will be preceded by “>>>>” as in:

```
red> matches apply*move-to-food*remove-move
1 (state <s> ^operator <o*1>)
1 (<o*1> ^name move-to-food)
1 (<s> ^io <i*1>)
1 (<i*1> ^output-link <ol>)
1 (<ol> ^move <move>)
>>>> (<move> ^status complete)
```

0 complete matches.

Matches can also print out working memory elements that match the conditions by using the -timetags (for just the timetags) or -wmes (for complete working memory elements) arguments. Matches can be invoked by clicking with the right mouse button on the name of a production and then selecting “production” from the menu that appears.

**preferences**

There is a special command to print out the preferences for selecting an operator. This command also works for any working memory element (prior versions of Soar allowed preferences for all working memory elements and there are still vestiges of them in the current version). To use the preference command, you give an identifier and an attribute. Soar will then print out all of the preferences for all values of that identifier attribute pair. For example, to print out all of the preferences for the operator attribute:

```
red> preferences s1 operator
Preferences for S1 ^operator:
acceptables:
  O7 (move-to-food) +
  O8 (move-to-food) +
  O9 (move-to-food) +
unary indifferents:
  O7 (move-to-food) =
  O8 (move-to-food) =
  O9 (move-to-food) =
```

This example shows that there are three operators proposed (O7, O8, O9), and each one has an acceptable and indifferent preference.

The preferences command has another feature that makes it extremely useful. Using the -name argument, it will tell you the name of the production that created the preference. For example, if you want to discover why there is the working memory element (I3 move m3), you can type:

```
red> preferences I3 move -names
Preferences for I3 ^move:
acceptables:
  M3 +
    From apply*move-to-food
```

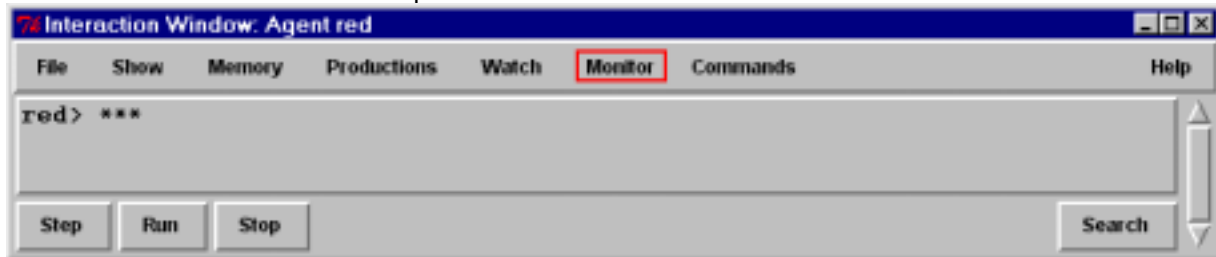
This tells you that apply\*move-to-food creates this working memory element. If you want to find out what working memory elements matched apply\*move-to-food when it created (I3 move m3), you can type:

```
red> preferences I3 move -wmes
Preferences for I3 ^move:
acceptables:
  M3 +
    From apply*move-to-food
  (212: S1 ^operator O8)
  (205: O8 ^name move-to-food)
  (206: O8 ^direction north)
  (3: S1 ^io I1)
  (5: I1 ^output-link I3)
```

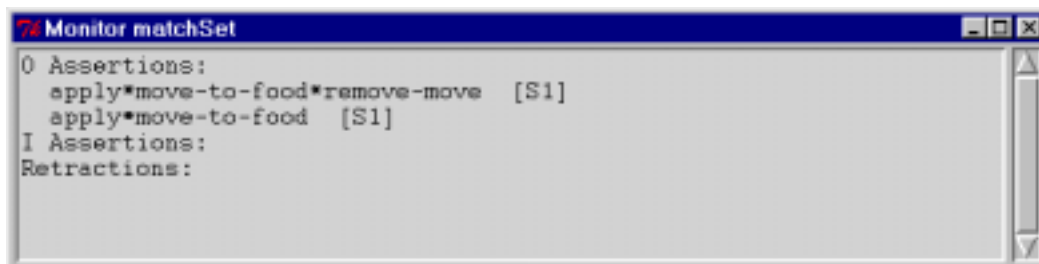
You can also invoke preferences by clicking with the right mouse button on an attribute of a working memory element in the interaction window. Select the preferences command from the menu that appears and then select the appropriate preferences command.

## 7.4 Monitoring

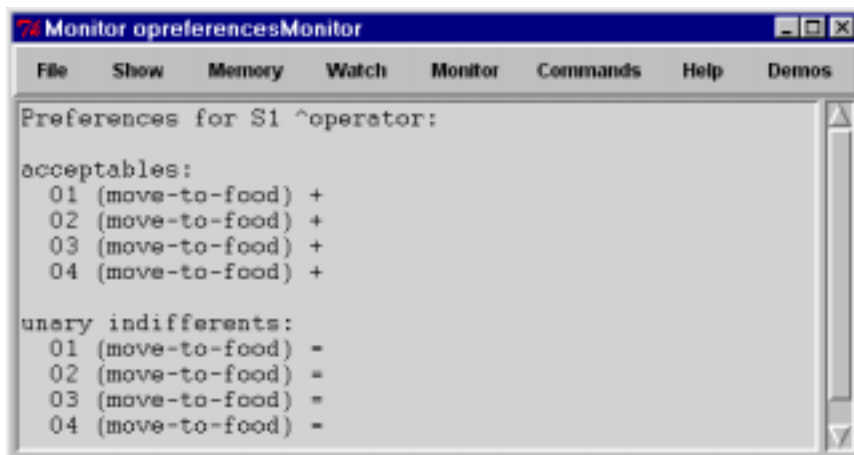
Soar also has some automated monitoring mechanisms that can be invoked through the View item on the Interaction Window bar. After clicking on this item, a menu will appear that allows you to enable the monitoring of the state stack (print-stack), the current set of productions about to fire (match set), or the preferences that have been created for the selection of the current operator (Op preferences). The first of these (print-stack) is not very useful for the eaters because they do not use Soar's subgoaling, but it will be useful for TankSoar in the next part of the tutorial.



The match set window will continually display all rules that currently match working memory and are about to fire. This is extremely useful because it let's you "look into the future". Below is an example match set monitor window for move-to-food when the "Stop After Decision" option is used. Usually, no productions will be in the match set when you halt Soar because all of the rules have fired and output is about to be called. However, if you select "Stop After Decision" under the Eaters button of the Control Panel, Soar will also stop after an operator is selected. In this case, you can see that the rule to remove the previous move command on the output-link is about to fire (apply\*move-to-food\*remove-move) and the rule to add the new move command to the output-link is about to fire (apply\*move-to-food).



The Op preferences window continually displays the preferences for selecting the next operator. This is helpful when you are trying to figure out which operator is about to be selected, or why the current operator is selected.



## 7.5 Semantic Errors

In this section you will use the techniques from the prior section to fix semantic errors. In a rule-based system, semantic errors have three general results:

1. A rule does not fire when it should.
2. A rule fires when it shouldn't.
3. The action of the rule is incorrect.

Unfortunately, when you have an error, you don't know which type it is. Luckily, there are general techniques for finding all types of semantic errors.

The first, and most important step, in finding semantic errors is knowing what you want the program to do at each step. Only by knowing what the program should do, will you be able to detect when it does something wrong. Usually you will notice that the wrong operator is selected, or that no operator is selected (a substate is created).

I've created a file called "semantic-error.soar" in the Tutorial directory for move-to-food that has some bugs. Create an eater with this file and also open the file in a text editor so you can modify as we go along. You should expect that propose\*move-to-food to fire during the first step, so be bold and click on the step button and see what happens. You should get a trace like:

```
red> ***
      0: ==>S: S1
      1:      ==>S: S2 (state no-change)
```

This clearly indicates that propose\*move-to-food did not fire. To find out why, try matches propose\*move-to-food.

```
red> matches propose*move-to-food
      1 (state <s> ^io <i*1>)
      1 (<i*1> ^input-link <i*2>)
      1 (<i*2> ^my-location <m*1>)
      7 (<m*1> ^<dir> <d*1>)
>>>> (<d*1> ^contant { << normalfood bonusfood >> <c*1> })
```

```
0 complete matches.
```

There is a problem with the last condition. You can examine working memory to find out what it should be matching and after some examination, you should realize that the name of the attribute should be "content", not "contant". Change the rule in the file. If you have been using Visual-Soar, it would have found this error while you were creating the Eater. To avoid destroying and creating a new eater, you can just reload the rules by clicking on the agent button in the menu bar of the Interaction Window. This will bring up a list of active eaters and you can select your eater and then select the "reload" button. Now you can try going one step again. You should have the following in your interaction window:

```
red> ***
      0: ==>S: S1
      1:      O: O1 (move-to-food)
```

At this point, you can check which rules are about to fire by using the matches command.

```
red> matches
0 Assertions:
  apply*move-to-food
1 Assertions:
Retractions:
```

That looks fine, so take one more step. Unfortunately, the eater doesn't move. What's the problem? You should examine the output-link to see if the move command was correctly created:

```
red> print -depth 2 i3
(I3 ^moves M1)
(M1 ^direction east)
```

Unfortunately, you have to inspect this structure and realize that the command it creates is "moves" not "move". Correct this, reload, and take another step. Now the eater moves and runs appropriately.



## 8. Generalized Move Operator

The move-to-food operator you created in the last section would get stuck when there was no food in the cells adjacent to the eater. It also did not prefer bonusfood to normalfood. In this section, you will generalize the move-to-food operator to be an operator that can move to a cell with any type of content. Once you have created such a generalized move operator, we will introduce additional preferences that allow you to create a greedy eater that never gets stuck.

### 8.1 Move Operator Proposal

The proposal for the move operator needs to test that there is an adjacent square that it can move into. It should not propose moving into a wall. There are two possible approaches to write this test. The first is to test all of the content values that are ok to move into: normalfood, bonusfood, eater, or empty. The second is to test that the content does not equal wall. Taking the first approach gives us the following English version of the proposal:

```
# Propose*move*1:
# If there is normalfood, bonusfood, eater, or empty in an adjacent cell,
#   propose move in the direction of that cell
#   and indicate that this operator can be selected randomly.
```

It is straightforward to translate this into Soar based on the move-to-food operator proposal:

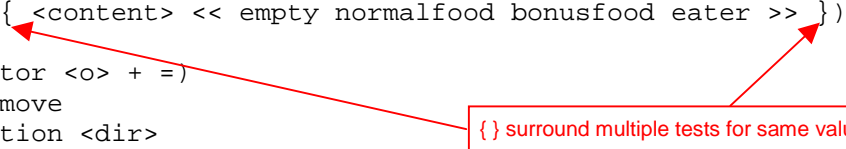
```
sp {propose*move*1
  (state <s> ^io.input-link.my-location.<dir>.content
    << empty normalfood bonusfood eater >>)
-->
  (<s> ^operator <o> + =)
  (<o> ^name move
    ^direction <dir>)}
```

Although this is adequate, it will make future selection rules simpler if the operator also contains the contents of the cell. Thus, an alternative version is:

```
# Propose*move*1a:
# If there is normalfood, bonusfood, eater, or empty in an adjacent cell,
#   propose move in the direction of that cell, with the cell's content,
#   and indicate that this operator can be selected randomly.
```

To translate this into Soar requires matching the cell's content to a variable and then using that variable in the action as an augmentation of the operator, such as ^content <content>. However, the value is already matched by << empty normalfood bonusfood eater >>. What is needed is a way to match both a variable and the list at the same time. In Soar this is done by surrounding the two (or more) things to match against the same item with curly braces: "{ }". Thus, the Soar version of the proposal becomes:

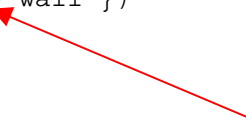
```
sp {propose*move*1a
  (state <s> ^io.input-link.my-location.<dir>.content
    { <content> << empty normalfood bonusfood eater >> })
-->
  (<s> ^operator <o> + =)
  (<o> ^name move
    ^direction <dir>
    ^content <content>)}
```



{ } surround multiple tests for same value

Although this rule is adequate, it forces you to list all of the contents except walls. This rule will have to be changed if we ever add other types of food (e.g., superbonusfood). It might be better to write a rule that tests that the content is *not equal* to wall. This can be done in Soar by using "<>". The not equal test can also be combined with the variable as in propose\*move\*1a, giving the following rule:

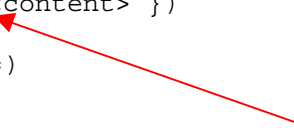
```
sp {propose*move*2a
  (state <s> ^io.input-link.my-location.<dir>.content
    { <content> <> wall })
-->
  (<s> ^operator <o> + =)
  (<o> ^name move
    ^direction <dir>
    ^content <content>)}
```



Will match any value except wall.

The "<>" must be placed directly *before* the value it compares, and in the example above, it is correctly before wall. The rule can also be written with the variable *after* the test that the content is not equal to wall:

```
sp {propose*move*2a
  (state <s> ^io.input-link.my-location.<dir>.content
    { <> wall <content> })
-->
  (<s> ^operator <o> + =)
  (<o> ^name move
    ^direction <dir>
    ^content <content>)}
```



The order of tests in { } does not matter.

The not equal test can also be used with variables. For example, if you want to test that the content of the cells to the north and south of the eater are not equal, you could use the following conditions:

```
(state <s> ^io.input-link.my-location <my-loc>)
(<my-loc> ^north.content <north>
  ^south.content <> <north>)
```

If you wanted to match the contents of both the north and south cells for use in the action, you could use the following conditions:

```
(state <s> ^io.input-link.my-location <my-loc>)
(<my-loc> ^north.content <north>
  ^south.content { <south> <> <north> })
```

Remember, the not equal test, "<>", must directly precede the symbol or variable it refers to. Soar also has tests for greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=). These can be used when matching numbers and also precede the value they refer to. For example, to test that the eater's score is greater than 25, the following condition can be used.

```
(state <s> ^io.input-link.eater.score > 25)
```

## 8.2 Move Operator Application

This application rule for the move operator is a copy of the move-to-food operator application. The English and Soar versions are the same except for the name of the operator:

```
# Apply*move
# If the move operator for a direction is selected,
#   generate an output command to move in that direction.

    sp {apply*move
      (state <s> ^io.output-link <ol>
        ^operator <o>)
      (<o> ^name move
        ^direction <dir>)
    -->
      (<ol> ^move.direction <dir>)}

# Apply*move*remove-move:
# If the move operator is selected,
#   and there is a completed move command on the output link,
#   then remove that command.
.
    sp {apply*move*remove-move
      (state <s> ^io.output-link <ol>
        ^operator.name move)
      (<ol> ^move <direction>)
      (<direction> ^status complete)
    -->
      (<ol> ^move <direction> -)}
```

The proposal rule together with these two rules will give you an eater that randomly moves around, avoiding walls. We can greatly improve its behavior by using more preferences.

### 8.3 Move Operator Selection

To improve the performance of the eater, we can add rules that prefer moving to bonusfood over normalfood or an empty cell and prefer moving to normalfood over moving into an empty cell or a cell with another eater. The preference language in Soar is rich enough to support a variety of ways of ordering the choices and we will explore different possibilities in this section.

To get started, you need to create a rule to prefer bonusfood to normalfood or empty or an eater. The condition part of the rule must match against operator proposals, while the action part must prefer the operator that moves to the bonusfood.

In English this would be:

```
# Select*move*bonusfood-better-than-normalfood
# If there is a proposed operator to move to a cell with bonusfood and
#   there is a second proposed operator to move to a cell that is empty or
#   has normalfood or another eater
#   prefer the first operator.
```

The conditions of this operator must match against proposed operators before they have been selected. A proposed operator can be matched by matching the acceptable preference for the operator, which is written in the condition as the `^operator` augmentation of the state, with a value for the operator identifier, followed by a plus sign, “+”.

```
(state <s> ^operator <o1> +)
```

Acceptable preferences are the only preferences that are added to working memory. All of the other preferences (better, best, worse, worst, and reject) are not added to working memory. They are held in preference memory and persist as long as the rule instantiations that created them still match. They are not in working memory because there is little advantage to having them available for other rules to match against, whereas the acceptable preferences must be in working memory because they define what operators are candidates for selection.

An operator can be preferred by creating a *better* than preference and the decision procedure will use that preference in determining which operator to select – it will never select an operator if it is worse than another candidate operator, unless that candidate is rejected. In addition to the better preference, there is a *worse* preference that is exactly the opposite. In an action, the better preference is the greater than sign: “>”. It is used in the same place as an acceptable or indifferent preference, except that a variable that matched the identifier of the better operator is put before the greater than sign, and the variable that matched the identifier of the worse operator is put after the greater than sign. Therefore, the Soar rule is as follows:

```
sp {select*move*bonusfood-better-than-normalfood-empty-eater
  (state <s> ^operator <o1> +
    ^operator <o2> +)
  (<o1> ^name move
    ^content bonusfood)
  (<o2> ^name move
    ^content << normalfood empty eater >>)
-->
  (<s> ^operator <o1> > <o2>)}
```

If there are adjacent cells with both bonusfood and normalfood, this rule will fire right after propose\*move creates acceptable preferences, but during the same proposal phase so that it will influence the next operator selection. It will fire multiple times if there are multiple cells with bonusfood or normalfood. After these preferences are created, the decision procedure will gather them up to make a decision.

You can use exactly the same approach to prefer moving to cells with normalfood over moving to empty cells or cells with eaters. Soar provides an alternative with the *worst* preference, which means don't select the operator unless there are no other choices. In this case, you can create worse preferences for operators that move the eater into an empty cell or a cell with another eater. With these additional preferences, operators that move into cells with bonusfood or normalfood will always be selected if they exist, and otherwise the eater will randomly select between moving into an empty cell or into a cell with another eater. The new selection rule can be written in English as:

```
# Select*move*avoid-empty-eater
# If there is a proposed operator to move to an empty cell or a cell with
#   another eater,
#   then avoid that operator.
```

Here we use the word avoid to mean that the operator will only be selected if there is nothing worse. A worst preference is written as a less than sign: "<". This is exactly the same as a worse preference, except that there is no second variable that the first variable is compared to.

```
sp {select*move*avoid-empty-eater
  (state <s> ^operator <ol> +)
  (<ol> ^name move
    ^content << empty eater >>)
-->
  (<s> ^operator <ol> <)>
```

+ signifies acceptable preference

< signifies the worst preference

Just as there is a worst preference, there is also a *best* preference. The best preference means that an operator should be selected as long as there is no other operator better than it (or it is not worse than another operator is). Thus the meaning of best is a bit odd in that better preferences are more important than the best preferences and an operator with a best preference will not be selected if another operator is better than it.

You could have used a best preference for a move into a cell with normalfood instead of using the worst preference. The move to normalfood would then be selected over a move to an empty cell or a cell with an eater. Select\*move\*bonusfood-better-than-normalfood-empty-eater will ensure that bonusfood is preferred to normalfood.

```
# Select*move*prefer-normalfood
# If there is a proposed operator to move to a cell with normalfood,
#   prefer that operator.
  sp {select*move*avoid-empty-eater
    (state <s> ^operator <ol> +)
    (<ol> ^name move
      ^content normalfood)
-->
    (<s> ^operator <ol> >)>
```

> signifies the best preference

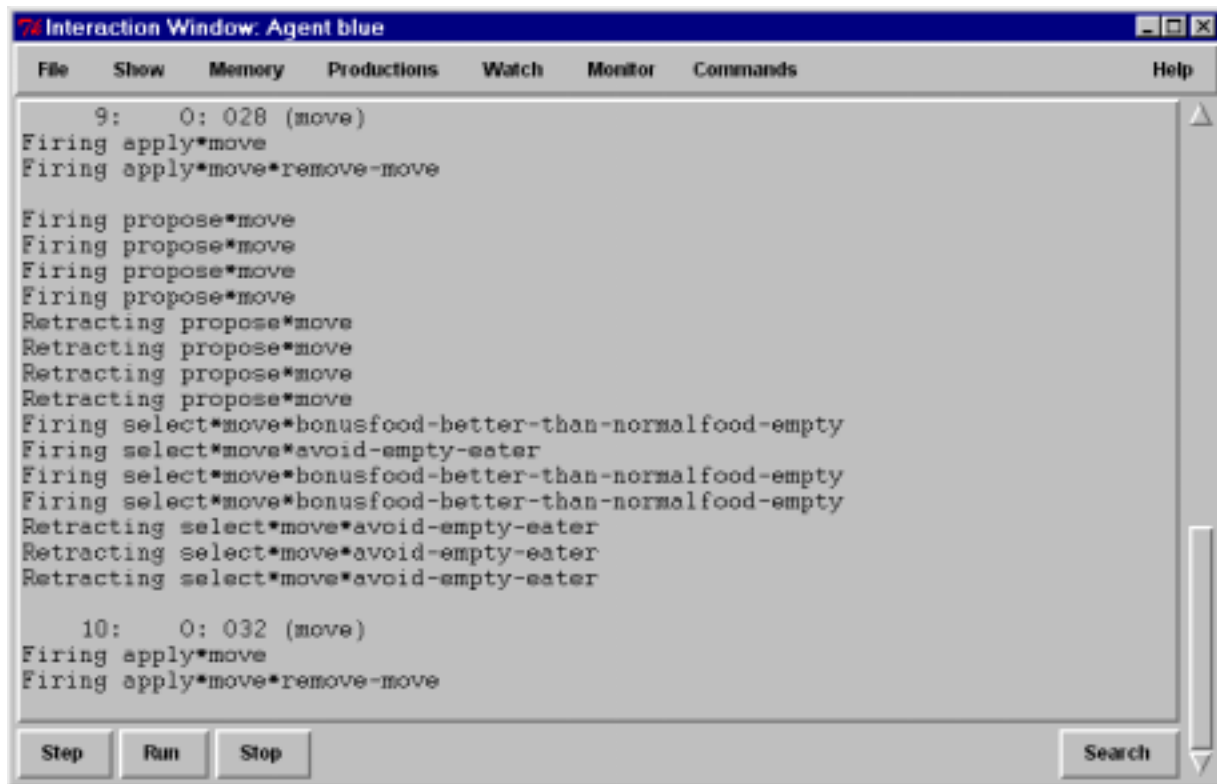
## 8.4 Summary of Preferences

This section summarizes the meaning of the preferences presented earlier. There are two additional preferences (require and prohibit) that are only rarely used and are not covered in this tutorial, but are described in the Soar 8 Manual. The preferences can be thought of as a sequence of filters, processed in the following order.

- **Acceptable (+)** An acceptable preference states that a value is a candidate for selection. Only values with an acceptable preference have the potential of being selected. If there is only one value with an acceptable preference, that value will be selected if it does not also have a reject preference.
- **Reject (-)** A reject preference states that the value is not a candidate for selection. A value will not be selected if it has a reject preference.
- **Better (>), Worse (<)** These preferences state that a value should not be selected if the better value is a candidate. If the better value does not have an acceptable preference, or is rejected, the better/worse preference is ignored. Otherwise, the worse value is removed from consideration. 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 should be selected if it is not rejected, or if there is no other value better than it. If a value is best (and not rejected, or worse than another), it will be selected over any other value that is not also best. If two values are best, then the remaining preferences (worst, indifferent) will be examined to determine the selection. If a value (that is not rejected) 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.)
- **Worst (<)** A worst preference states that the value should be selected only if there are no alternatives. A worst will only be considered if the above preferences have not filtered the choices to a single choice. In this case, any value with a worst preference will be discarded, unless all of the values have worst preferences.
- **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. Indifferent preferences are used to signal that it does not matter which operator is selected, which results in a random selection is made from among the alternatives.

## 8.5 Example Run

The following trace shows how Soar fires and retracts many rules in parallel, with operator proposals following selections.



## 9. Advanced Move Operator

The eater you created in the last section will greedily consume food and never get stuck. However, its behavior doesn't always appear to be very intelligent, especially if it is surrounded by empty cells. The specific problem addressed in this section is that the eater will sometimes move randomly back and forth when it is surrounded by empty cells. That is clearly wasted effort and the eater should at least move to a cell different from the one it just came from. There are other approaches to improving the eater, such as having it test for food two spaces away, but the point of this section is to avoid moving back to the same cell, which will give you experience with creating persistent structures in working memory.

In order for the eater to avoid moving back to the cell it just came from, the eater must remember which direction it moved to get to the current cell. That information is available as an augmentation of the move operator while the eater is moving, but it disappears, along with the operator, once the move is completed. What is needed is a persistent augmentation of the state that records the direction of the last operator. This direction can then be used to avoid moving in the opposite direction. You will need to change the operator proposal to avoid moving back to the prior cell and the operator application to record the direction of the prior cell. A further change is needed to initialize some data structures in working memory that will make it easy to determine the opposite of each direction, which is needed to avoid moving back to the last cell.

In this section we start with the initialization rules, and then cover the operator application rules and operator proposals.

### 9.1 Initialization: Opposite Directions

The rules that you will write for operator application and selection need to determine the opposite of the direction the eater moved last, because that is the direction that needs to be avoided. The easiest way to do this is to create a structure in working memory that contains each direction, and each direction has an attribute that is its opposite direction. This structure can be on the state and the following rule creates one possible structure that encodes this information:

```
sp {initialize*state*directions
  (state <ss> ^type state)
-->
  (<ss> ^directions <n> <e> <s> <w>)
  (<n> ^value north ^opposite south)
  (<e> ^value east ^opposite west)
  (<s> ^value south ^opposite north)
  (<w> ^value west ^opposite east)}
```

Multiple values for a single attribute can be listed following the attribute.

This is the first rule you've seen where there is an action with multiple values (<n> <e> <s> <w>) for a single attribute (^direction). Each value can be written as individual actions; however, as a shortcut, all of the values can be listed following their common attribute.

One might be tempted to have the values of ^direction be the symbols north, east, south, and west. However, these values cannot be further augmented with their opposites; only identifiers can have augmentations. It is not legal to have augmentations of the form: (north ^opposite south).

Rules such as initialize\*state\*directions are quite common in Soar program because they create fixed working memory structures. These structures simplify operators by making it possible to directly match relations such as opposite instead of having to write rules that are specific to each of the directions. This rule will fire during the first cycle when (s1 ^type state) is added to working memory. The structures it creates will have i-support because the rule conditions do not test an operator. However, the structures will never be removed because the conditions match throughout the life of the eater.



## 9.2 Advanced Move Operator Application

Our planned extension to the move operator does not require any changes to the existing application rules. However, you must add two rules to maintain a memory of the last direction moved. One of the rules must create the memory by adding the persistent (o-supported) structure when an operator is applied, and one must remove the old value when the next operator is selected. (You may think that you could write a single rule that both creates the last-direction and removes any old value for last-direction, but then a second rule would have to be written to cover the initial situation when there is no existing last-direction.) Try to write the English and Soar versions of these two rules.

```
# Apply*move*create*last-direction
# If the move operator for a direction is selected,
#   create an augmentation called last-direction with that direction.

sp {apply*move*create*last-direction
  (state <s> ^operator <o>)
  (<o> ^name move
    ^direction <direction>)
-->
  (<s> ^last-direction <direction>)}

# Apply*move*remove*last-direction
# If the move operator for a direction is selected,
#   and the last-direction is not equal to that direction,
#   then remove the last-direction.

sp {apply*move*remove*last-direction
  (state <s> ^operator <o>
    ^last-direction <direction>)
  (<o> ^direction <> <direction>
    ^name move)
-->
  (<s> ^last-direction <direction> -)}

```

Since both of these rules test the current operator, their actions are persistent. The first rule creates an augmentation of the state with attribute `^last-direction`. This augmentation will always contain the direction of the last operator applied by copying the direction from the move operator. The next rule removes the `^last-direction` attribute if it does not equal the direction on the current operator by using a reject preference. These rules will fire during the operator application phase.

Why can't you use an i-supported rule? Conceptually, you need the `^last-direction` augmentation to persist after the operator is no longer selected because it is going to be tested to select and apply the next operator. More pragmatically, Soar will automatically make the augmentation o-supported because it is created as part of an operator application.

This section and the previous one demonstrate two different ways to create persistent structures in working memory. If the structure can persist throughout the life of the eater, you can use a rule that tests that the state exists and then creates the structure in working memory in the action. Whenever the structure must change during the life of the eater, you must use a pair of rules that are part of the application of an operator - one to add the structure when appropriate, and one to remove the structure when it is no longer appropriate.

### 9.3 Advanced Move Operator Proposal

There are two ways to modify the proposal and selection rules so that the eater does not move backward. One requires modifying the proposal rule so that the operator is never proposed, and the second is to create a new selection rule that creates a preference that prevents the operator from being selected. We will look at both possibilities.

In order to modify the proposal rule, recall the final move proposal:

```
# Propose*move:
# If the content of an adjacent cell is not a wall,
#   propose move in the direction of that cell, with the cell's content,
#   and indicate that this operator can be selected randomly.
```

A condition can easily be added that tests that the direction of the adjacent cell (<d>) is not equal to the opposite of the direction of the last move. However, this will not work for the first move when there is no direction for the last move because no operator has applied to create the memory of a previous move. An alternative is test that there does not exist an opposite of last direction that is equal to the direction of the adjacent cell. How is that different? The first tests that there does exist in working memory an augmentation of the state, but its value is not equal. The second tests that there does not exist in working memory an augmentation that is equal. The second one will be correct for the first move. Thus the revised English description is:

```
# Propose*move*no-backward:
# If there is normalfood, bonusfood, eater, or empty in an adjacent cell,
#   and there is no last direction equal to the opposite direction for that
#   cell,
#   propose move in the direction of that cell, with the cell's content,
#   and indicate that this operator can be selected randomly.
```

To test for the absence of an augmentation, Soar uses a dash, "-". The dash, called negation, precedes the augmentation that must not be in working memory for the rule to match. The new proposal rule is as follows:

```
sp {propose*move*no-backward
  (state <s> ^io.input-link.my-location.<dir>.content { <co> <> wall }
    ^directions <d>
    - ^last-direction <o-dir>)
  (<d> ^value <dir>
    ^opposite <o-dir>)
-->
  (<s> ^operator <o> + =)
  (<o> ^name move
    ^direction <dir>
    ^content <co>)}
```

- means that this rule will match only if no working memory elements match this attribute value.

<dir> matches the direction of the adjacent cell that is not a wall.

<o-dir> matches the opposite of direction of the adjacent cell that is not a wall.

A negation can also precede a complete object, where the identifier is followed by multiple attribute-values. In that case, its is a test that there is no object with all of those attributes in working memory. The semantics of using negation can be a bit tricky and the details of it are spelled out in the manual.

## 9.4 Advanced Move Operator Selection

The alternative to modifying the proposal rule is to add a rule that eliminates from consideration any operator that moves opposite to the last direction. This requires a new preference, called reject, represented by a dash, "-". This is different from the operator application action that is also called rejection because that rejection removes a working memory element. When a reject preference is created for an operator, the decision procedure will not select that operator, no matter which other preferences have been created. However, the reject preference will be retracted when the rule that creates it no longer fires, allowing the rejected operator to be selected when the situation changes. Reject is used when an operator should not be selected for the current situation even if there are no other options. In contrast, a worst preference is used when an operator can be selected, but only if there are no better options.

One side effect of reject is that it disables better/worse preferences where the rejected operator is better than some other operator. Normally, that other operator would not be selected, but if the better operator is rejected, the better/worse preference is ignored.

Try writing both English and Soar versions of this rule.

```
# Select*move*reject*backward
# If there is a proposed operator to move in the direction
#   opposite the last move,
#   reject that operator.
```

```
sp {select*move*reject*backward
  (state <s> ^operator <o> +
    ^directions <d>
    ^last-direction <dir>)
  (<d> ^value <dir>
    ^opposite <o-dir>)
  (<o> ^name move
    ^direction <o-dir>)
-->
  (<s> ^operator <o> -)}
```

Uses <dir> from operator to match <dir> in directions and then match opposite

<odir> must match opposite of last-direction and direction of proposed operator

- means reject so that this operator will not be selected.

Note that this rule will not reject the proposed operator for the first move because the attribute ^last-direction will not initially exist.

## 10. Jump Operator

All of the eaters you've created so far have a single type of operator: move. In many situations, more than one instance of the operator is created, but still, there was only one operator. Adding operators does not change the way Soar operates, but it does give you a chance to use what you've learned on a slightly different problem. In this section, you are going to write the jump operator. An eater can jump over a cell to a cell two moves away; however, a jump costs the eater 5 points (the same as it gains from a normal food). The eater can jump over a wall, but it cannot jump into a cell with a wall.

In writing the jump operator, you should write and test the proposal and application rules without including the move operator. This will let you debug the jump rules first, without getting things mixed up with the rules for the move operator. You should then write operator selection rules and combine your jump operator with the move operator. In writing the jump operator, you will want to use the initialization rule for directions from the previous section, so you will want to copy that to your file in which you are writing the jump operator.

The rest of this section goes through the rules for jump. If you feel confident in your knowledge of the aspects of Soar that have been presented so far, you should try to write your own jump operator before reading this section. As you are writing your own jump operator, you should try to reuse or generalize some of the rules you wrote for the move operator. You might find it is a bit tricky for your eater to correctly keep track of the previous direction it moved or jumped. If you get stuck, or are unsure about what to do, you should read these sections.

After you finish this section, you can try your own strategies by creating different eaters with different control knowledge. You can even have contests between them to see which strategies work best.

### 10.1 Jump Operator Proposal

There are only two differences between the proposal for the move operator and the jump operator.

- The first difference is that the name of the operator should be jump, not move.
- The second is that instead of testing that an adjacent cell does not contain a wall, the jump operator needs to test that a cell two moves away in a direction does not contain a wall. This is easy to add because every cell has the same four directional pointers, so the desired cell can be tested via the direction augmentation on the adjacent cell: instead of just `<dir>` use `<dir>.<dir>` which tests two steps in the same direction because the same directional pointer must match both uses of `<dir>`.

In these examples, we will use the proposal that does not test for the absence of backward moves. We will include that in the selection knowledge. Thus the English version is:

```
# Propose*jump:
# If the content of a cell two steps away in a direction is not a wall,
#   propose jump in the direction of that cell, with the cell's content,
#   and indicate that this operator can be selected randomly.
```

```
sp {propose*jump
  (state <s> ^io.input-link.my-location.<dir>.<dir>.content
    { <content> <> wall })
-->
  (<s> ^operator <o> + =)
  (<o> ^name jump
    ^direction <dir>
    ^content <content>))}
```

The first `<dir>` leads to an adjacent cell.

The second `<dir>` leads to a cell two away from the eater in the same direction.

## 10.2 Jump Operator Application

The application of jump is exactly the same as the application of move, except that the jump name must be issued instead of a move. Instead of creating a new rule for each operator, you can reuse and generalize the original operator application rules by allowing them to match an operator named either move or jump, and then copying the operator name to the output link.

```
# Apply*move*jump
# If the move or jump operator for a direction is selected,
#   generate an output name to move in that direction.
```

```
sp {apply*move
  (state <s> ^io.output-link <ol>
    ^operator <o>)
  (<o> ^name { <name> << move jump >> }
    ^direction <dir>)
-->
  (<ol> ^<name>.direction <dir>))}
```

<name> will match  
move or jump

<name> be move or jump as  
matched in condition

```
# Apply*move*jump*remove-name:
# If the move or jump operator is selected,
#   and there is a completed name on the output link,
#   then remove that name.
```

```
sp {apply*move*remove-move
  (state <s> ^io.output-link <ol>
    ^operator.name <name>)
  (<ol> ^<name> <direction>)
  (<direction> ^status complete)
-->
  (<ol> ^<name> <direction> -))}
```

<name> will match  
move or jump

### 10.3 Simplified Operators

In writing the operator application rules for the jump operator, you have probably noticed that both operators include a rule to create the action command (move or jump) on the output-link, and then remove that command when it has completed. You can simplify the writing of future operators by writing two general rules that perform these functions for every operator that performs external actions.

The first rule creates the action command on the output-link. To make this a completely general rule, you need to have a standard way of representing action commands on operators so that a single rule can create the commands on the output-link. The simplest convention is to have the action command be an augmentation of the operator, such as: ^jump.direction <direction>. Another possibility would be to have a specific augmentation on the operator, say ^actions, that had the commands as subobjects:

```
(<o> ^actions.jump.direction <direction1>)
```

You probably want to adopt this second approach because it makes it possible to write a single rule that copies the action commands to the output-link without risk of copying other augmentations. Under that convention, the general command creation rule is:

```
sp {apply*operator*create-action-command
  (state <s> ^operator.actions.<att> <value>
    ^io.output-link <ol>)
  -->
  (<ol> ^<att> <value>)}
```

The <att> variable will match jump and the value will match the identifier of the object that has direction as an augmentation. It copies ^jump and that identifier on to the output-link.

The second general rule removes a completed action command.

```
sp {apply*operator*remove-command
  (state <s> ^operator.actions
    ^io.output-link <ol>)
  (<ol> ^<att> <value>)
  (<value> ^status complete)
  -->
  (<ol> ^<att> <value> -)}
```

Included so action has  
o-support.

With these rules included in an Eater, both the move and the jump operator require only proposals (except when other actions are performed, such as remembering prior moves). To use both of these rules, you need to modify the proposal rules so that they create the actions structure. Below is an example of the jump operator proposal.

```
sp {propose*jump
  (state <s> ^io.input-link.my-location.<dir>.<dir>.content
    { <content> <> wall })
  -->
  (<s> ^operator <o> + =)
  (<o> ^name jump
    ^actions.jump.direction <dir>
    ^content <content>)}
```

Replaced ^direction  
<dir> with this.

Rules like this simplify the writing of Soar programs by creating conventions; however these conventions also restrict what can be written in a Soar program. For example, if this rule is used in a Soar program, then the status complete augmentation will always be removed and will not be available for other rules to test. For this domain, it doesn't seem like that will be a problem, and it might not be for others as well. It is just that before creating a rule that applies to all operators, you should be careful. Throughout this tutorial you will see additional rules that are used to simplify the writing of programs. We have tried to be careful in selecting these types of rules.

## 10.4 Jump Operator Selection

There are many strategies for selecting jump operators. One simple strategy is to prefer operators that jump into cells with bonusfood to operators that move into empty cells, while rejecting operators that jump into empty cells. We can generalize some of the rules for move to cover both jump and move.

```

sp {select*move*bonusfood-better-than-normalfood-empty-eater
  (state <s> ^operator <o1> +
    ^operator <o2> +)
  (<o1> ^name { << jump move >> <name> }
    ^content bonusfood)
  (<o2> ^name <name>
    ^content << normalfood empty eater >>)
-->
  (<s> ^operator <o1> > <o2>))}

sp {select*jump*bonusfood-better-than*move*empty
  (state <s> ^operator <o1> +
    ^operator <o2> +)
  (<o1> ^name jump
    ^content bonusfood)
  (<o2> ^name move
    ^content empty)
-->
  (<s> ^operator <o1> > <o2>))}

```

## 10.5 Jump and Move Operator Selection

In order to cover all of the cases of move and jump operators for all of the different cell contents, you would need to write lots of rules. Is there an easier way? One way is to translate the different names and contents into numbers that correspond to the number of points the eater will get. A rule could then compare the numbers and create better preferences for operators with higher numbers. You can translate the names and contents into numbers by writing one rule for each name and content pair; however, it is much easier to create a structure in working memory with the information and have a general rule match against it and do the translation. This is similar to the way opposite directions were computed.

The first rule to write is the one that creates the structure in working memory. It needs to include an object for each pair of name and content, and for each pair, the expected value to the eater.

```
sp {init*elaborate*name-content-value
  (state <s> ^type state)
-->
  (<s> ^name-content-value <c1> <c2> <c3> <c4>
                                <c5> <c6> <c7> <c8>)
  (<c1> ^name move ^content empty ^value 0)
  (<c2> ^name move ^content eater ^value 0)
  (<c3> ^name move ^content normalfood ^value 5)
  (<c4> ^name move ^content bonusfood ^value 10)
  (<c5> ^name jump ^content empty ^value -5)
  (<c6> ^name jump ^content eater ^value -5)
  (<c7> ^name jump ^content normalfood ^value 0)
  (<c8> ^name jump ^content bonusfood ^value 5)}
```

To compare two operators, you can write a rule that matches the operators and the name-content-value structure, but it is easier to understand the reasoning if you break it into two rules. The first rule matches each operator and the appropriate name-content-value and copies the value onto the operator.

```
sp {elaborate*operator*value
  (state <s> ^operator <o> +
    ^name-content-value <ccv>)
  (<o> ^name <name> ^content <content>)
  (<ccv> ^name <name> ^content <content> ^value <value>)
-->
  (<o> ^value <value>)}
```

Now you can write a rule that compares the values associated with each operator and then creates a better preference for the operators with higher values.

```
sp {select*compare*best*value
  (state <s> ^operator <o1> +
    ^operator <o2> +)
  (<o1> ^value <v>)
  (<o2> ^value <v>)
-->
  (<s> ^operator <o1> > <o2>)}
```

During a run, the first rule will fire for every proposed operator, and the second rule will fire for every pair of operators with different values. All of these firings, including the proposals of the operators, happen during the proposal phase.



## 11. Advanced Eaters

If you want to build more complex eaters, here are a few ideas. You will find it easier to build some of these after you complete the next part of the tutorial.

1. An eater that chases another eater.
2. An eater that tries to stay away from other eaters.
3. An eater that systematically searches for food when empty cells surround it.
4. An eater that looks ahead to see what move would be better based on what surrounds the cell it is going to move into.
5. An eater that takes advantage of the fact that bonus food comes in vertical lines, each 4 cells apart.

## 12. Top-state Structure

This page has a summary of the structure of the top-state for every eater. It is a useful reference when you build your own eaters and you will probably want to make a copy of it. All of the attributes of an object appear below it in outline form, with indentation used to signify sub-objects. Possible values are listed on the same line as the attribute.

```

^io
  ^input-link
    ^eater
      ^direction east/north/south/west
      ^name red/blue/yellow/green/purple/black
      ^score 0-1000
      ^x 1-15
      ^y 1-15
    ^my-location
      ^content bonusfood/normalfood/eater/empty/wall
      ^east
        ^content bonusfood/normalfood/eater/empty/wall
        ...
      ^north
        ^content bonusfood/normalfood/eater/empty/wall
        ...
      ^south
        ^content bonusfood/normalfood/eater/empty/wall
        ...
      ^west
        ^content bonusfood/normalfood/eater/empty/wall
        ...
    ^output-link
      ^move
        ^direction east/north/south/west
        ^status complete - created by Soar as feedback
      ^jump
        ^direction east/north/south/west
        ^status complete - created by Soar as feedback
^superstate nil
^type state

```