The Engineer’s Guide to Soar
Course 01: Soar Essentials

Project 03: Operator Basics

By Dr. Bryan Stearns, 2024

Problem

Our agent detects input elements, but it does not yet do anything with them.

Solution

Use **operators** to have the agent choose suppliers from its input.

- Use an operator **proposal rule** to propose an operator for each input supplier that the agent detects.
- Use an operator **preference rule** to have the agent choose them in a particular order.
- Use an operator **apply rule** to print a message for each selected supplier in the order it is selected.

Project Goal

We want our agent to write out each input supplier name in descending order of its `total-score`.

Its messages should be as follows:

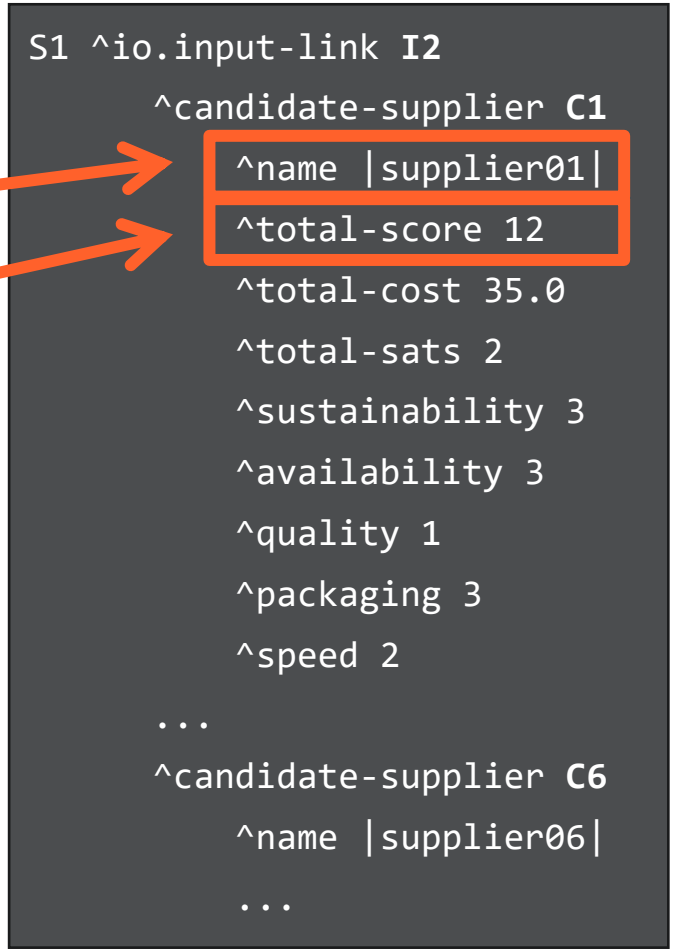
- `Selected supplier “supplier05” (total-score = 14)`
- `Selected supplier “supplier03” (total-score = 13)`
- `Selected supplier “supplier01” (total-score = 12)`
- `Selected supplier “supplier04” (total-score = 11)`
- `Selected supplier “supplier06” (total-score = 10)`
- `Selected supplier “supplier02” (total-score = 9)`

Accessing Supplier Input Properties

Our input includes many different attributes for each supplier.

In this project, we'll only use two:

- `^name STRING`
  - The name of the supplier
- `^total-score INT`
  - The sum of the non-total numeric inputs



Lesson 03 – Outline

This lesson explains the following new concepts:

1. Soar Operators
   - The Decision Cycle
   - Propose Rules
   - Preference Rules
   - Apply Rules
2. Stateful Actions
3. Negative Conditions

Starting with an overview of the main ideas

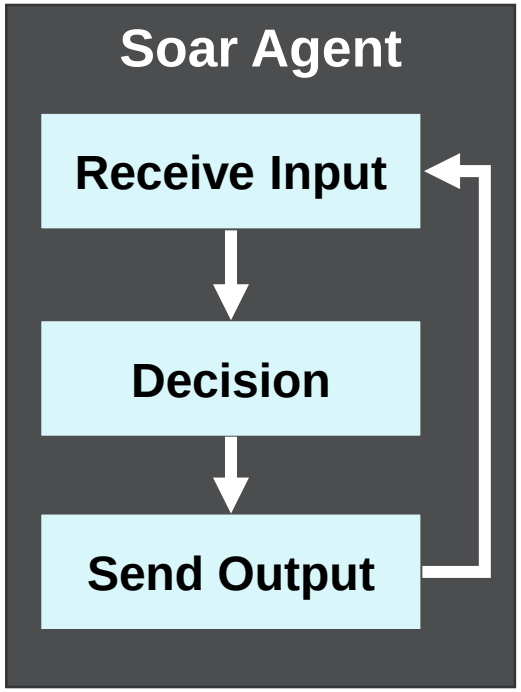
Soar Operators

The Core of Soar Decision Making

The Soar Decision Cycle

The **decision cycle** is the main processing loop of Soar processing

- It begins with updating *input-link* contents to match the environment
- Then agent rules update WM based on the current decision
- It ends with sending *output-link* contents to the environment



Operator Definition

An **operator** is a choice that the agent can select during a decision cycle

- It is a special WM structure with two forms:
  - **Proposed:** `(<s> ^operator <o> +)`
  - **Selected:** `(<s> ^operator <o>)`

The proposed operator:

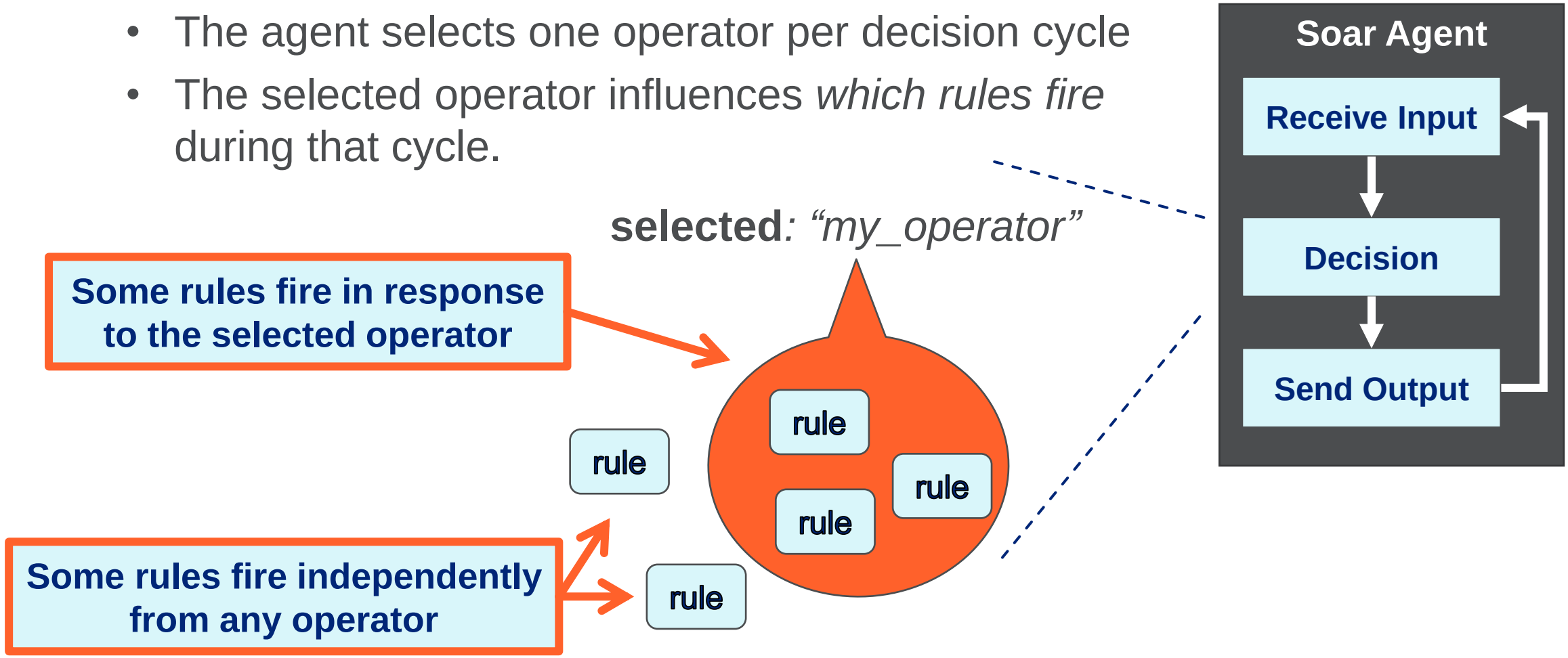
- Is a decision choice that the agent *could* select
- Is created by programmer-defined rules

The selected operator:

- Is a choice that the agent has currently selected
- Is created by Soar when the agent selects a proposed operator
- Is a copy of the proposed operator structure

Operator Effects

- The agent selects one operator per decision cycle
- The selected operator influences *which rules fire* during that cycle.



Soar Processing: Parallel and Serial Together

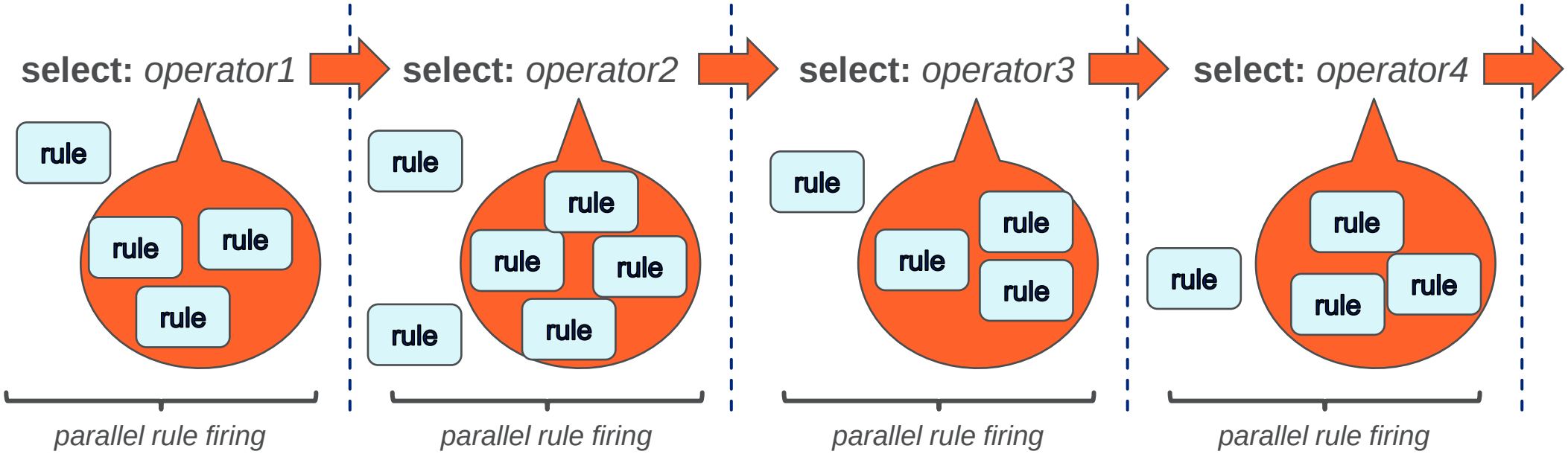
Satisfied Soar rules fire together at once whenever they can

- For effective parallel computation

But a Soar agent selects only a single operator per decision cycle

- For serial computation

Soar processing represents a reasoned sequence of parallel rule operations



Operator Code

Operators are defined in code via three types of rules:

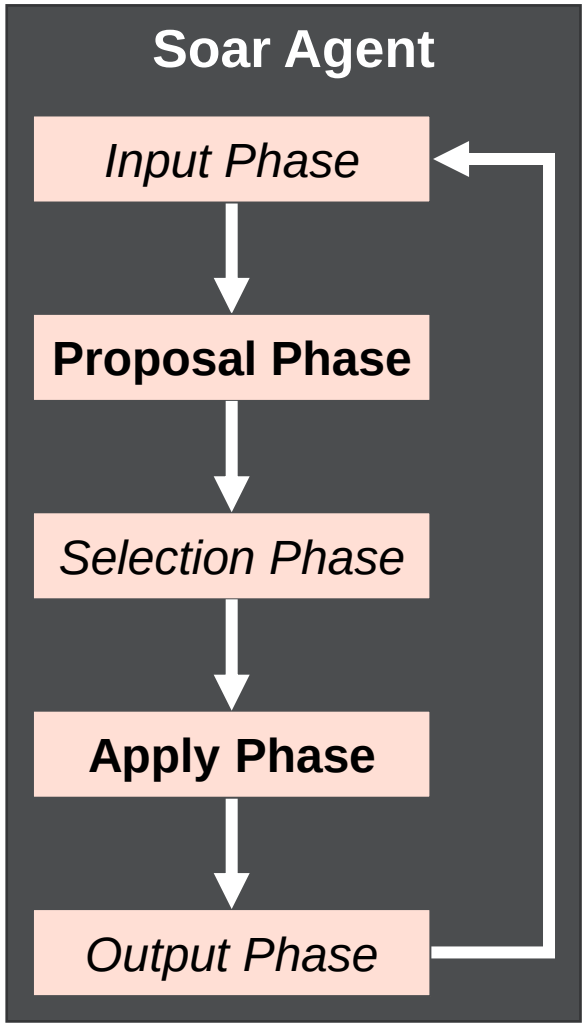
- **Proposal rules** propose operators to let the agent know what possible operators it can select from at any point in time
- **Preference rules** help the agent select from among the proposed operators
- **Apply rules** carry out the selected operator, once the agent selects one, by making changes to working memory

(We'll see later that a rule's type depends on what the rule does in its LHS and its RHS.)

Soar Decision Cycle – Phases

The decision cycle is in 5 main phases:

1. **Input Phase**
   - When input is updated
2. **Proposal Phase**
   - When rules propose and prefer operators
3. **Selection Phase**
   - When Soar selects an operator
4. **Apply Phase**
   - When rules apply the selected operator
5. **Output Phase**
   - When output is updated

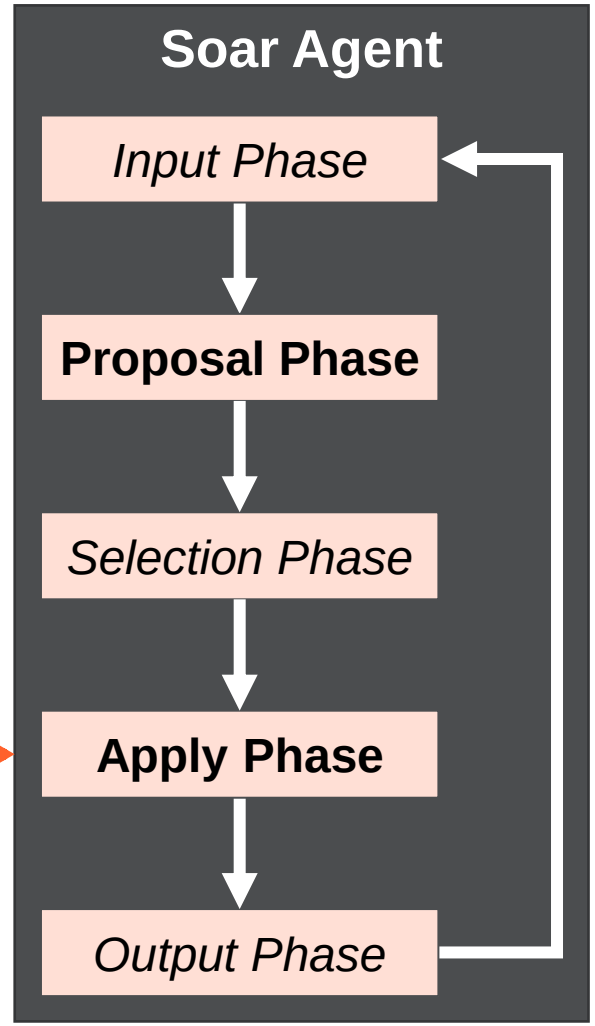


When Apply Rules Fire

Apply rules fire at different times from other rules

Non-apply rules can fire at any time

Apply rules only fire after Soar selects an operator



Soar Decision Cycle – Quiescence

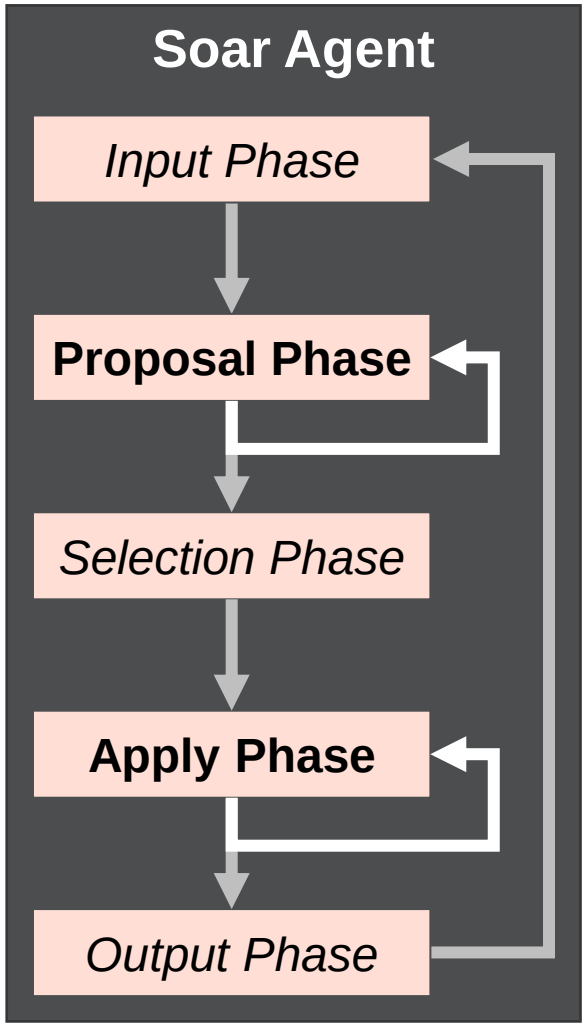
One rule's results might satisfy another rule.

- So there can be multiple cycles of cascading rule firings within a phase
- Each iteration is called an ***elaboration cycle***

The proposal & apply phases loop until *quiescence*

- **Quiescence**: When no more rules fire

(But Soar will automatically halt if it detects that rules have created an infinite loop.)

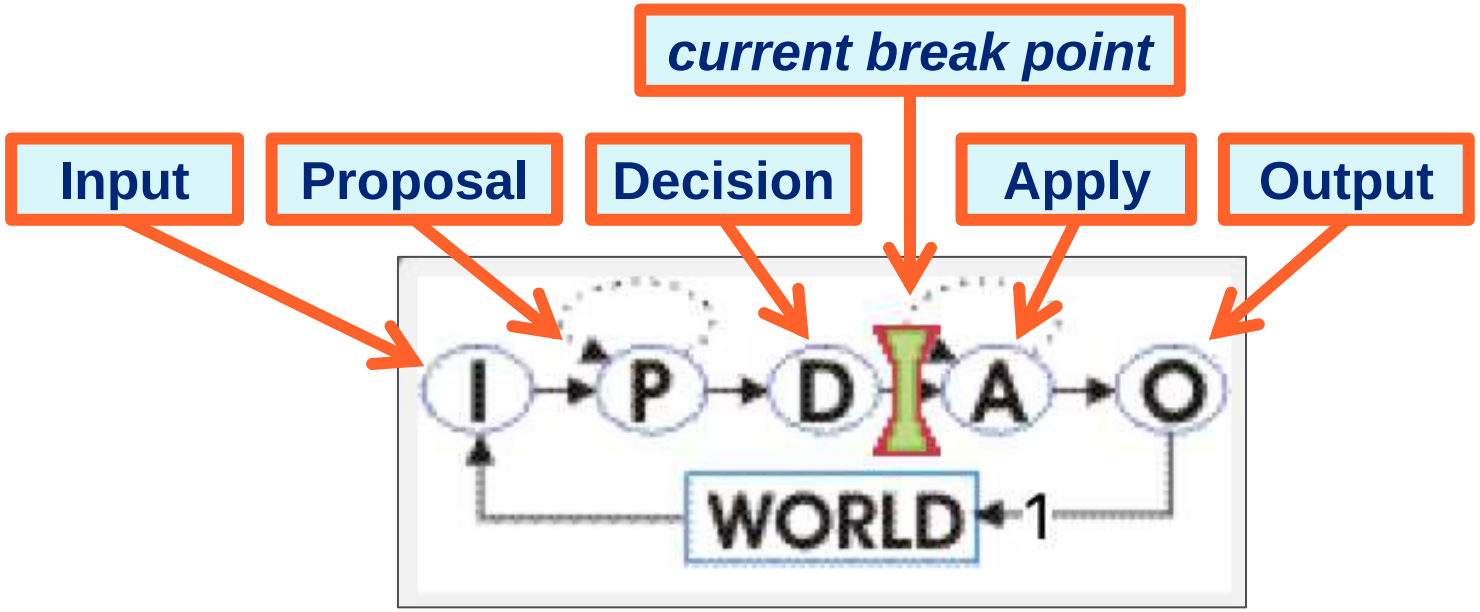


Decision Cycle Breakpoint in the Debugger

The Soar Debugger shows a small chart of the decision cycle for easy reference

You can also click on a link in this chart to control where Soar pauses when it finishes running a given number of decision cycle steps.

- By default, Soar will pause immediately after the *Decision Phase*, before any rules fire in the *Apply Phase*



Now let's put the ideas into practice!

Soar Operators

The Core of Soar Decision Making

Coding Our First Operator

We want our agent to:

1. Propose an operator for each input supplier
2. Prefer the operators in order of the supplier `total-score`
3. Apply each operator by writing out the selected supplier's name

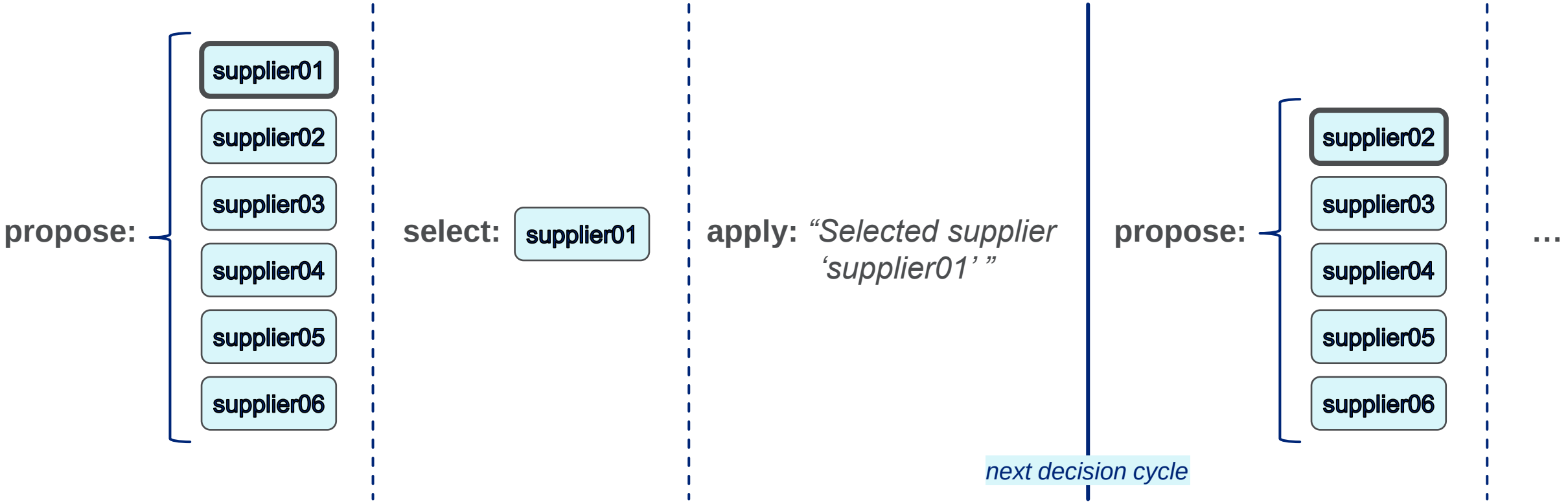
We will need:

- A single proposal rule
- A single preference rule
- A single apply rule

Project Solution Details

We use this sequence for each cycle until we've selected and processed each of the input suppliers:

1. Propose an operator for each input supplier
2. Prefer the operators in descending order of the supplier `total-score`
3. Apply each operator by writing out the selected supplier's name



Coding Proposal Rules

A proposal rule is any rule that creates an **`operator`** object as one of its actions

1. The attribute `^operator` is a reserved keyword in Soar
2. The attribute should point to a new ID (a new variable name that isn't on the LHS)
   - It is convention to use `<o>` to refer to operators.
3. The new ID should have a + after it
   - The + tells Soar to consider this operator as a choice during the selection phase
4. The new ID should ideally have a `( ^name STRING)` WME that describes the operator
   - It can have any number of other child WMEs too, as desired.

```
sp {propose*my-first-operator
    (state <s> ^type state)
    -->    1          2
    (<s> ^operator <o> +) 3
   4(<o> ^name |my-first-operator|)}
```

Let's Write Some Code!

1. Open your `agent_starter.soar` file and look at the first rule there.
2. Fill in the rule to match the solution shown below.
3. Run the agent again for 1 step and see what happens!
4. Proceed to the next slide when you're done.

```
sp {propose*suppliersort-main*select-supplier
    "Print any input priority weights"
    (state <s> ^io.input-link.candidate-supplier <sup>)
    -->
    (<s> ^operator <o> +)
    (<o> ^name select-supplier
         ^supplier <sup>)}
```

A Tie Impasse!

You should have seen something like the below output:

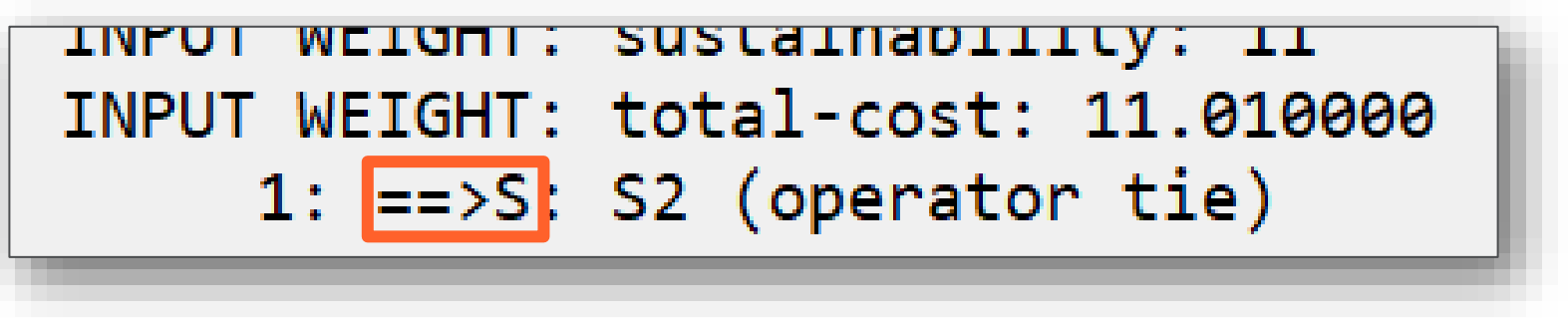
```
INPUT WEIGHT: sustainability: 11
INPUT WEIGHT: total-cost: 11.010000
     1: ==>S: S2 (operator tie)
```

What happened?

- Our single rule proposed multiple copies of the `select-supplier` operator
  - One for each input `candidate-supplier` structure
- The Soar agent did not know which operator to choose
- So the agent reported that it hit an "`operator tie`" impasse

What is an Impasse?

An impasse is indicated when you see the “==>” arrow in the agent trace.



An impasse occurs any time the agent is not able to continue its decision cycle.

- The text in parentheses after the arrow describes the type of impasse.
- We will see later that we can use impasses to design hierarchical reasoning

For now, if we see an impasse that means there is a bug in our code

- In this case, our agent is missing knowledge for how to choose an operator

Operator Preferences

Operator preferences

- Are special operator properties
  - Created by rules
- Guide the agent to automatically select a single operator from among proposed operators.

Examples:

- op1 > op2 (prefer op1 over op2 if both are proposed)
- op1 = op2 (choose randomly between op1 and op2 if tied between them)
- op2 < (tag op2 as a bad operator – only pick it if there's nothing better)
- op1 > (tag op1 as a good operator – pick it over others missing this tag)

Preference Types

Here is a complete list of the kinds of preferences we can give operators:

Preference Name	Syntax	Description
`Acceptable`	+	States that a value is a candidate for selection
`Reject`	–	Removes the value as a candidate for selection
`Better / Worse`	> value, < value	States, for the two values involved, that one should not be selected if the other is a candidate
`Best`	>	If a value is best, it will be selected over any other value that is not also best (or required)
`Worst`	<	States that the value should be selected only if there are no alternatives
`Unary Indifferent`	=	When two or more competing values both have unary indifferent preferences, choose one randomly
`Binary Indifferent`	= value	Like unary indifferent, but the operator is only made indifferent to the other operator value given
`Numeric-Indifferent`	= number	Like unary indifferent, but the number weights the chance of the operator's selection relative to others
`Require`	!	States that the value must be selected (preferred over all others)
`Prohibit`	~	States that the value cannot be selected

Coding Preference Rules

A preference rule is any rule that assigns an **operator preference** as one of its actions

1. To test for a *proposed* operator on the LHS, use a "+" after the operator ID condition
2. Assign a preference to that operator ID on the RHS
   - The example below makes any two proposed operators "indifferent" to each other
   - (Which means that Soar will select between them randomly if tied between them.)

```
sp {prefer*my-first-operator
    (state <s> ^operator <o1> +    1
               ^operator <o2> +)
    -->
    (<s> ^operator <o1> = <o2>)}
                          2
```

Using Operator Preferences

For our project, we need to use a *preference rule* to avoid the tie impasse!

We want rule logic along these lines:

```
“IF there are two proposed operators
    where the total-score of the second’s supplier
          is lower than that of the first,
THEN prefer the first over the second.”
```

That rule should fire for *every pair* of proposed operators to create a preference between each pair.

Let's Write Some Code!

1. Open your `agent_starter.soar` file and uncomment the second rule there.
2. Fill in the rule to match the solution shown below.
3. Proceed to the next slide for a description of how this rule works.
4. Run the agent again for *2 steps* and see what happens!

```
sp {prefer*suppliersort-main*select-supplier*total-score
    (state <s> ^operator <o1> +
               ^operator <o2> +)
    (<o1> ^name select-supplier
          ^supplier.total-score <sup1-score>)
    (<o2> ^name select-supplier
          ^supplier.total-score < <sup1-score>)
    -->
    (<s> ^operator <o1> > <o2>)}
```

The First Operator Selected!

When you run your agent now, you should see output like the following:

```
INPUT: Supplier candidate: supplier02
INPUT: Supplier candidate: supplier01
     1: O: O2 (select-supplier)
     2: ==>S: S2 (operator no-change)
```

What happened?

- Soar used our rule to select one of the 6 proposed operators in the 1st cycle

What Does this Preference Rule Do?

Note that "<" or ">" has a different meaning on the LHS vs the RHS.

1. It tests for two *proposed* operators
2. It tests that one has a lower supplier total-score than the other
3. It gives the one with the higher total-score ***better preference*** over the other

```
sp {prefer*suppliersort-main*select-supplier*total-score
    (state <s> ^operator <o1> +
               ^operator <o2> +)
    (<o1> ^name select-supplier
          ^supplier.total-score <sup1-score>)
    (<o2> ^name select-supplier
          ^supplier.total-score < <sup1-score>)
    -->
    (<s> ^operator <o1> > <o2>)}
```

1

2

3

This *compares* two values

This creates a *preference*

Condition Comparisons

Soar syntax supports many kinds of comparisons:

- (See page 53 in the manual for more details)

Comparison	Semantics
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
<>	Not equal
<=>	Same type (int, float, string, or ID)

- To test equality, you don't need a comparison symbol. Use the variable you want equality with.
  - Example:

```
(<s> ^fizz <same-value>
     ^buzz <same-value>)
```

Inspecting Preferences

The "`preferences <ts> operator`" command lets you see the agent's current preferences.

Use it after running for one step. You should see this. →

- Note: The ID numbers (e.g. "04") may vary each run.

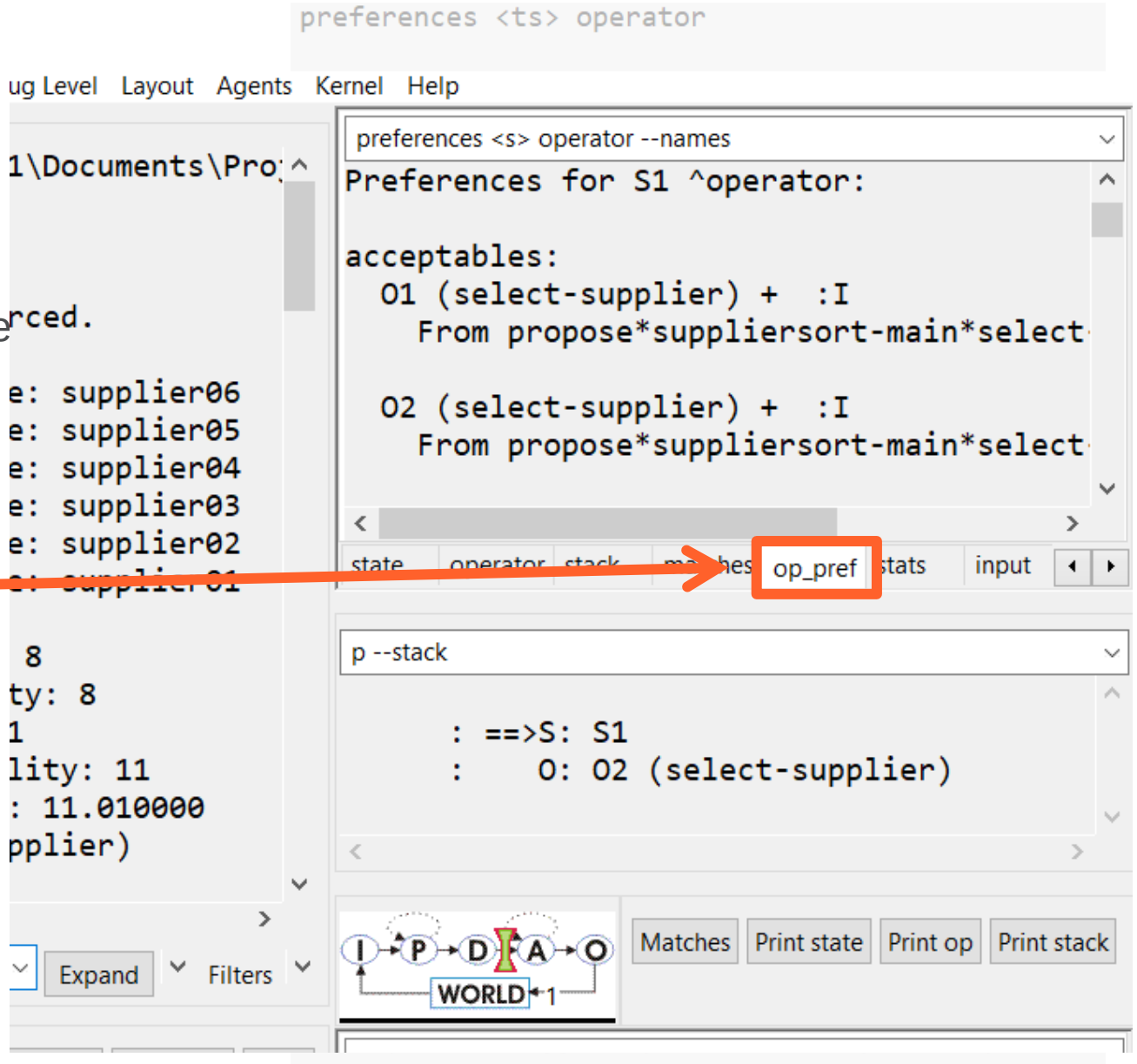
```
preferences <ts> operator

Preferences for S1 ^operator:

acceptables:
  O1 (select-supplier) +  :I
  O2 (select-supplier) +  :I
  O3 (select-supplier) +  :I
  O4 (select-supplier) +  :I
  O5 (select-supplier) +  :I
  O6 (select-supplier) +  :I

betters:
  O6 (select-supplier) > O1 (select-supplier) :I
  O4 (select-supplier) > O1 (select-supplier) :I
  O3 (select-supplier) > O1 (select-supplier) :I
  O2 (select-supplier) > O1 (select-supplier) :I
  O1 (select-supplier) > O5 (select-supplier) :I
  O2 (select-supplier) > O6 (select-supplier) :I
  O2 (select-supplier) > O5 (select-supplier) :I
  O2 (select-supplier) > O4 (select-supplier) :I
  O2 (select-supplier) > O3 (select-supplier) :I
  O6 (select-supplier) > O3 (select-supplier) :I
  O4 (select-supplier) > O3 (select-supplier) :I
  O3 (select-supplier) > O5 (select-supplier) :I
  O4 (select-supplier) > O6 (select-supplier) :I
  O4 (select-supplier) > O5 (select-supplier) :I
  O6 (select-supplier) > O5 (select-supplier) :I

selection probabilities:
  O2 (select-supplier) +  = 0. :I (100.00%)
```

Inspecting Preferences

The "`preferences <ts> operator`" command the agent's current preferences.

Use it after running for one step. You should see

This command is also provided in a Debugger auto-update tab.



Inspecting Preferences

The "`preferences <ts> operator`" command lets you see the agent's current preferences.

Use it after running for one step. You should see this. →

- The "01" … "06" labels are the IDs of the proposed operators
- You can see which operator is which by printing the ID
  - "`print O1 --depth 2`"
  - In this example, O1 is the operator for supplier06:

```
print O1 --depth 2
(O1 ^name select-supplier ^supplier C8)
  (C8 ^availability 3 ^name supplier06 ^packaging 3 ^quality 1 ^speed 1
      ^sustainability 2 ^total-cost 35.000000 ^total-sats 0 ^total-score 10)
```

```
preferences <ts> operator

Preferences for S1 ^operator:

acceptables:
  O1 (select-supplier) +  :I
  O2 (select-supplier) +  :I
  O3 (select-supplier) +  :I
  O4 (select-supplier) +  :I
  O5 (select-supplier) +  :I
  O6 (select-supplier) +  :I

betters:
  O6 (select-supplier) > O1 (select-supplier) :I
  O4 (select-supplier) > O1 (select-supplier) :I
  O3 (select-supplier) > O1 (select-supplier) :I
  O2 (select-supplier) > O1 (select-supplier) :I
  O1 (select-supplier) > O5 (select-supplier) :I
  O2 (select-supplier) > O6 (select-supplier) :I
  O2 (select-supplier) > O5 (select-supplier) :I
  O2 (select-supplier) > O4 (select-supplier) :I
  O2 (select-supplier) > O3 (select-supplier) :I
  O6 (select-supplier) > O3 (select-supplier) :I
                         pplier) > O3 (select-supplier) :I
                         pplier) > O5 (select-supplier) :I
                         pplier) > O6 (select-supplier) :I
                         pplier) > O5 (select-supplier) :I
                         pplier) > O5 (select-supplier) :I

selection probabilities:
  O2 (select-supplier) +  = 0. :I (100.00%)
```

Another Impasse…

There was another impasse after we ran the agent:

```
INPUT: Supplier candidate: supplier02
INPUT: Supplier candidate: supplier01
     1: O: O2 (select-supplier)
     2: ==>S: S2 (operator no-change)
```

What happened?

- The agent is still missing some knowledge
  - It has no apply rule for this operator!
- We need to tell our agent how to apply the `(select-supplier)` operator!

Coding Apply Rules

An apply rule is any rule that tests a selected **`operator`** in its LHS

1. The `^operator` ID is tested *without the +*.
2. You usually want to also test the operator's name so that your rule only applies that operator.
3. The RHS can include any desired actions.

```
sp {apply*my-first-operator
    (state <s> ^operator <o>) 1
    (<o> ^name |my-first-operator|) 2
    -->
    (<s> ^my-attribute |my-value|) 3
```

Let's Write Some Code!

1. Open your `agent_starter.soar` file and uncomment the third rule there.
   - `apply*suppliersort-main*select-supplier`
2. On your own: Fill in the blanks to test for a proposed `(select-supplier)` operator.
   - Collect the operator's supplier name to print it out from the RHS
3. Run your agent for 2 steps and see what happens!
   - You should see output like the following:

```
INPUT WEIGHT: sustainability: 11
INPUT WEIGHT: total-cost: 11.010000
     1: O: O2 (select-supplier)
Selected supplier "supplier05" (total-score = 14)
     2: ==>S: S2 (operator no-change)
```

4. Proceed to the next slide when you're done.
   - The impasse you see is not the one you saw before. (Your rule solved that one!)
   - This impasse is for a new reason that we'll address in the next section.

Inspecting Operators

The Soar CLI treats "<o>" as a keyword to refer to the current selected operator, if there is one.

- Run your agent from the beginning for only 1 step
- Then enter "`print <o> --depth 2`"
- You should see the following:

```
     1: O: O2 (select-supplier)


--> 1 decision cycle executed. 35 rules fired.
print <o> --depth 2
(O2 ^name select-supplier ^supplier C7)
  (C7 ^availability 3 ^name supplier05 ^packaging 3 ^quality 3 ^speed 2
         ^sustainability 3 ^total-cost 25.000000 ^total-sats 1 ^total-score 14)
```

Note: *This print example will only work if you have not yet triggered an impasse, for reasons that will become clear in Lesson 06.*

Stateful Actions & Negative Conditions

Finishing Our Project

Progressing Through Operators

Your operator code is almost complete:

- An operator is proposed for each supplier that could be selected.
- Operators for suppliers with lower names are preferred over those with higher names.
- When an operator (and its supplier) are selected, the supplier name is printed.

We want to select a different operator each cycle

- But right now, the same operator is still proposed after its apply rule takes effect.
- That's the `(operator no-change)` impasse: The operator isn't changing!

How do we get an operator to no longer be proposed after it has been applied?

- (So the next-best operator can be selected next)

We need our agent to *remember* once it has applied an operator

- (So that it knows it doesn't need to keep proposing it)

Stateful Actions

We will modify our apply rule to create a state variable

- Marking that the agent has processed a particular supplier

We will modify our proposal rule to only propose *unprocessed* suppliers

- The proposal will retract as soon as the apply rule has fired
  - Because its conditions will no be longer satisfied
- When a proposal rule retracts, the operator proposal is removed from WM
  - So the next-best operator can be selected during the next decision cycle.

Let's Write Some Code!

1. Open your `agent_starter.soar` file and modify the apply rule as follows:

```
sp {apply*suppliersort-main*select-supplier
    (state <s> ^operator <o>)
    (<o> ^name select-supplier
         ^supplier <sup>)
    (<sup> ^name <sup-name>
          ^total-score <sup-score>)
    -->
    (write |Selected supplier "| <sup-name> |" (total-score = | <sup-score> |)| (crlf))
    (<s> ^selected <sup>)}
```

Mark that the supplier has been selected and processed

2. Modify your proposal rule as follows:

```
sp {propose*suppliersort-main*select-supplier
    (state <s> ^io.input-link.candidate-supplier <sup>
               -^selected <sup>)
    -->
    (<s> ^operator <o> +)
    (<o> ^name select-supplier
         ^supplier <sup>)}
```

Only propose suppliers that have not yet been selected.

(Notice the "–" at the front!)

3. Proceed to the next slide when you're done.

Negative Conditions

To test that the described WME pattern does *not* exist in WM, just add a "–" in front.

You can add the "–" in multiple ways:

1. • In front of a condition block (to negate the entire block)
2. • Or just in front of an attribute (to negate just that attribute pattern)

The following pieces of code both demonstrate valid negations:

1

```
"Not (first & second)"
-(<x> ^first |value|
      ^second |value|)
```

Satisfied if either WME is missing

2

```
"(Not first) & (Not second)"
(<x> -^first |value|
     -^second |value|)
```

Satisfied if both WMEs are missing

Syntax for Negative Conditions

Soar will give a syntax error if you put a negative condition *as the first condition* of a rule.
If needed, you can always add "`^type state`" as a filler starting condition.

This gives an error:

```
sp {bad*negative-condition
    (state <s> -^selected <any>)
    ...
```

This is fine:

```
sp {good*negative-condition
    (state <s> ^type |state|
               -^selected <any>)
    ...
```

Soar allows you to omit the variable after an attribute test if the value doesn't matter.

- You only need to include a value in a negation test if you want to test that a *specific value* is not there.

This is fine:

```
sp {good*negative-condition*shorthand
    (state <s> ^type |state|
               -^selected )
    ...
```

Rule Effect Persistence

When our **proposal** rule retracted, the operator proposal was removed from WM.

But when our **apply** rule retracted, the `^selected <sup>` WME stayed in WM.

- *Why the difference?*

For normal Soar rules:

- The action effects are always undone as soon as the rule instance retracts

Apply rules are special:

- **Apply rule actions are permanent**
- (Until some other rule makes further changes)

i-support vs o-support

More formally:

If an i-supported rule removed a WME, the WME will *reappear* once the instantiation retracts!

Non-apply rules are said to have "**`i-support`**"

- This stands for "instantiation support"
- That is, their RHS results are supported in WM by the rule instantiation that created them.
- Once the instantiation retracts (its LHS no longer matches), the results lose support and are undone.

Apply rules are said to have "**`o-support`**"

- This stands for "operator support"
- That is, their RHS results are supported in WM by the operator decision.
- Even when the instantiation retracts, the results keep their support.

Final Results

After running your agent for 7 steps, you should see the following:

- Each input supplier is selected in alphanumeric order!

If so, **congratulations**!
You have completed Project 03!

```
INPUT WEIGHT: sustainability: 11
INPUT WEIGHT: total-cost: 11.010000
     1: O: O2 (select-supplier)
Selected supplier "supplier05" (total-score = 14)
     2: O: O4 (select-supplier)
Selected supplier "supplier03" (total-score = 13)
     3: O: O6 (select-supplier)
Selected supplier "supplier01" (total-score = 12)
     4: O: O3 (select-supplier)
Selected supplier "supplier04" (total-score = 11)
     5: O: O1 (select-supplier)
Selected supplier "supplier06" (total-score = 10)
     6: O: O5 (select-supplier)
Selected supplier "supplier02" (total-score = 9)
     7: ==>S: S2 (state no-change)
```

Extra: Operator No-Change Impasses

When we were missing an apply rule, we got an `(operator no-change)` impasse

When we had an apply rule but the proposal didn't retract, we still got the same impasse

- *Was this the same impasse?* Not quite!

In Soar you can get an `(operator no-change)` impasse for two reasons:

1. Your operator isn't applied (no matching apply rules)
2. Your operator was applied but the apply rules didn't resolve the proposal (because the propose rule didn't retract)

If you see this impasse from your code where it shouldn't be, check for both causes!

Extra: Types of Impasses

Here is a reference table of the types of impasses and their meanings:

Impasse Name	Cause	Description
`(state no-change)`	When the proposal phase runs to quiescence and no operators are proposed	The agent doesn't know what it could try to do next. (Such as when there are no rules, or when the task is done.)
`(operator no-change)`	When the proposal phase runs to quiescence and no new operator has been selected	The agent isn't making progress toward its goal.
`(operator tie)`	When there is a collection of equally eligible operators competing for selection	The agent doesn't know how to choose among two or more operators.
`(operator conflict)`	When two or more objects are better than each other and not dominated by a third operator	The agent has conflicting preferences among proposed operators.
`(constraint-failure)`	When there are conflicting *necessity* preferences	The agent has been told that more than one operator is mandatory to select this cycle.