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

Project 09: Combining Preferences

By Dr. Bryan Stearns, 2024

Problem

- Our agent is supposed to filter input suppliers before sorting by weighted scores.
- Our agent is supposed to limit the number of recommendations it gives.
- Our agent can only handle a single round in input/output.

Solution

- Use preferences to add the needed behavior.
- Learn how to reset our agent after it sends output.
- Finish our Supplier Sort agent code!

Lesson 07 Review: Sorting Steps

Recall from Lesson 07 the following goal for our agent's sorting:

1. **Exclude suppliers where total-sats == 0**
2. **Sort suppliers by total-sats in descending order**
3. Iteratively sort suppliers by attributes in priority weight order
4. Sort remaining tied suppliers by total-score
5. **Sort remaining tied suppliers randomly**

We've implemented steps 3-4.

We implement the rest now!

```
S1 ^io.input-link I2
     ^priorities P1
       ^total-cost 11.01
       ^sustainability 11
       ^quality 10
       ^availability 8
       ^packaging 8
       ^speed 7

     ^candidate-supplier C1
       ^name |supplier01|
       ^total-score 12
       ^total-sats 2
       ^total-cost 35.0
       ^sustainability 3
       ^availability 3
       ^quality 1
       ^packaging 3
       ^speed 2
     ...
```

Project Goals

1. Combine (`select-supplier`) preferences to filter and pre-sort suppliers before sorting them with (`evaluate-weight`):
   - Reject any suppliers that have a `total-sats` value of 0.
   - Prefer suppliers with a higher `total-sats` value.
   - Limit the output list size based on an input setting.
2. Be able to process new input after giving output
   - Without killing and respawning the agent

Deployable File Names

In the past, we've named files that we've imported from previous projects with the prefix:

- "`existing_code_X_...`"

And files that we edited in that lesson were named:

- "`agent_starter_...`"
- "`instructor_solution_...`"

Obviously, this naming scheme is not what you want to see in a real-world agent.

In this lesson, we wrap up our agent code, so we use a more deployable naming scheme:

- All your "`agent_starter_...`" files are simply named for their state.
- (Not to worry: There are still "`instructor_solution_...`" files too for reference.)

By the end of this lesson, all your agent files will look production-ready!

Lesson 09 – Outline

This lesson explains the following new concepts:

- Using the "reject" (`-`) preference
- Using the "`soar init`" command

And we finish our agent code!

STEP 1: Pre-Sort by `total-sats`

To Meet Client Constraints

What Does `total-sats` Mean?

Each input supplier has the `^total-sats` augmentation

- This metric is independent from other augmentations.

It represents the count of data elements that the client needs that can be supplied by this supplier.

Example: The client needs to choose a set of suppliers from which to purchase the following two pieces of data:

1. The member’s colorectal screening status
2. The member’s BMI

If a supplier cannot provide either of these, its `^total-sats` is 0.

If a supplier has one of these but not the other, `^total-sats` is 1.

And so on.

```
S1 ^io.input-link I2
     ^priorities P1
       ^total-cost 11.01
       ^sustainability 11
       ^quality 10
       ^availability 8
       ^packaging 8
       ^speed 7

     ^candidate-supplier C1
       ^name |supplier01|
       ^total-score 12
       ^total-sats 2
       ^total-cost 35.0
       ^sustainability 3
       ^availability 3
       ^quality 1
       ^packaging 3
       ^speed 2
     ...
```

Pre-Sort: Combining Preferences

We want to use `^total-sats` to pre-sort the suppliers:

- Prefer suppliers that provide more required data over others.

If any ties remain after sorting by `^total-sats`, then break the tie with our tie substate.

If desired, we could combine any number of preference rules on the topstate to create relative orderings among suppliers in parallel.

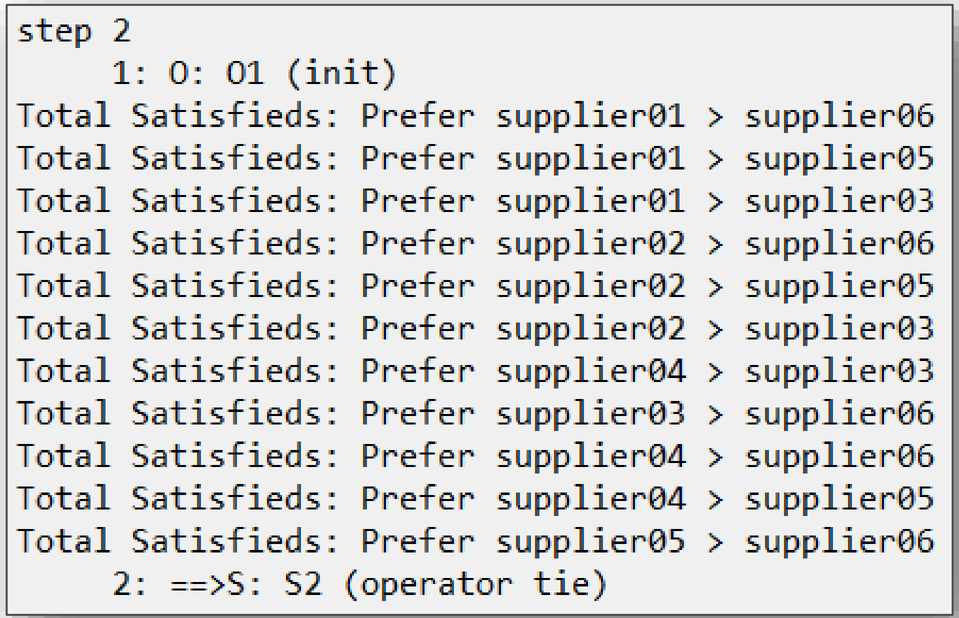
But we would need to be careful to avoid creating conflicting preferences

Example:

- Prefer A > B because A.fizz > B.fizz
- Prefer B > A because B.buzz > A.buzz

STEP 1: Let’s Write Some Code!

1. Open `suppliersort_main.soar`.
2. Find the comment instructions for STEP 1.
3. Write a preference rule that that prefers one (`select-supplier`) operator over another if it has a higher `^total-sats` value.
   - Recall that to test for a *proposed* operator, test for: `(state <s> ^operator <o1> + )`
   - `total-sats` is then under the operator: `(<o1> ^supplier.total-sats <y1>)`
   - Don’t forget to add a (`write …`) statement so you can quickly see what preferences your rule makes.
4. Test your code by running for at least 2 steps.
   - You should see the following preferences created →



```
step 2
     1: O: O1 (init)
Total Satisfieds: Prefer supplier01 > supplier06
Total Satisfieds: Prefer supplier01 > supplier05
Total Satisfieds: Prefer supplier01 > supplier03
Total Satisfieds: Prefer supplier02 > supplier06
Total Satisfieds: Prefer supplier02 > supplier05
Total Satisfieds: Prefer supplier02 > supplier03
Total Satisfieds: Prefer supplier04 > supplier03
Total Satisfieds: Prefer supplier03 > supplier06
Total Satisfieds: Prefer supplier04 > supplier06
Total Satisfieds: Prefer supplier04 > supplier05
Total Satisfieds: Prefer supplier05 > supplier06
     2: ==>S: S2 (operator tie)
```

STEP 2: Using the Reject Preference

Filtering Out Invalid Suppliers

The Reject Preference

The **`Reject` (“–”)** preference is one of the unary preferences we saw in Lesson 03.

- A Soar agent will never select an operator that has the `Reject` preference,
- Even if it has other preferences like Best (“>”) or Require (“!”).

```
sp {prefer*my-operator*reject*completed
    (state <s> ^operator <o1> +
               ^work-completed <w>)
    (<o1> ^work <w>)
    -->
    (<s> ^operator <o1> - )}
```

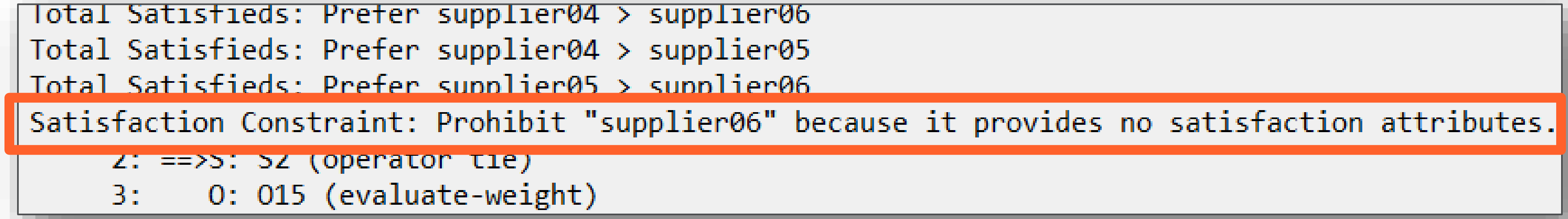
The difference between `Reject` (“–”) and `Prohibit` (“~”) has to do with Soar’s “Chunking” learning algorithm, which is beyond the scope of this course.

STEP 2: Reject Suppliers With 0 `total-sats`

If a supplier does not satisfy any client's needs, the agent should not recommend it.

- Rejecting its (`select-supplier`) operator will prevent it from being added to the output list!

1. Find the comment instructions in `suppliersort_main.soar` for STEP 2.
2. Add a preference rule there that rejects any (`select-supplier`) operator that proposes a supplier that has a `total-sats` count of 0.
   - Don't forget to add a (`write …`) statement so you can quickly see what preferences your rule makes.
3. Test your rule by running your agent to completion.
   - You should see "supplier06" rejected now.
   - The agent's output list should not include supplier06 anymore.



```
Total Satisfieds: Prefer supplier04 > supplier06
Total Satisfieds: Prefer supplier04 > supplier05
Total Satisfieds: Prefer supplier05 > supplier06
Satisfaction Constraint: Prohibit "supplier06" because it provides no satisfaction attributes.
     2: ==>S: S2 (operator tie)
     3:    O: O15 (evaluate-weight)
```

STEP 3: Limit List Size

Another Use for Reject Preferences

Responding to List Length

Agent input includes a structure named “`^settings`”.

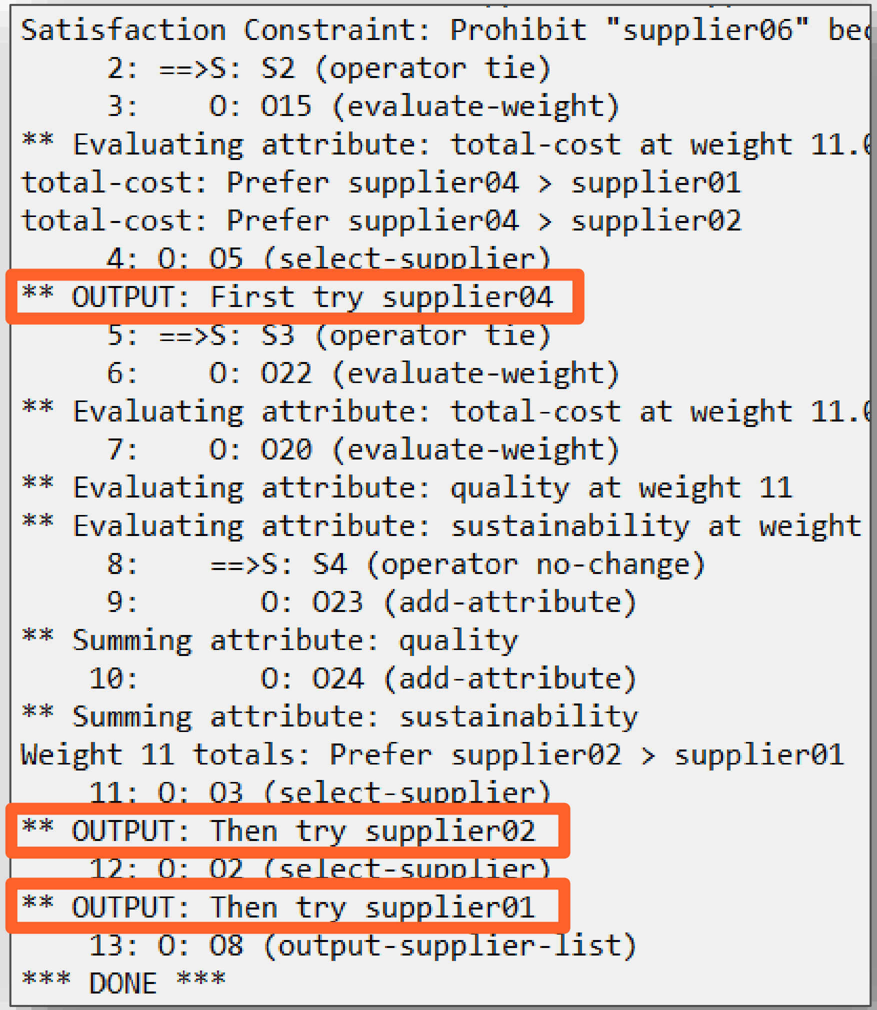
- It has one augmentation, `^max-output-suppliers`, which sets the *maximum* number of suppliers that should be in the agent’s output recommendation list.
- The list can be smaller, but it should not be larger than this count.

```
S1 ^io.input-link I2
     ^settings I4
       ^max-output-suppliers 3
     ...
```

We can add a single `Reject` preference rule to implement this behavior!

STEP 3: Reject (`select-supplier`) Once the Max is Met

1. Find the comment instructions in `suppliersort_main.soar` for STEP 3.
2. Add a preference rule there that:
   - Tests if the current size of `^supplier-list` is *greater or equal* to `max-output-suppliers`.
   - Rejects any (`select-supplier`) operator if so.
3. Test your rule by running your agent to completion.
   - The agent should finish after about 13 steps. →
   - The agent's output list should only include 3 suppliers.



```
Satisfaction Constraint: Prohibit "supplier06" be
     2: ==>S: S2 (operator tie)
     3:    O: O15 (evaluate-weight)
** Evaluating attribute: total-cost at weight 11.
total-cost: Prefer supplier04 > supplier01
total-cost: Prefer supplier04 > supplier02
     4: O: O5 (select-supplier)
** OUTPUT: First try supplier04
     5: ==>S: S3 (operator tie)
     6:    O: O22 (evaluate-weight)
** Evaluating attribute: total-cost at weight 11.
     7:    O: O20 (evaluate-weight)
** Evaluating attribute: quality at weight 11
** Evaluating attribute: sustainability at weight
     8:    ==>S: S4 (operator no-change)
     9:       O: O23 (add-attribute)
** Summing attribute: quality
    10:       O: O24 (add-attribute)
** Summing attribute: sustainability
Weight 11 totals: Prefer supplier02 > supplier01
    11: O: O3 (select-supplier)
** OUTPUT: Then try supplier02
    12: O: O2 (select-supplier)
** OUTPUT: Then try supplier01
    13: O: O8 (output-supplier-list)
*** DONE ***
```

STEP 4: Break True Ties Randomly

Using Indifferent Preferences

Lesson 07 Review: Sorting Steps

Recall from Lesson 07 the following 5-step goal for our agent’s logic:

1. Exclude suppliers where total-sats == 0
2. Sort suppliers by total-sats in descending order
3. Iteratively sort suppliers by attributes in priority weight order
4. Sort remaining tied suppliers by total-score
5. **Sort remaining tied suppliers randomly**

We’ve implemented the first 4 steps.

Only 1 step left to add!

```
S1 ^io.input-link I2
     ^priorities P1
       ^total-cost 11.01
       ^sustainability 11
       ^quality 10
       ^availability 8
       ^packaging 8
       ^speed 7

     ^candidate-supplier C1
       ^name |supplier01|
       ^total-score 12
       ^total-sats 2
       ^total-cost 35.0
       ^sustainability 3
       ^availability 3
       ^quality 1
       ^packaging 3
       ^speed 2
     ...
```

Where To Code Final Sorting Step?

We can describe our agent as having 3 stages of sorting:

1. Preferences that fire before (`evaluate-weight`)
2. Preferences generated by (`evaluate-weight`)
3. Preferences that fire after (`evaluate-weight`), using (`break-attr-tie`)

We want to sort suppliers randomly if they are still tied after these 3 stages.
Can we do this without adding a new operator?

- Yes!

We can add this final step as a *second* apply rule of (`break-attr-tie`).

- Recall: The existing apply rule sorts suppliers by `total-score`.
- The new rule should only sort suppliers randomly if their `total-scores` are the same!

Using Different Test Input

The test input we’ve used so far will not test our desired behavior.

- We need an alternate test case!

Open your Project 09 folder and look at “`fake_agent_input_2.soar`”.

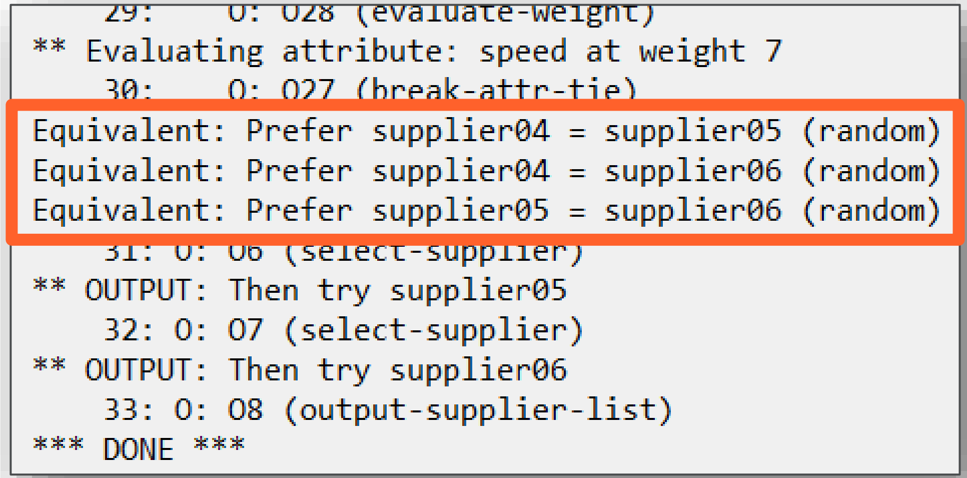
- The suppliers have different values than we’ve used so far.
- The `max-output-suppliers` setting is set to 5 instead of 3.

Figure out how the agent ought to sort these suppliers and why:

- First Supplier01 (why?)
- Then Supplier02 (why?)
- Then Supplier03 (why?)
- Then two of Suppliers 04, 05, or 06 in random order.

STEP 4: Let’s Write Some Code!

1. Open `_firstload_suppliersort.soar` and uncomment the line that loads `fake_agent_input_2.soar`.
   - (You can comment out the previous line that loads the other input.)
2. Now find the comments in `suppliersort_tie.soar` labeled “STEP 4.”
3. Write a new apply rule there for `break-attr-tie` that creates a binary indifferent preference between any two tied suppliers that have the same `total-score`.
   - Hint: You can copy most of the rule code from the other apply rule, (`apply*suppliersort-tie*break-attr-tie*prefer-total-score`)
   - Include an appropriate (`write`) statement that shows which pairs of suppliers are given binary indifference.
4. Test your rule by running your agent to completion.
   - You should see suppliers 4, 5, and 6 given binary indifference relative to each other. →



```
    29:    O: O28 (evaluate-weight)
** Evaluating attribute: speed at weight 7
    30:    O: O27 (break-attr-tie)
Equivalent: Prefer supplier04 = supplier05 (random)
Equivalent: Prefer supplier04 = supplier06 (random)
Equivalent: Prefer supplier05 = supplier06 (random)
    31: O: O6 (select-supplier)
** OUTPUT: Then try supplier05
    32: O: O7 (select-supplier)
** OUTPUT: Then try supplier06
    33: O: O8 (output-supplier-list)
*** DONE ***
```

The `soar init` Command

So One Agent Can Process Many Jobs

Multi-Job Agents

Reminder: Use the "Source" button in the Java Debugger to quickly load a .soar file to your agent.

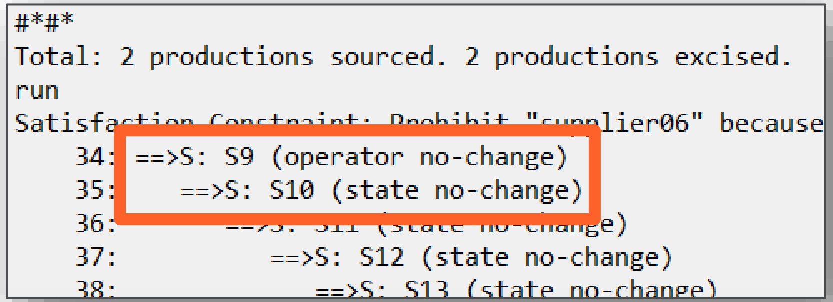
After our agent sends output, what if we want it to then process different inputs?

Test it:

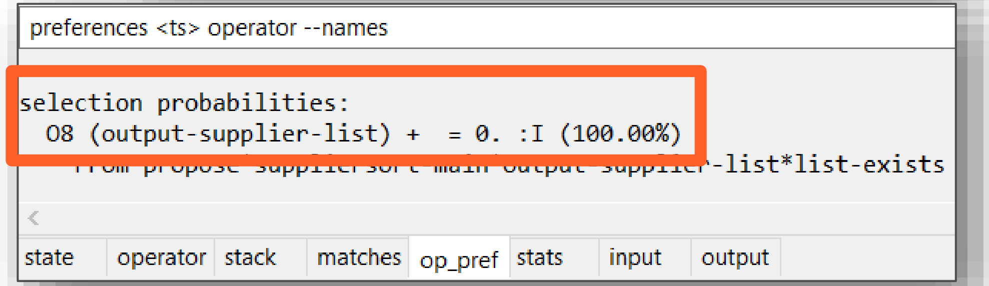
- Run your agent to completion using `fake_agent_input_2.soar` as an initial set of inputs.
- Then manually load `fake_agent_input.soar` into your agent as a *new* set of inputs.
    - The `fake_agent_input_...` rules use the same rule names, so one file will overwrite the other.
- Run your agent from its current state. What happens?
    - The agent enters a *state no-change* loop.

The problem: Our `supplier-list` object is still there.

- So our agent still tries to apply `(output-supplier-list)`
- This causes an *operator no-change* impasse, followed by the *state no-change* impasses.



```
#*#*
Total: 2 productions sourced. 2 productions excised.
run
Satisfaction Constraint: Prohibit "supplier06" because
    34: ==>S: S9 (operator no-change)
    35:    ==>S: S10 (state no-change)
    36:       ==>S: S11 (state no-change)
    37:          ==>S: S12 (state no-change)
    38:             ==>S: S13 (state no-change)
```



```
preferences <ts> operator --names

selection probabilities:
  O8 (output-supplier-list) +  = 0. :I (100.00%)
    From propose-supplier-sort-main-output-supplier-list*list-exists
```

Attempt: Remove Old Output Manually

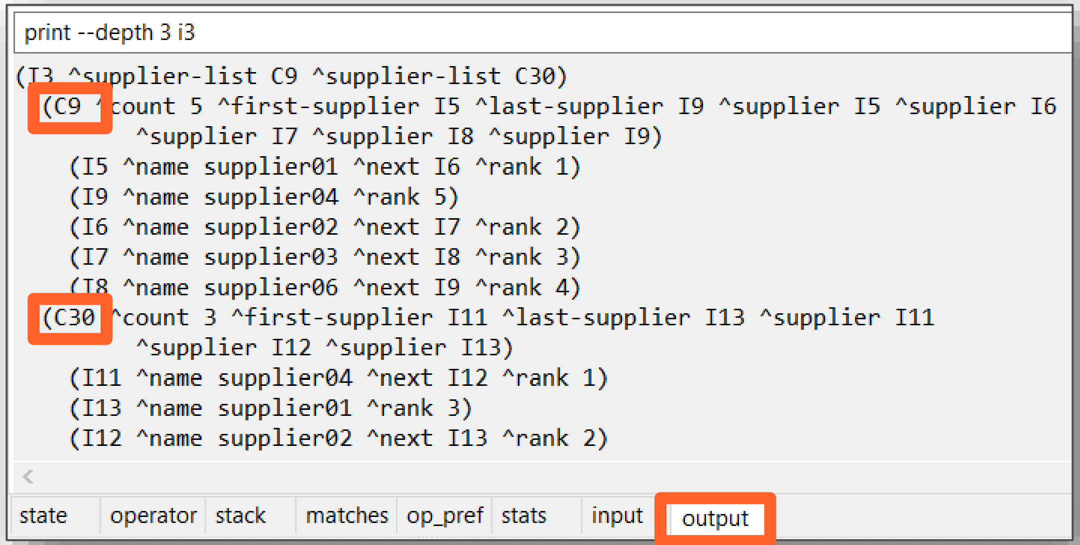
We can remove the `supplier-list` object from the state at the same time as we send it to output.

- Find the comment labeled “STEP 5” and add a line there that removes supplier-list.

Test it:

- Run your agent to completion using `fake_agent_input_2.soar` as an initial set of inputs.
- Then manually load `fake_agent_input.soar` into your agent as a *new* set of inputs.
- Run your agent again from its current state. What happens?
  - The agent seems to run okay again…
  - But check the output:
    - – The old output is still being sent!
    - – In addition to the new output!

The problem: o-supported WMEs (like the output list) don’t go away on their own!



```
print --depth 3 i3
```

```
(I3 ^supplier-list C9 ^supplier-list C30)
  (C9 ^count 5 ^first-supplier I5 ^last-supplier I9 ^supplier I5 ^supplier I6
       ^supplier I7 ^supplier I8 ^supplier I9)
    (I5 ^name supplier01 ^next I6 ^rank 1)
    (I9 ^name supplier04 ^rank 5)
    (I6 ^name supplier02 ^next I7 ^rank 2)
    (I7 ^name supplier03 ^next I8 ^rank 3)
    (I8 ^name supplier06 ^next I9 ^rank 4)
  (C30 ^count 3 ^first-supplier I11 ^last-supplier I13 ^supplier I11
       ^supplier I12 ^supplier I13)
    (I11 ^name supplier04 ^next I12 ^rank 1)
    (I13 ^name supplier01 ^rank 3)
    (I12 ^name supplier02 ^next I13 ^rank 2)
```

The `soar init` Command

Soar provides an easy way to reset the agent's memory

- So that it can be ready to receive a new task

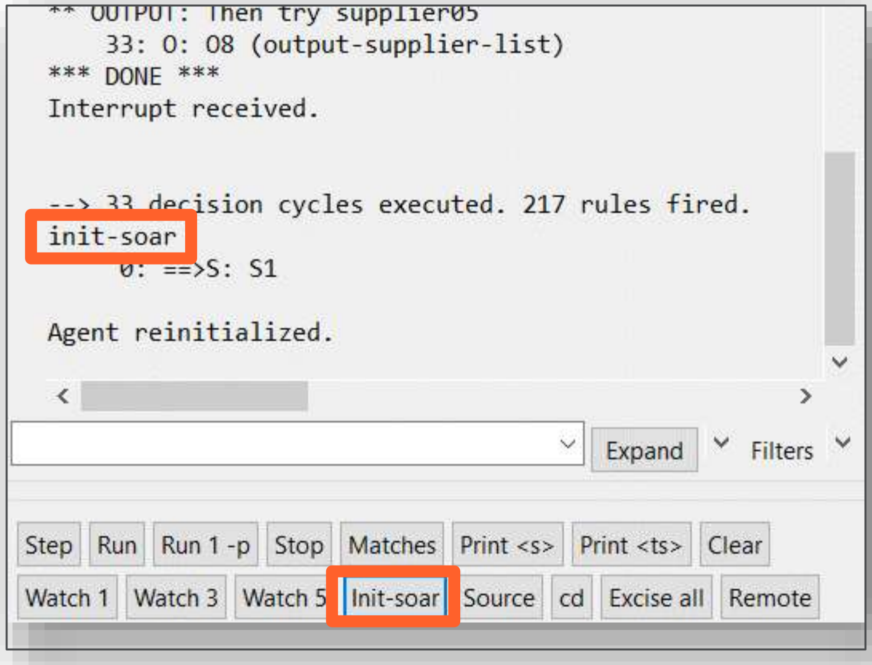
The "`soar init`" CLI command completely resets an agent's state while preserving its long-term memories (i.e. rules).

- "`init-soar`" or simply "`init`" also work as aliases.
- The Java Debugger also provides an "`Init-soar`" button that does the same thing.

External code environments are able to send Soar the `init` command before they send new inputs.

Try it:

- Repeat the same steps as on the previous slide, but this time re-initialize the agent between runs.
- Your agent should run properly with the new input!



```
** OUTPUT: Then try supplier05
    33: O: O8 (output-supplier-list)
*** DONE ***
Interrupt received.


--> 33 decision cycles executed. 217 rules fired.
init-soar
     0: ==>S: S1

Agent reinitialized.
```

Extra Credit: Alternate Solution

Instead of using `init`, you could *manually remove* the output as well…

- That is: remove the `supplier-list` WME from `output-link` once there is new input

On your own, experiment with different ways of accomplishing this.

- HINT: Is there an existing operator you could use for this?
- (By adding a new apply rule to the existing operator?)

Wrapping Up

The Agent Itself, Anyway

Final Checks

Think about edge cases. Does your agent still work:

1. If no suppliers are valid (all have `total-sats` = 0)?
   - The file `fake_agent_input_3.soar` is provided for you to test this case.
2. If `max-output-suppliers` is 0?
3. If the input changes part-way through the agent's processing?

Extra Credit: Add rules to catch bad input:

1. Candidate suppliers that are missing required fields like `total-cost`
2. No candidate suppliers at all
3. No priorities for different attributes
4. A cost and non-cost attribute sharing the same priority weight

And so on.

A good engineer pursues *quality.*

Congratulations!

You have completed Lesson 09!

- And you have completed all required Soar rules for your agent!

One lesson remains:

- Complete the next lesson to learn how to run a Soar agent from *app code*, with proper I/O
- (You won’t need to edit any .soar files for that lesson.)