

**The Engineer's Guide to Soar**  
**Course 01: Soar Essentials**

# **Project 08:**

# **Code**

# **Organization**

By Dr. Bryan Stearns, 2024



## Problem

- Our project code is getting fairly large!
- Our agent does not yet handle the case where two weights have the same value.

## Solution

- Adopt better practices for organizing and maintaining a large Soar project!
- When attributes share a weight, prefer by the subtotal of their scores.
  - Let a shared weight trigger an impasse and additional substate.
  - Calculate and return the subtotal from that substate so that regular processing can continue as normal.

## Project Goal

When (evaluate-weight) processes a weight with  $> 1$  attribute:

- A new apply rule sorts suppliers by their `^subtotal` attribute
  - This attribute doesn't exist at first.
  - A new substate calculates and attaches this attribute.
  - Then the (evaluate-weight) apply rule can fire.

## Lesson 08 – Outline

This lesson explains the following new concepts:

- Organizing and documenting larger agents
- Augmenting operator structures
- Error catching
- The cmd RHS Function
- Parallel summation
- Negated conjunctions

# Organizing and Documenting Larger Agents

When One File Isn't Enough

## Outgrowing a Single File

Our agent code is getting too big to navigate in just one or two files. What are best practices for managing larger Soar agents?

### **1. Separate files by their contents**

- So that related rules are in the same spot

### **2. Organize rules in each file by a consistent pattern**

- So it's easy to find a rule you want

### **3. Write documentation comments in each file**

- So it's easy to understand how the rules work

# 1 Separate Files by Contents

When organizing rules in .soar files, the following convention is suggested:

1. Have a separate file for loading all other files and setting Soar parameters
  - Make it responsible for all CLI commands other than “sp {...}”
  - Tip: Begin its filename with “\_” so that it is always first in the directory.
2. Group rules into separate files so that **each substate** has its own file.
  - Though if a substate has only 1 rule, you *might* include it in its superstate’s file.
  - Make a separate file for rules that fire in multiple substates.

In this project, we will have 3 states, and each gets its own file:

Take a moment to look through the project folder and familiarize yourself with these files.

File Name	Purpose
<code>_firstload_agent_starter.soar</code>	Loads all other files
<code>existing_code_06_main.soar</code>	Rules for state <b>suppliersort_main</b>
<code>agent_starter_tie.soar</code>	Rules for state <b>suppliersort_tie</b>
<code>agent_starter_find_subtotal.soar</code>	Rules for state <b>suppliersort_find_subtotal</b>

Provided solution files are named with “*instructor\_solution*” instead of “*agent\_starter*”.

## 2 Organize Rules In Each File

The following convention will help you easily navigate your rules:

1. Put elaboration rules at the top of the file
2. Then put preference rules that involve operators of different names, if any (e.g. “op-1 > op-2”)
3. Then put rules for each operator, in order of rule type:
  1. Proposal rules
  2. Preference rules specific to only this operator, if any
  3. Apply rules

**We've been using this convention  
already throughout our projects!**



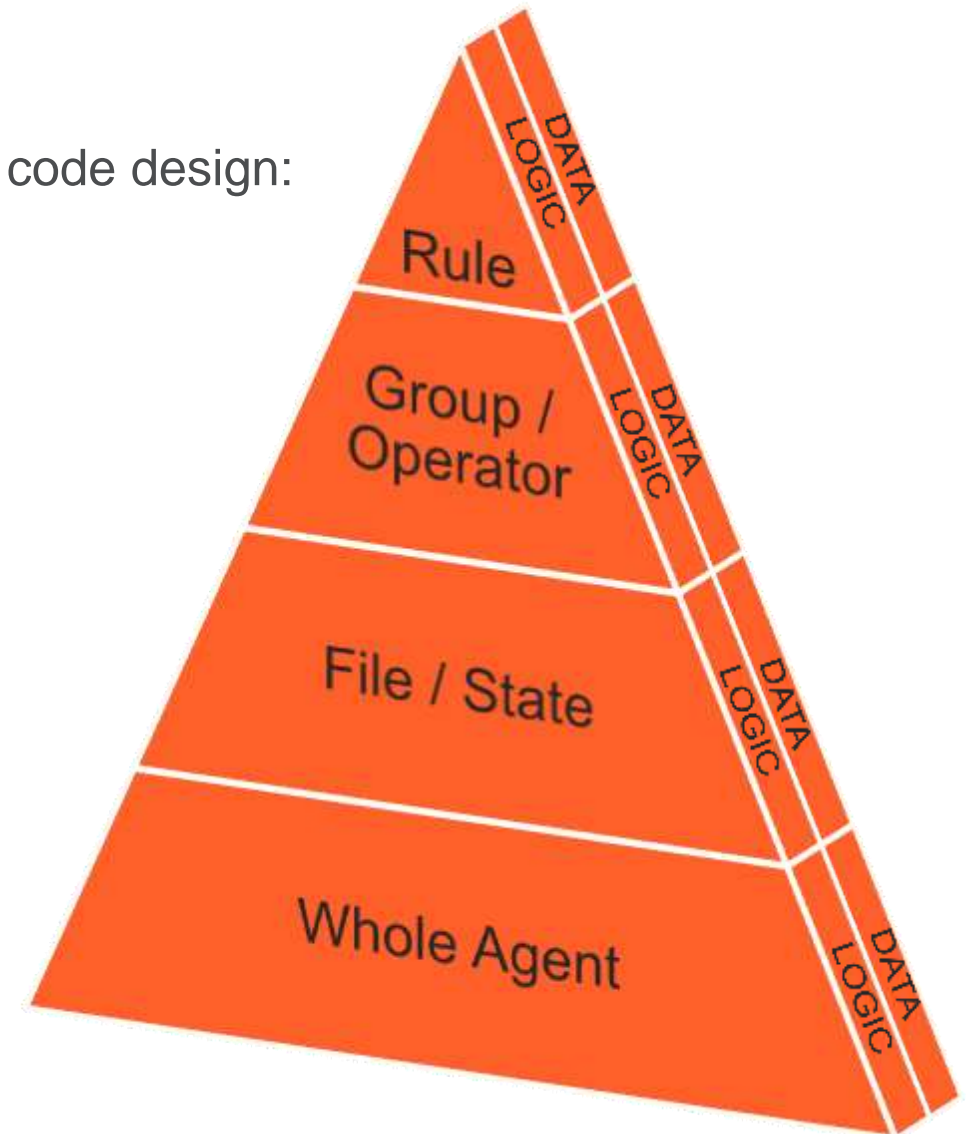
### 3 Write Documentation in Each File

Good documentations should explain each level of code design:

- What **each rule** does
- What **each group** of rules does
  - (such as for a single operator)
- What the processing in **each state** does
- What the **whole agent** does

For each level, documentation should explain:

- The LOGIC of the code
  - (the goal and steps of processing)
- The DATA STRUCTURES used by the code
  - (the WMEs and what they represent)



# Whole Agent README Convention



In the file that loads all other files (i.e. `_firstload_agent_starter.soar`):

- Have a README at the top that explains the whole agent:

```
### AGENT README #####
# AUTHOR: <names here>
# DATE: <date here>
# DESCRIPTION: <one-line description here>
#
## AGENT FILES
# file_1.soar          | <one-line file description here>
# file_2.soar          | <one-line file description here>
#
## AGENT LOGIC
# <longer whole-agent overview>
#
## I/O
# <required I/O WME schema here>
#####
```

**This is a useful reference for app developers  
who want to interface with your agent!**

Look at the README in `_firstload_agent_starter.soar` file for an example.

# WME Schema Convention

*Using “|” as a column separator makes it easier to copy this into a markdown table if desired.*

The following is a recommended way to document WME structures, such as for I/O:

# ^settings ID	0/1	The ID of the settings object
#   ^max-output-suppliers INT	0/1	The maximum suppliers to output
# ^priorities ID	1	The ID of the priorities object
#   ^total-cost FLOAT	1	The weight of the total-cost attribute
#   ^<attribute-name> INT	0+	The weight of the named attribute

Nested indentation shows tree structure

Data type of the WME value (ID / INT / FLOAT / STRING)

Short description of the WME's purpose. Prefix with "(ss)" if this WME is copied from a superstate.

WME attribute name.  
<brackets> indicate a wildcard pattern for WMEs other than those already described.

How many instances of this WME pattern can exist at once.  
(0/1) = Zero or One, (0+) = Zero or More, (1+) = One or More

# State File README Convention



Use a similar README format for the other files that hold your rules

- It should only describe the logic and WMEs used across that file as a whole

**SUBSTATE DEPTH** shows how deeply this logic appears in the agent's reasoning. (The topstate is depth 1.)  
If it can be at multiple depths, indicate so here as well.

After each operator's name is its set of unary preferences

```
### <STATE-NAME> STATE README #####
# AUTHOR: <names here>
# DATE: <date here>
# SUBSTATE DEPTH: <which layer in the state stack>
# DESCRIPTION: <one-line description here>
#
## OPERATORS
# (op1-name) + >          | <one-line operator description here>
# (op2-name) + = <       | <one-line operator description here>
#
## WORKING MEMORY
# <WME schema here>
#####
```

Look at the README in `existing_code_06_main.soar` for an example.

# Operator Header Convention



Put a multi-line header in front of each operator

If an operator has a non-trivial set of WMEs attached to it, add WME schema documentation to the header for that operator.

- Put the schema here instead of with the state-level README, because the WMEs are specific to the operator, not the whole state.
- See the header for (evaluate-weight) in agent\_starter\_tie.soar for an example.

```
#####  
##  OPERATOR: <name here>  
#  <full description here>  
##  WORKING MEMORY  
#  <WME schema here if needed>  
#####
```

# Individual Rule Documentation



There are 2 main types of comments you can add to rules:

## 1. Syntactic documentation strings

- These are loaded with the rule and can be inspected in an agent when printing the rule.
- Use these to summarize the rule's purpose.

## 2. Regular comments

- The parser ignores these when loading rules into the agent.
- Use these to label sections of a long rule to make it easier to read
- Or use these to explain a single complicated/unusual line.

```
sp {propose*suppliersort-main*select-supplier
  "Propose any supplier given on the input-link."
  (state <s> ^io.input-link.candidate-supplier <sup>
    ^supplier-list <sup-list>)
  # Check that there is an input supplier that is not yet in the list
  (<sup> ^name <name>)
  (<sup-list> -^supplier.name <name>) # Retract the proposal once this supplier is added
  -->
  (<s> ^operator <o> +)
  (<o> ^name select-supplier
    ^supplier <sup>))}
```

Describes  
a section

Describes a single line

Look at the rules in  
existing\_code\_06\_main.soar for  
more examples.

## That's a lot of comments!

*"But that seems like a lot of work to write all that documentation!"*

As with any large project (especially ones that other people might need to look at):

- It will save **YOU** time later when you need to come back and edit your own code!
- You will need to write documentation anyway for any formal industry project.
- It makes code development and collaboration **MUCH EASIER** even when it's fresh in your head!
  - For navigating your files to find the operators/rules you want to edit
  - For quickly double-checking a WME's name/usage without having to read through all the rules that might reference it.
  - For assisting LLM tools with suggesting meaningful code
- It doesn't actually take that much time if you write it as you write your code!

These are the documentation conventions that I find most useful as a developer.

## Other Conventions

There are other conventions you can use for organization and documentation

- Such as putting elaborations and each operator into their own files,
  - (The *Visual Soar* editor uses this approach.)
- Or making a separate markdown file for each README.

*But be consistent!*

- The goal is to make it easy for a programmer to understand what the code is doing.
- It's okay to keep rules for substates or operators in a single file when the code is small.
  - Like we did for our first projects
- Err on the side of more comments, not fewer!



# Augmenting Operators

For Detecting Multiple Attributes with a Shared Weight

## Lesson 07 Review: Handling Tied Weights

What if two attributes share the same weight?

- Then we want to sort by the *sum* of those attribute scores

*Example:* Say sustainability and quality **both** have weight 11

sustainability + quality	3+3	3+3	3+2	2+3	3+1	2+1
operator's supplier	supplier03	supplier05	supplier04	supplier02	supplier01	supplier03

***We first need to be able to detect when there are multiple attributes for a single weight!***

```
S1 ^io.input-link I2
    ^priorities P1
        ^total-cost 11.01
        ^sustainability 11
        ^quality 11
        ^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

    ...
```

## Goal: Detecting Multiple Attributes

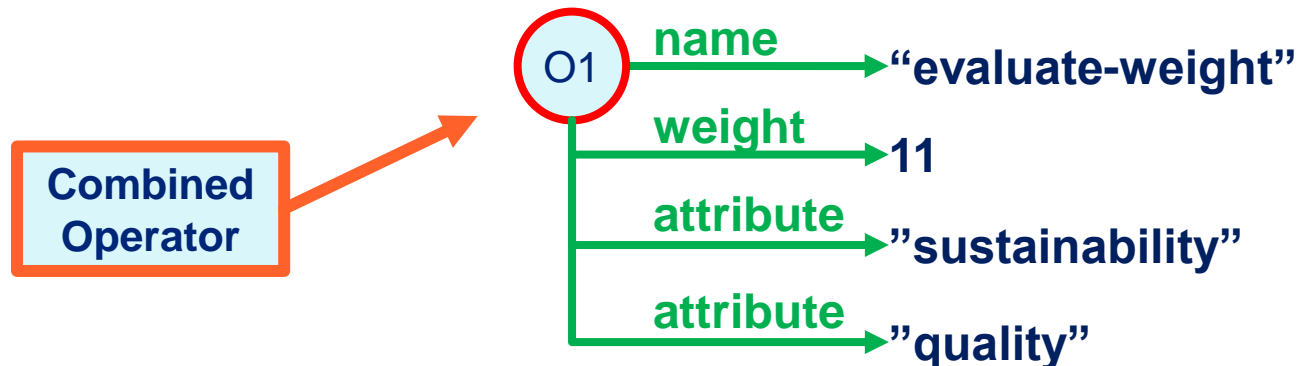
Our proposal for (evaluate-weight) only attaches a single attribute to the operator.

- If there are multiple attributes per weight, the proposal fires once for each one.



It would be nice if both attributes for the one weight were attached to the same operator.

- Then our apply rules could quickly detect if there was more than one attribute.



## Solution: Augment the Operator

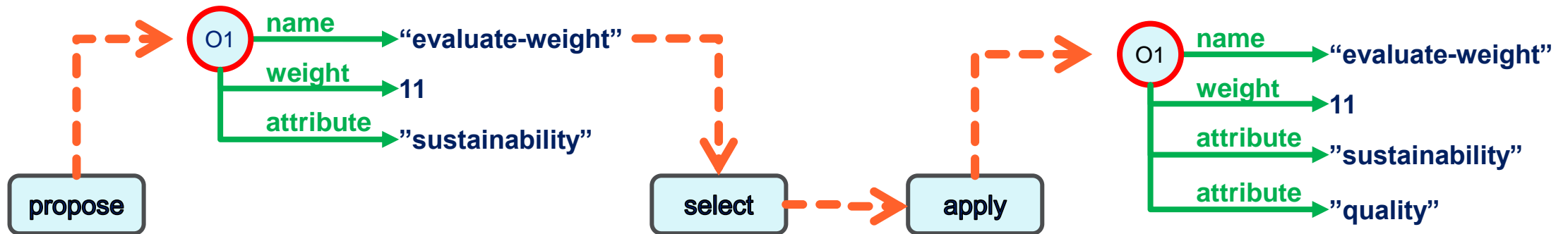
We can write a rule that will attach these attributes to the operator *after it is selected*.

- We can modify the ^operator object at any time just like any other object.
- Our new rule should match for any attribute that matches the current operator's weight, and then attach that attribute to the current operator.

This rule could be either an elaboration rule or an apply rule.

We prefer to use an *apply* rule, because it is more computationally efficient.

- (The ^operator condition is very specific, so Soar can usually exclude apply rules from match computations very quickly when they don't match.)
- Our apply rule will modify its own operator:



## STEP 1: New (evaluate-weight) Apply Rule

1. Open your agent\_starter\_tie.soar file and find the comments for STEP 1.
2. Write the desired rule according to the instructions.
  - It should fire once for each input priority attribute that has the selected weight value and attach that attribute to the selected operator.
    - HINT: Reference input priority attributes in the same way the proposal does.
  - The RHS will add a single ^attribute WME to the selected operator.
3. Test that your rule by running for 5 steps.
  - After the agent evaluates total-cost, you should see two messages from your new rule firing for weight 11, one for each of quality and sustainability.

```
total-cost: Prefer supplier04 > supplier00
total-cost: Prefer supplier05 > supplier06
4: 0: 013 (evaluate-weight)
** Evaluating attribute: quality at weight 11
** Evaluating attribute: sustainability at weight 11
quality: Prefer supplier03 > supplier04
quality: Prefer supplier05 > supplier04
5: 0: 011 (evaluate-weight)
```

## An i-supported Apply Rule?

This new rule is actually i-supported, not o-supported.

- Its results are undone as soon as its instantiation no longer matches. (see Lesson 06)

In Soar, any rule that *modifies* an ^operator object is given i-support, even apply rules.

- So operator proposals/preferences will *always* be i-supported (they modify operators).
- See Soar Manual section 2.7.6 (page 32) for more explanation.

This makes no difference for our agent.

- Our new rule won't retract before the proposal does.

But sometimes (rarely) this behavior becomes important.

**Main takeaway:** Treat rules that augment operators as if they are elaboration rules.

```
sp {apply*my-op*modify-operator
    "An i-supported apply rule"
    (state <s> ^operator <o>
        ^fizz <buzz>)
    (<o> ^name my-op)
    -->
    (<o> ^fizz <buzz>)}
```

# **STEP 2: Detecting Multiple Attributes**

And Triggering and New Substate

## Review: Desired (evaluate-weight) Behavior

If there is only 1 attribute for the selected operator's weight:

- Prefer the supplier with the best score for that attribute.
- (This is the behavior we've already written.)

But if there are more attributes:

- Calculate subtotals for each supplier across these attributes (in a new substate).
- Prefer the supplier with the best subtotal.

This means we **don't** want our existing apply rules to fire when there is more than 1 attribute!



```
total-cost: Prefer supplier04 > supplier00
total-cost: Prefer supplier05 > supplier06
4: 0: 013 (evaluate-weight)
** Evaluating attribute: quality at weight 11
** Evaluating attribute: sustainability at weight 11
quality: Prefer supplier03 > supplier04
quality: Prefer supplier05 > supplier04
5: 0: 011 (evaluate-weight)
```



## Conditional Sets of Apply Rules

When there is only 1 attribute for the selected weight:

- Our existing apply rules sort by supplier score for that attribute.

When there is >1 attribute for the selected weight:

- Initially, *no* apply rule should match.
  - This will trigger an *operator no-change* impasse
  - We will calculate our subtotal in the resulting substate.
  - (We need to modify our old apply rules so they *don't* match in this case.)
- Then an apply rule should match and create a preference via the calculated subtotal.
  - (We also still need to make this apply rule.)

## STEP 2: Modify the Old Apply Rules

1. In `agent_starter_tie.soar`, find the rule marked with comment “STEP 2.1”.
2. Add a new LHS condition that tests that the `^operator` object has no `^attribute` WME values that are not `<attr>`.
  - Notice the double negative: You will need both a negation condition and a not-equals (“`<>`”) comparison.
3. Find the rule marked “STEP 2.2” and add the same new condition there.
  - (NOTE: We don’t need to add this negative condition to the `prefer*by-total-cost` rule, because `total-cost` is guaranteed to never share a weight with another attribute.)
4. Test your code by running for 5 steps again.
  - You should see only an *operator no-change* impasse after the “\*\* Evaluating attribute ...” messages.

```
total-cost: Prefer supplier04 > supplier06
total-cost: Prefer supplier05 > supplier06
      4:      0: 014 (evaluate-weight)
** Evaluating attribute: quality at weight 11
** Evaluating attribute: sustainability at weight 11
      5:      ==>S: S3 (operator no-change)
```

# STEP 3: Error Catching

For Quicker Debugging

## Provided Code For `suppliersort-find-subtotal`

Open your `agent_starter_find_subtotal.soar` file.

- This file is for the `suppliersort-find-subtotal` substate
- This substate will calculate our subtotals.

It provides rules that name the state and copy important structures down for you.

You will need to fill in the steps for summing the subtotal values,

- Referencing the WM Schema provided in the README at the top of the file.

Review the file now and familiarize yourself with the README and the provided content.

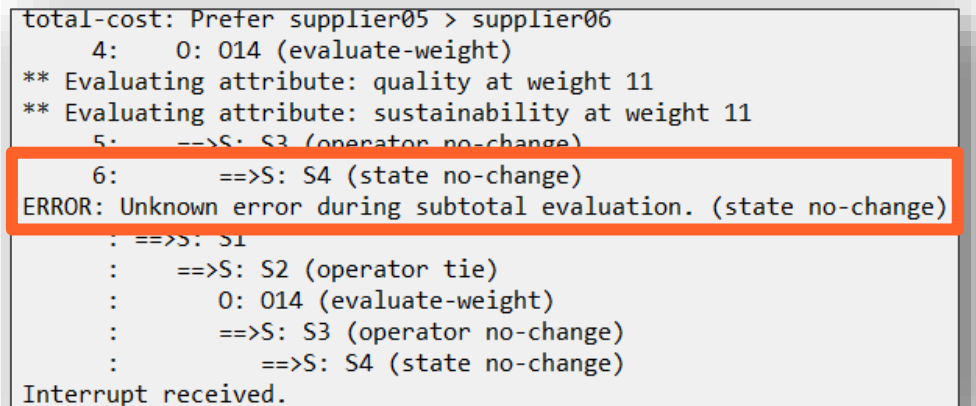
## STEP 3: Error Catching

The `suppliersort-find-subtotal` state is the deepest state our agent should ever reach.

- If the agent hits any more impasses, it indicates an error.

It is very good practice to write rules that catch error situations in your agent.

- Similar to try/catch blocks in classic programming languages
1. Uncomment the “STEP 3” rule found at the end of `agent_starter_find_subtotal.soar`.
    - Notice the comment section header that notes this rule is technically for a different state than the rest of the file. But the rule is placed in this file because it catches errors from logic in this file.
  2. Fill in the underscores according to the instructions in the comments.
    - (We’ll explain the provided RHS next.)
  3. Test your rule by entering “run” on your agent.
    - You should see an interrupt after step 6.



```
total-cost: Prefer supplier05 > supplier06
4: 0: 014 (evaluate-weight)
** Evaluating attribute: quality at weight 11
** Evaluating attribute: sustainability at weight 11
5: ==>S: S3 (operator no-change)
6: ==>S: S4 (state no-change)
ERROR: Unknown error during subtotal evaluation. (state no-change)
: ==>S: S1
: ==>S: S2 (operator tie)
: 0: 014 (evaluate-weight)
: ==>S: S3 (operator no-change)
: ==>S: S4 (state no-change)
Interrupt received.
```

# The `cmd` RHS Function

Running CLI Commands From Rules

## Using cmd For Custom Printouts

Our error catching rule included the following in its (write) arguments:

- (cmd |print --stack|)

The (cmd) RHS Function:

1. Runs the given string argument as a CLI command.
  - (In this case, we use it to tell Soar to print out the state stack.)
2. Returns the string output of the given command (if any)

You can use (cmd) to print a custom message to help with diagnosing the caught error.

- This can be helpful when running your agent from an app rather than from the debugger.
- Try replacing the given command with |print <ss> -d 2|.

You can also use it to run any command you like!

```
total-cost: Prefer supplier05 > supplier06
4: 0: 014 (evaluate-weight)
** Evaluating attribute: quality at weight 11
** Evaluating attribute: sustainability at weight 11
5: ==>S: S3 (operator no-change)
6: ==>S: S4 (state no-change)
ERROR: Unknown error during subtotal evaluation (state no-change)
: ==>S: S1
: ==>S: S2 (operator tie)
: 0: 014 (evaluate-weight)
: ==>S: S3 (operator no-change)
: ==>S: S4 (state no-change)
Interrupt received.
```

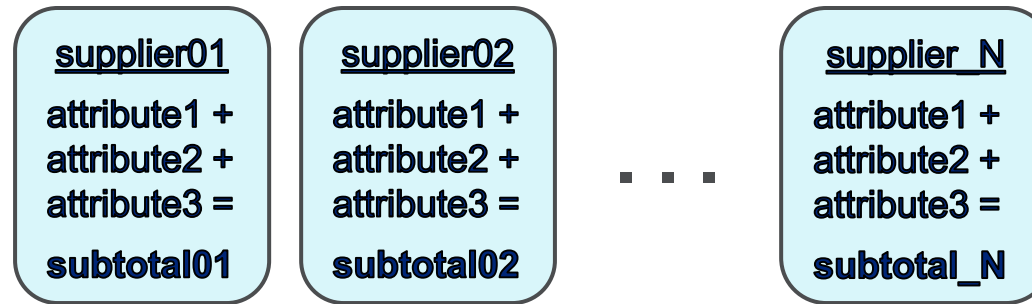
# Parallel Summation

Using Soar for Multi-Step Math



## Parallelism Constraints

We want to sum any number of attribute scores across any number of suppliers:



We can take advantage of Soar's parallelism when performing math in Soar!

- And avoid the need to iterate through individual suppliers!

But we have constraints common to all parallel logic:

- We can't add more than one value to the same variable at once!

# Parallelism Constraints – Example

Say we wanted to sum the addends (1,3,2) with an existing subtotal of 0, for a new subtotal of 6.

We could try to use a rule like this:

```
sp {elaboration*add-to-total
    (state <s> ^subtotal <old-total>
      ^addend <addend>)
  -->
  (<s> ^subtotal <old-total> -
    ^addend <addend> -
    ^subtotal (+ <old-total> <addend>))
```

**Removes the old total and the addend so that they are not added together again. Replaces the subtotal with the sum.**

Addend	Subtotal
1	0
3	
2	

But each addend would then be added to the same <old-total> value in parallel,

- Replacing 1 old subtotal with 3 new subtotals!!!

(<s> ^subtotal 0 -  
 ^addend 1 -  
 ^subtotal 1)

(<s> ^subtotal 0 -  
 ^addend 3 -  
 ^subtotal 3)

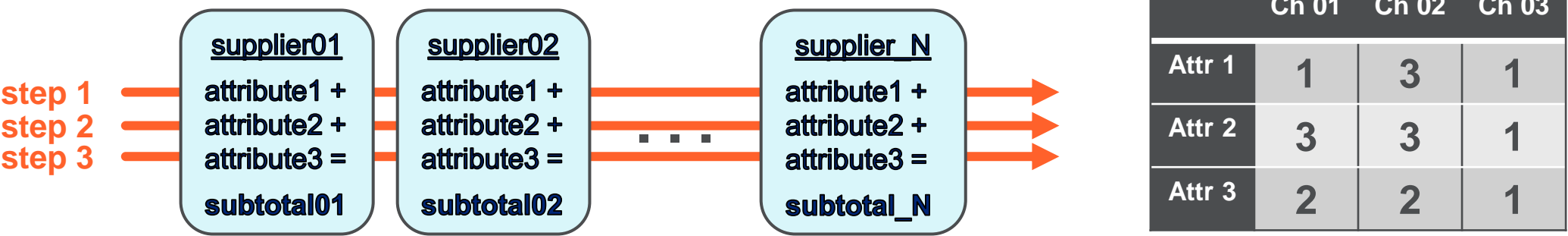
(<s> ^subtotal 0 -  
 ^addend 2 -  
 ^subtotal 2)

Subtotal
1
3
2

# Guided Parallelism with Operators

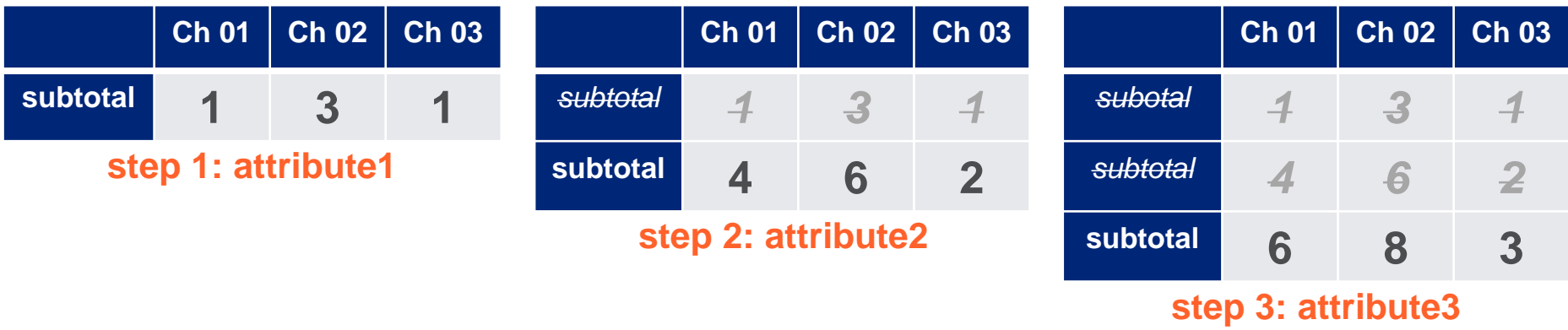
We can only add a single value to each subtotal at a time.

- But we *can* add to each subtotal in parallel!



We can use operators to select a *particular attribute* to add at a time,

- And then add the selected attribute's scores to all supplier subtotals in parallel:



## STEP 4: Proposing (add-attribute)

We want to propose an instance of the (add-attribute) operator for each attribute that shares the given weight for this substate.

1. Find the rule marked “STEP 4” in your agent\_starter\_find\_subtotal.soar file.
2. Uncomment it and fill in the underscores as directed to make the proposal rule.
3. Test your code by running your agent until your error-catch rule interrupts it.
  - Notice the provided apply rule (right below your new rule) that prints the attribute.
  - If your rule works correctly, it should trigger this apply rule before reaching an impasse:

```
4:      0: 013 (evaluate-weight)
** Evaluating attribute: quality at weight 11
** Evaluating attribute: sustainability at weight 11
5:      ==>S: S3 (operator no-change)
6:      0: 016 (add-attribute)
** Summing attribute: quality
7:      ==>S: S4 (operator no-change)
```

## STEP 5: Applying (add-attribute) – Initialize Subtotals

We want to use an apply rule to initialize the subtotal for each supplier

- Only whenever the subtotal doesn't yet exist.
1. Find the apply rule marked “STEP 5” in the same file.
  2. Uncomment it and fill in the underscores as directed.
    - You'll want to use a negated condition chain.  
(e.g. <s> -^fizz.buzz <var>)
    - You'll need to read the README's WM schema and create the ^subtotal structure to match what is described there.
  3. Test your code by running your agent until interrupted.
    - “print <ss> --depth 2” should show your initialized subtotals under S3 in WM.

```
p <ss> -d 2
(S3 ^attribute operator ^choices none ^epmem E
  ^name suppliersort-find-subtotal ^opera
  ^operator 017 ^priorities P1 ^quiescenc
  ^subtotal T2 ^subtotal T3 ^subtotal T1
  ^supplier C5 ^supplier C6 ^type state ^
(E3 ^command C12 ^present-id 1 ^result R8)
(016 ^attr quality ^name add-attribute)
(017 ^attr sustainability ^name add-attribu
(P1 ^availability 8 ^packaging 8 ^quality 11
  ^total-cost 11.010000)
(12 ^command C12 ^result R8)
(T2 ^supplier C6 ^value 0 ^weight 11)
(T3 ^supplier C7 ^value 0 ^weight 11)
(T1 ^supplier C5 ^value 0 ^weight 11)
(S2 ^attribute operator ^choices multiple ^e
  ^item 06 ^item 05 ^item-count 3 ^name
  ^non-numeric 06 ^non-numeric 05 ^non-
  ^operator 014 + ^operator 09 + ^opera
  ^operator 013 ^operator 010 + ^operat
  ^quiescence t ^reward-link R4 ^smem L
  ^weight-evaluated 11.010000)
(C7 ^availability 3 ^name supplier05 ^packag
  ^sustainability 3 ^total-cost 25.0000
(C5 ^availability 3 ^name supplier03 ^packag
  ^sustainability 3 ^total-cost 25.0000
(C6 ^availability 3 ^name supplier04 ^packag
  ^sustainability 3 ^total-cost 25.0000
```

## STEP 6: Applying (add-attribute) – Add to Subtotals

We want to use an apply rule to add appropriate attribute scores to each subtotal

1. Find the apply rule marked “STEP 6” in the same file.
2. Uncomment it and fill in the underscores as directed. It should:
  - Remove the old subtotal’s value and replace it with the sum of the old subtotal plus a single attribute addend.
  - Mark the selected attribute as summed, so your proposal retracts.
    - (Make sure your proposal tested for `-^addends-summed <attr>.`)
3. Test your code by running your agent until interrupted.
  - You should now see (add-attribute) selected and applied twice,
  - Followed by a *state no-change* instead of an *operator no-change*.
  - “`print <ss> -d 2`” should show non-zero values for subtotals.

```
S: ==>S: S3 (operator no-change)
6:      0: 016 (add-attribute)
** Summing attribute: quality
7:      0: 017 (add-attribute)
** Summing attribute: sustainability
8:      ==>S: S4 (state no-change)
```

```
(E3 ^command C12 ^present-id 1 ^result R8)
(P1 ^availability 8 ^packaging 8 ^quality 11 ^speed
   ^total-cost 11.010000)
(I3 ^command C13 ^result R9)
(T2 ^supplier C6 ^value 5 ^weight 11)
(T3 ^supplier C7 ^value 6 ^weight 11)
(T1 ^supplier C5 ^value 6 ^weight 11)
(S2 ^attribute operator ^choices multiple ^epmem E
   ^item 06 ^item 05 ^item-count 3 ^name suppl
   ^non-numeric 06 ^non-numeric 05 ^non-numeric
   ^operator 014 + ^operator 09 + ^operator 01
   ^operator 013 ^operator 010 + ^operator 013
   ^quiescence t ^reward-link R4 ^smem L2 ^sup
   ^weight-evaluated 11.010000)
```

# Negated Conjunctions

Testing That a Particular Combo Doesn't Exist

## Almost There: Returning Results

Our (add-attribute) operator calculates subtotals now!

- The only step remaining for this substate is to return the subtotals to the superstate. We want to return a supplier's subtotal as soon as all its attributes have been summed.
- Return any supplier's subtotal as soon as there is an “^addends-summed” WME for each of its attributes for this substate's weight.

Can we use another apply rule to do that?

- Not easily: Our proposal retracts as soon as the ^addends-summed WME is added.
  - The operator will end before an apply rule could react to ^addends-summed.

The simplest approach is to instead use an elaboration.

- It can return the results whether an operator is selected or not.



## Detecting Completion

A more formal way to say that there is an ^addends -summed WME for each of a supplier's attributes for the substate's weight is as follows:

- For a given supplier's subtotal and a given substate's weight, It is **NOT** the case that:
  1. The supplier has an attribute score for that weight, **AND**
  2. There is no ^addends -summed WME for that attribute.

This is a bit more complex than LHS tests we've worked with so far.

- Notice that neither of the following would work:

✗

```
(<s> ^weight <weight>  
  ^priorities.<attr> <weight>  
  ^subtotal.supplier.<attr> <score>  
  ^addends-summed <attr>)
```

**“Any 1 attribute has a  
matching ^addends -summed.”**

✗

```
(<s> ^weight <weight>  
  ^priorities.<attr> <weight>  
  ^subtotal.supplier.<attr> <score>  
  -^addends-summed <attr>)
```

**“Any 1 attribute is missing a  
matching ^addends -summed.”**

## Negated Conjunctions

In Lesson 03 we learned about negative WME tests.

- e.g. “-(<s> ^fizz |buzz|)”

In Lesson 07 we learned about condition conjunctions.

- e.g. “(<s> ^fizz {<buzz> <> |bizz|})”

Soar lets you surround whole WME blocks with “{ }” brackets to form a conjunction that you can negate!

- e.g. “-{( <s> ^fizz <buzz>  
                  ^child <c>  
                  (<c> ^fizz <buzz> ) } }”
- This example tests, “It is not the case that any (<s> ^fizz <buzz>) has the same value as any (<s> ^child.fizz <buzz>).”

## Using a Negated Conjunction

To represent our desired test:

- It is **NOT** the case that:
  1. The supplier has an attribute score for that weight, **AND**
  2. There is no ^addends-summed WME for that attribute.

We can use the following:

```
(<s> ^weight <weight>
      ^subtotal.supplier <sup>)
-{{(<sup> ^<attr> <score>)
   (<s> ^priorities.<attr> <weight>
    -^addends-summed <attr>)}}
```

# Using a Negated Conjunction

To represent our desired test:

- It is **NOT** the case that:
  1. The supplier has an attribute score for that weight, **AND**
  2. There is no ^addends-summed WME for that attribute.

We can use the following:

It is NOT the case that...

```
(<s> ^weight <weight>
  ^subtotal.supplier <sup>)
-{{(<sup> ^<attr> <score>)
  (<s> ^priorities.<attr> <weight>
    -^addends-summed <attr>)}}
```

...the supplier has an attribute score for the selected weight...

...AND there is no ^addends-summed WME for that attribute.

## STEP 7: Returning Subtotals

1. Find the elaboration rule marked “STEP 7” in agent\_starter\_find\_subtotal.soar.
2. Uncomment it and fill in the underscores as directed.
  - Use the negated conjunction shown on the previous slide.
    - Read it carefully to make sure you understand the logic behind it!
3. Test your code by running your agent until interrupted.
  - “print <ss> -d 2” should show that your subtotals have been copied to S2.
  - (We still impasse because the superstate doesn’t respond to subtotal results yet!)

```
(E3 ^command C12 ^present-id 1 ^result R8)
(P1 ^availability 8 ^packaging 8 ^quality 11 ^speed 7 ^sustainability 11
   ^total-cost 11.010000)
(I3 ^command C13 ^result R9)
(T2 ^supplier C6 ^value 5 ^weight 11)
(T3 ^supplier C7 ^value 6 ^weight 11)
(T1 ^supplier C5 ^value 6 ^weight 11)
(S2 ^attribute operator ^choices multiple ^epmem E2 ^impasse tie ^item 04
   ^item 06 ^item 05 ^item-count 3 ^name suppliersort-tie
   ^non-numeric 06 ^non-numeric 05 ^non-numeric 04 ^non-numeric-count 3
   ^operator 09 + ^operator 010 + ^operator 014 + ^operator 011 +
   ^operator 012 + ^operator 013 + ^operator 014 ^priorities P1
   ^quiescence t ^reward-link R4 ^smem L2 ^subtotal T3 ^subtotal T1
   ^subtotal T2 ^superstate S1 ^type state ^weight-evaluated 11.010000)
```

## STEP 8: Respond to Returned Subtotals

We're almost done!

1. Open `agent_starter_tie.soar` and find the rules for STEP 8.1 and STEP 8.2.
2. Uncomment these. They are already written for you.
  - They respond to returned subtotals by creating preferences for the topstate,
    - Very similar to other rules we wrote previously for the `suppliersort-tie` state.
  - But they still need proper documentation!
3. Examine the rules to figure out how they work.
4. Write meaningful comments and documentation strings that explains them.
  - Write what would explain them best to you as a programmer!
5. Test the agent by running it.
  - It should now run to completion, using subtotals to make preferences from the `suppliersort-tie` state!

**Congratulations!** You have completed Lesson 08!

```
19:    ==>S: S6 (operator no-change)
20:      0: 028 (add-attribute)
** Summing attribute: quality
21:      0: 027 (add-attribute)
** Summing attribute: sustainability
Weight 11 totals: Prefer supplier02 > supplier01
Weight 11 totals: Prefer supplier01 > supplier06
Weight 11 totals: Prefer supplier02 > supplier06
22: 0: 03 (select-supplier)
** OUTPUT: Then try supplier02
23: 0: 02 (select-supplier)
** OUTPUT: Then try supplier01
24: 0: 07 (select-supplier)
** OUTPUT: Then try supplier06
25: 0: 08 (output-supplier-list)
*** DONE ***
```