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

Project 05: Multi Apply

By Dr. Bryan Stearns, 2024

Problem

Our agent does not output suppliers with any particular order.

- It selects suppliers in a particular order, but that order is not represented in the final output.

Solution

- When the agent selects a supplier, append it to a linked-list data structure (which we'll define) in WM.
- After all suppliers have been added to the list, in order, then copy this list to the output-link.
- (We will not reuse code from previous projects this time, because we will need to make slight changes to the rules we wrote before.)

Project Goal

We want our agent to output an ordered list of recommended suppliers. The resulting output-link structure should be as follows:

```
S1 ^io.output-link I3
    ^supplier-list C9
        ^count 6
        ^first-supplier C10
        ^last-supplier C15
        ^supplier C10
            ^name |supplier05| ^rank 1 ^next C11
        ^supplier C11
            ^name |supplier03| ^rank 2 ^next C12
        ...
        ^supplier C15
            ^name |supplier02| ^rank 6)
```

Lesson 05 – Outline

This lesson explains the following new concepts:

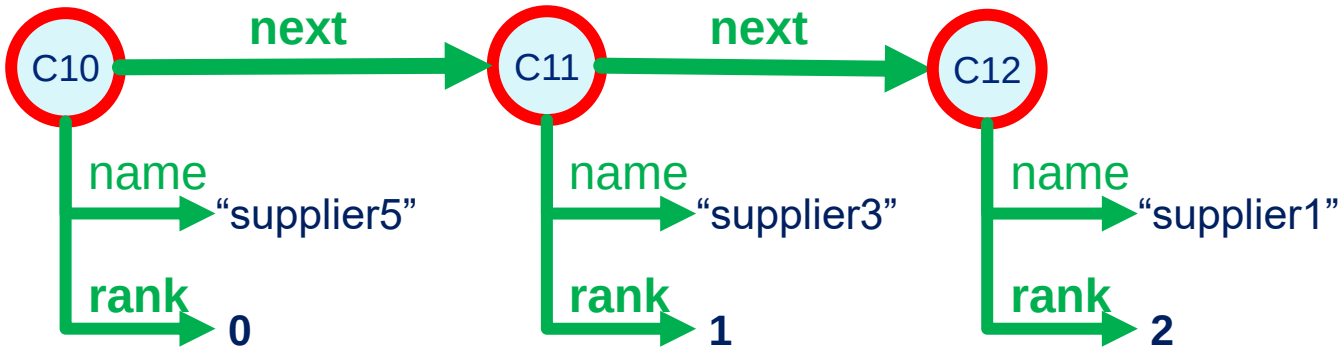
1. Making a custom linked-list with WMEs
2. Using an `(init)` operator
3. Using the "require" preference
4. Multiple apply rules for a single operator
5. Using math RHS Functions

Linked-Lists with WMEs

A Custom Graph Data Structure

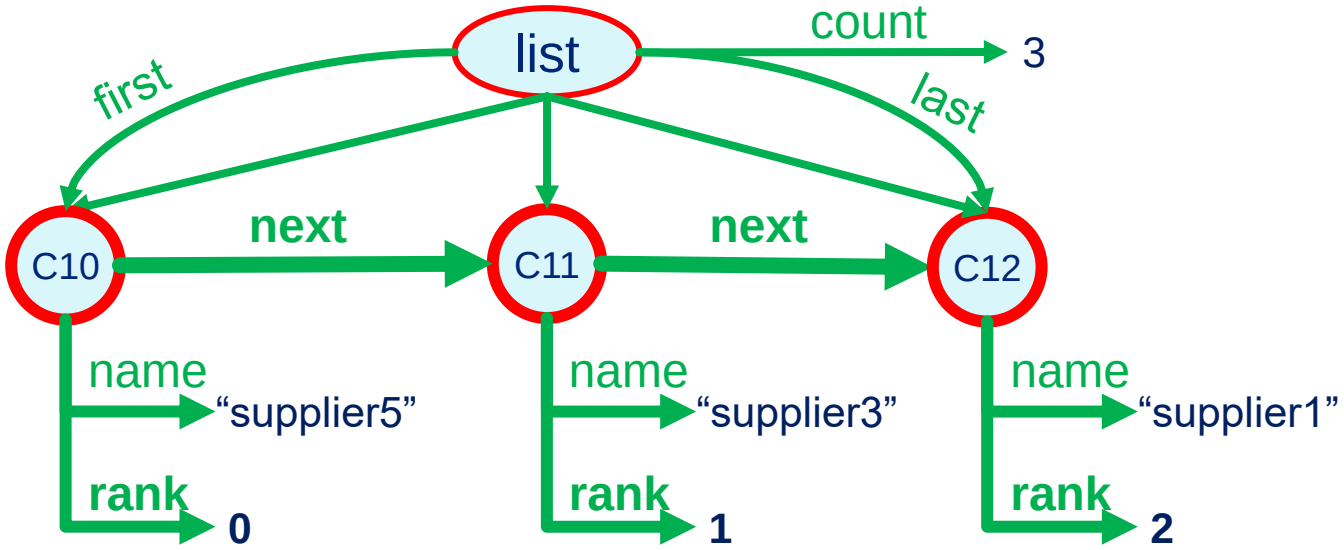
Linked-List Elements

Soar's WM graph format lets us make any data structure we like.
The simplest kind of linked-list would look something like this:



Linked-List for Soar

For Soar, it is useful to have a parent node with direct links to each list element.



We connect from the parent node to every list element so that agent rules can easily access any item in the list without traversing the whole sequence.

Linked-List Format

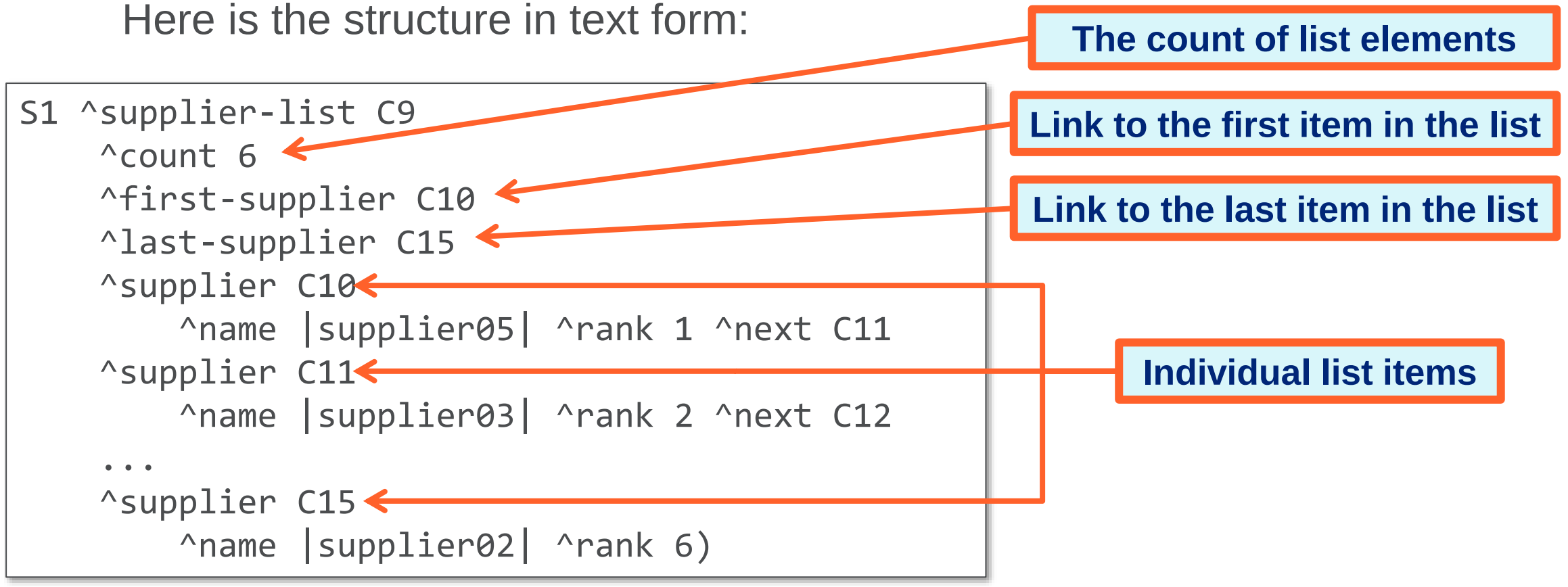
Here is the structure in text form:

```
S1 ^supplier-list C9
    ^count 6
    ^first-supplier C10
    ^last-supplier C15
    ^supplier C10
        ^name |supplier05| ^rank 1 ^next C11
    ^supplier C11
        ^name |supplier03| ^rank 2 ^next C12
    ...
    ^supplier C15
        ^name |supplier02| ^rank 6)
```

Linked-List Format

Here is the structure in text form:

```
S1 ^supplier-list C9
    ^count 6
    ^first-supplier C10
    ^last-supplier C15
    ^supplier C10
        ^name |supplier05| ^rank 1 ^next C11
    ^supplier C11
        ^name |supplier03| ^rank 2 ^next C12
    ...
    ^supplier C15
        ^name |supplier02| ^rank 6)
```



Building the List

There are 3 distinct kinds of operations we need to do with our list:

1. **Create** the empty list structure
   - Create the empty parent node
   - Attach to it a `^count` of 0

2. **Add the first item** to the list
   - Attach the new item to the parent node
   - Initialize the `^first-supplier` and `^last-supplier` links to point to the new item.
   - Replace the old list "`^count 0`" value with "`^count 1`"

3. **Append** subsequent items to the list
   - Attach the new item to the parent node
   - Add a `^next` link from the old `^last-supplier` item to the new item
   - Replace the old `^last-supplier` link with a link to the new item
   - Replace the old list "`^count`" value with (1 + the old count value)

Using the `(init)` Operator

To Create the Empty List Structure

The `(init)` Operator

We want to create our empty list when the agent first starts up.

So we'll define an `(init)` operator:

- It will create initial structures that the agent will need later.
- The agent should select it before anything else.

Most Soar agents use an `(init)` operator of some sort.

We can modify the `(init)` operator later to also create any other structures we might want in addition to our list.

Using the Require Preference

To Guarantee an Operator is Selected First

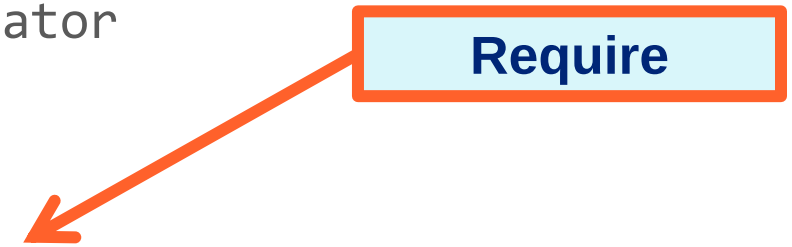
The Require Preference

Require (“**!**”) is a unary preferences we can use when proposing an operator.

It guarantees that any operator with that preference is selected over other operators.

- If more than one operator has the Require preference at once, this creates an impasse.

```
sp {propose*my-required-operator
    (state <s> ^type state)
    -->
    (<s> ^operator <o> + !)
    (<o> ^name |my-required-operator|)}
```



It is a great way to guarantee that our `(init)` operator is selected before other operators.

Let's Write Some Code!

1. Open your `agent_starter.soar` file and look at the first 2 rules.
   - They are propose and apply rule shells for an `(init)` operator.
2. Fill out the rules as instructed in comments so `(init)` creates an empty list.
3. Run the agent for 2 steps to test it
   - After 2 steps, enter the command "`print S1 -d 2`" into the debugger to see if your apply rule worked. You should see the following output:

```
(S1 ^epmem E1 ^io I1 ^reward-link R1 ^smem L1
        ^supplier-list C9 ^type state)
  (E1 ^command C1 ^present-id 1 ^result R2)
  (I1 ^input-link I2 ^output-link I3)
  (L1 ^command C2 ^result R3)
  (C9 ^count 0)
```

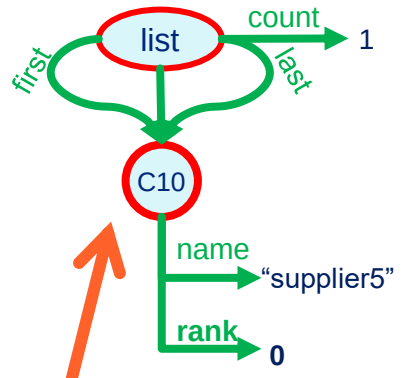
4. Proceed to the next slide when you are done.

Updating (select-supplier)

Previously, (select-supplier) created a ^selected WME on the state.

We want it to instead add selected suppliers to our list, in selection order:

apply: *"First try supplier05"*



List of 1

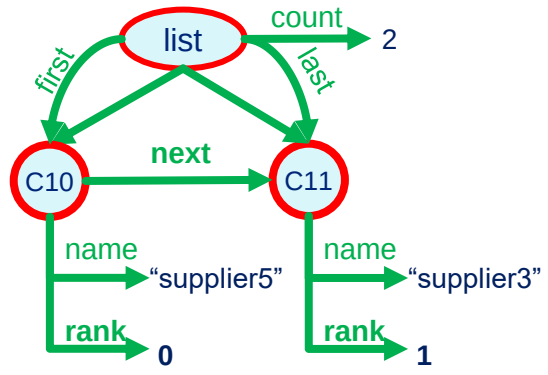
next decision cycle

propose:

- supplier01
- supplier02
- supplier03
- supplier04
- supplier06

select: supplier03

apply: *"Then try supplier03"*



List of 2

...

next decision cycle

Let's Write Some Code!

1. Uncomment the propose rule for `(select-supplier)`
   - Note that a preference rule is already provided beneath it.
2. Fill out the rule so the proposal retracts once the selected supplier exists in the list.
   - (We'll make an apply rule later that adds it to the list.)
   - Test that the list object has no "`^supplier`" object with the same name as the proposed supplier.
   - HINT: Use a negation condition with dot-notation: `(<sup-list> -^__.__ <name>)`
3. Run the agent for 3 steps to test it.
   - You should see the following →

```
step 3
     1: O: O1 (init)
     2: O: O6 (select-supplier)
     3: ==>S: S2 (operator no-change)
```

4. Proceed to the next slide when you are ready to make the apply rule(s).

Multiple Apply Rules for One Operator

Using Different Rules For Different Apply Scenarios

No Limit to Apply Rules

There is no limit to the number of apply rules you give an operator

- In Soar, all rules that can fire will fire

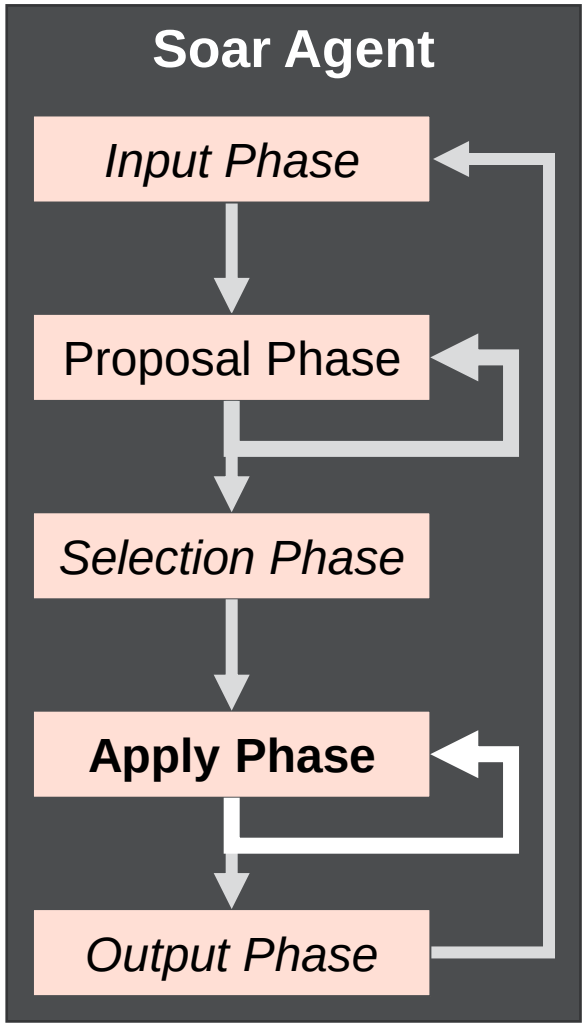
If more than one apply rule matches for an operator, they all fire together.

Apply rules can even fire in sequence one after another.

Example:

- One apply rule fires: “If no object “parent” exists, then create it.”
- Then another apply rule fires: “If object “parent” exists, create object “child” under it.”
- And so on.

The *Apply Phase* will only end once no more rules fire!



Different Apply Scenarios

The `(select-supplier)` operator should grow the recommendation list

We need two apply rules, one for each of two scenarios:

1. **The list is empty**
   - If so, initialize "`^first-supplier`" to the new item, and add the new item
2. **The list is not empty**
   - If so, add the new item and add a "`^next`" link from the old "`^last-supplier`" to it.

In both cases, adding a new item must involve three actions:

- Attach the new item to the parent node
- Replace the old `^last-supplier` link with a link to the new item
- Replace the old list "`^count`" value with (1 + the old count value)

Let's Write Some Code!

1. Uncomment the first apply rule for (`select-supplier`)
   - "`apply*suppliersort-main*select-supplier*create-first`"
2. Fill out the rule according to the instructions in the comments
   - The LHS should end up being about 5 lines long.
   - For the RHS, you can copy this code for the indicated sections:

```
# Add the first supplier to the list
(<sup-list> ^supplier <sup-new>)
(<sup-new> ^name <sup-name>
           ^rank 1)
```

```
# Create the first- and last-supplier augmentations
(<sup-list> ^first-supplier <sup-new>
            ^last-supplier <sup-new>)
```

   - You'll need to add RHS code for updating `^count` on your own.
3. Proceed to the next slide when ready to test your code.

Let's Write Some Code!

```
step 4
     1: O: O1 (init)
     2: O: O6 (select-supplier)
** OUTPUT: First try supplier05
     3: O: O4 (select-supplier)
     4: ==>S: S2 (operator no-change)
```

4. Run the agent for 4 steps to test it
   - You should see the following →
   - The first (`select-supplier`) works.
   - The second still needs an apply rule.
     – The one we just wrote won't match, since the list isn't empty anymore.
5. Uncomment the second apply rule for (`select-supplier`)
   - "`apply*suppliersort-main*select-supplier*append`"
6. Fill out the rule's LHS according to the instructions in the comments
   - The first few lines can be the same as for the first apply rule
   - Instead of testing for `^count 0`, test that `^last-supplier`'s name is not the operator's supplier's name, so the rule will only fire once.
7. Proceed to the next slide when you are ready to make the RHS.

Using Math RHS Functions

To Increment the List Element Count

Using the + RHS Function

Other math RHS Functions work similarly.
(See the manual for details for each function.)

To increment the `^count` WME, we can use the (+) RHS Function

- (+) takes any number of arguments and returns their sum
- In our case, if `<count>` is the old count that we want to increment, we can write:

```
# Update the list's count of items
(<sup-list> ^count <count> -
            ^count (+ 1 <count>))
```

to replace the old count with a new count that is 1 more than the old count.

Note that RHS Functions only work on the *RHS* of a rule.

- **This means that you cannot test the sum of items on the LHS.**
- **If you ever find you want to use math operations on the LHS, you must first perform the operations using a rule's RHS and store the computed value in WM.**
- **Then you can compare the stored value with something else in another rule's LHS.**

Let's Write Some Code!

1. Fill out the rule's RHS according to the instructions in the comments
   - You will need to use the (+) RHS Function twice
     - Once to set the new list item's `^rank` to (1 + the old list count)
     - Once to set the new list `^count` to (1 + the old list count)
   - Reference `instructor_solution.soar` if you get stuck.
2. Run the agent for 4 steps to test it.
   - You should see the following →
   - There should be no impasses.

```
step 4
     1: O: O1 (init)
     2: O: O6 (select-supplier)
** OUTPUT: First try supplier05
     3: O: O4 (select-supplier)
** OUTPUT: Then try supplier03
     4: O: O2 (select-supplier)
```

3. Proceed to the next slide when you are done.

Updated Output Operator

Notice that the `(output-supplier-list)` operator code is provided at the end of the `agent_starter.soar` file.

- `(output-supplier-list)` is almost the same as `(output-suppliers)` from Project 04.
- Take a quick look at the code to observe the differences.
  - We output the single list structure instead of individual suppliers.

Because `(output-supplier-list)` interrupts the agent, we can now test the whole agent by simply using the "`run`" command.

Final Results

After running your agent, you should see the following:

- Each of the 6 input suppliers is selected in order.
- The list is then output and the agent stops on its own.

You should be able to see the output appear in the output auto-update tab in the debugger, as shown.

- All 6 suppliers are present
- Each has correct `^next` and `^rank` values
- The list `^first-supplier`, `^last-supplier`, and `^count` values are correct

If so, **congratulations**!
You have completed Project 05!

```
run
     1: O: O1 (init)
     2: O: O6 (select-supplier)
** OUTPUT: First try supplier05
     3: O: O4 (select-supplier)
** OUTPUT: Then try supplier03
     4: O: O2 (select-supplier)
** OUTPUT: Then try supplier01
     5: O: O5 (select-supplier)
** OUTPUT: Then try supplier04
     6: O: O7 (select-supplier)
** OUTPUT: Then try supplier06
     7: O: O3 (select-supplier)
** OUTPUT: Then try supplier02
     8: O: O8 (output-supplier-list)
*** DONE ***
Interrupt received.
```

```
(I3 ^supplier-list C9)
  (C9 ^count 6 ^first-supplier I5 ^last-supplier I10 ^supplier I6 ^supplier I5
         ^supplier I7 ^supplier I8 ^supplier I9 ^supplier I10)
    (I5 ^name supplier05 ^next I6 ^rank 1)
    (I10 ^name supplier02 ^rank 6)
    (I6 ^name supplier03 ^next I7 ^rank 2)
    (I7 ^name supplier01 ^next I8 ^rank 3)
    (I8 ^name supplier04 ^next I9 ^rank 4)
    (I9 ^name supplier06 ^next I10 ^rank 5)
```

state | operator | stack | matches | op_pref | stats | input | output