##### **You said:**

You are a code translator for cognitive architectures between ACT-R and Soar. Start by telling me what you know about Soar?

###### **ChatGPT said:**

Soar is a general cognitive architecture designed to model and simulate human cognition. It has been developed over several decades and is one of the most well-known and widely used cognitive architectures alongside ACT-R. Here's an overview of key aspects of Soar:

### **🔧 Core Concepts and Architecture**

#### **1. Symbolic Production System**

Soar operates primarily as a **production-rule system**, using condition-action rules:

* **If-then** format rules (productions) operate on a working memory.
* Productions are fired in response to the current contents of working memory.

#### **2. Working Memory (WM)**

* All knowledge about the current situation is stored here as a **connected graph** of **identifier–attribute–value** triples (also called **WMEs**, or Working Memory Elements).
* The top-level structure is the **state** identifier, which anchors all substructures.

#### **3. Decision Cycle**

Soar's operation proceeds in a **decision cycle** with the following stages:

1. **Input Phase** – perception or external input is added to WM.
2. **Proposal Phase** – operators are proposed based on the current state.
3. **Decision Phase** – one operator is selected if multiple are proposed.
4. **Application Phase** – operator’s effects are applied via apply\* rules.
5. **Output Phase** – actions are sent to the environment or external systems.

#### **4. Operators and Substates**

* Soar decomposes behavior into **operators**, which modify the current state.
* If an impasse occurs (e.g., two operators are equally good), Soar **automatically creates a substate** to resolve it. This allows for hierarchical problem solving.

#### **5. Learning: Chunking**

* Soar supports **chunking**, a form of explanation-based learning.
* When a substate is resolved, a **new production rule** (chunk) is generated to avoid repeating the same subproblem in the future.

### **🧠 Knowledge Representation**

* All knowledge is **symbolic** and consists of structured graphs in working memory.
* Rules match against these structures using pattern matching.
* Goals, preferences, and operators are explicitly represented.

### **📦 Modules and Interfaces**

* Soar includes:  
  + **Input/Output Interface** (external sensor and effector representation).
  + **Episodic Memory** and **Semantic Memory** (extensions).
  + **Reinforcement Learning** module (with numeric preference learning).

### **🔄 Comparison to ACT-R**

| **Feature** | **Soar** | **ACT-R** |
| --- | --- | --- |
| Architecture | Production-rule-based | Hybrid (symbolic + subsymbolic) |
| Learning | Chunking (EBC) | Declarative and procedural with utility |
| Memory Types | WM, episodic, semantic | Declarative memory, procedural memory |
| Decision Making | Rule-based with preferences | Utility-based selection |
| Subgoals | Explicit substates for impasses | Goal stack |
| Modeling Style | More symbolic, hierarchical problem solving | More subsymbolic, ACT-based timing |

### **📘 Example: A Simple Rule in Soar**

soar

CopyEdit

sp {propose\*turn-on-light

(state <s> ^room <r>)

(<r> ^light off)

-->

(<s> ^operator <o> +)

(<o> ^name turn-on-light

^room <r>)

}

This rule proposes an operator turn-on-light when the room's light is off.

If you're working on **translating between Soar and ACT-R**, we’ll need to align their different assumptions:

* ACT-R uses **modules** and **buffers**, while Soar uses a **central working memory**.
* Operator application in Soar can be mapped to **production firing** and **goal stack updates** in ACT-R.
* The **decision processes** (especially how conflicts are resolved and rules are fired) need explicit attention during translation.

Would you like me to walk through how to translate a Soar rule into ACT-R format next?

##### **You said:**

What does the syntax of a Soar production look like?

###### **ChatGPT said:**

The syntax of a **Soar production rule** (also called a *production* or *rule*) follows a **condition-action** format, written using the sp { ... } construct. Here's the general structure:

### **🧩 Basic Soar Production Syntax**

soar

CopyEdit

sp {rule-name

(condition1)

(condition2)

...

-->

(action1)

(action2)

...

}

* sp means "Soar production".
* {rule-name} is an arbitrary name for the rule (usually category\*description).
* The --> separates the **conditions** (left-hand side, LHS) from the **actions** (right-hand side, RHS).
* **Conditions** match patterns in **working memory**.
* **Actions** change working memory (e.g., add WMEs, propose operators).

### **📌 Condition Format**

Each condition looks like:

soar

CopyEdit

(<id> ^attribute <value>)

* <id> is a variable (e.g., <s> for a state, <r> for a room).
* ^attribute accesses a slot in the identifier.
* <value> is either a variable (starting with <...>) or a constant (e.g., on, room, light).

Conditions may also include:

* **Negative conditions**: - before a condition (e.g., -(<s> ^status done))
* **Relational conditions** using =, >, <, etc., in preferences or numeric comparisons.

### **🎯 Action Format**

Actions modify working memory and typically take one of these forms:

soar

CopyEdit

(<id> ^attribute <value>) ; Add a WME

(<id> ^attribute <value> -) ; Remove a WME

(<id> ^operator <op> +) ; Propose an operator

* + means **propose** this operator.
* - means **remove** this WME.
* Without + or -, it's just a WME addition.

### **✅ Full Example: Propose Operator**

soar

CopyEdit

sp {propose\*move-left

(state <s> ^name move-state

^position <p>)

(<p> ^x >0)

-->

(<s> ^operator <o> +)

(<o> ^name move-left)

}

* **Conditions**: Current state is move-state, has a position with x > 0.
* **Actions**: Propose operator move-left.

### **⚙️ Operator Application Example**

soar

CopyEdit

sp {apply\*move-left

(state <s> ^operator <o>)

(<o> ^name move-left)

(<s> ^position <p>)

(<p> ^x <x>)

-->

(<p> ^x -)

(<p> ^x (- <x> 1))

}

* Removes old x and replaces with x-1.

### **📎 Notes**

* Soar does not allow side effects outside WM unless through **output link** (^io.output-link).
* You can chain actions, propose multiple operators, or use nested structures (^part-of, ^status, etc.).
* Variables must be bound on the LHS before being used on the RHS.

Would you like to see how this compares to ACT-R syntax or go deeper into specific production types (like preference rules, numeric comparisons, or substates)?

##### **You said:**

Here is what I know about Soar syntax for your reference: Knowledge in Soar is encoded in production rules. A rule has conditions on its Left Hand Side (LHS), and actions on the Right Hand Side (RHS): C --> A. Two of Soar's memories are of relevance here: the production memory (PM) or long-term memory, permanent knowledge in the form of production rules; and the working memory (WM), temporary information about the situation being dealt with, as a collection of elements (WMEs). The LHSs of productions test WM for particular patterns of WMEs. Unlike most other production systems, Soar has no syntactic conflict resolution to decide on a single rule to fire at each cycle. Instead, all productions whose conditions are satisfied fire in parallel. For example, the following rule proposes an operator 'eat' if we are hungry and desire not to be hungry. ;; Propose eat. sp {ht\*propose-op\*eat (state <s> ^problem-space <p> ^desired <d>) (<p> ^name hungry-thirsty) (<d> ^hungry no) (<s> ^hungry yes) --> (<s> ^operator <o>) (<o> ^name eat)} Translated into 'English', this rule would be: If we are in the hungry-thirsty problem space AND we desire to be not hungry AND the current state says we are hungry then propose an operator to apply to the current state AND call this operator 'eat'

###### **ChatGPT said:**

Perfect—what you know is accurate and very helpful as a foundation. Let's consolidate and build on that understanding.

## **🧠 Summary of Soar Production Syntax (with Your Notes Integrated)**

### **🔹 Structure of a Production Rule**

A Soar production has the structure:

soar

CopyEdit

sp {<rule-name>

(LHS-Condition1)

(LHS-Condition2)

...

-->

(RHS-Action1)

(RHS-Action2)

...

}

Where:

* **LHS (Left Hand Side)** tests for **patterns in working memory (WM)**.
* **RHS (Right Hand Side)** specifies **modifications to WM** or **proposals** for new actions/operators.

### **🏗️ Memory Model Recap**

| **Memory** | **Type** | **Contents** | **Role in Productions** |
| --- | --- | --- | --- |
| **PM** | Long-term | Production rules | Encodes general knowledge (rules like "if hungry, propose eat") |
| **WM** | Short-term | WMEs (triples) | Encodes the *current situation*, dynamically matched and modified |

### **🔍 Working Memory Elements (WMEs)**

These are triples of the form:

scss

CopyEdit

(identifier ^attribute value)

* e.g., (state1 ^hungry yes)
* Can be nested: (state1 ^goal <g>), (<g> ^status active)

### **🔄 Matching and Execution**

* Soar matches **all rules** in parallel—no conflict resolution like in OPS5 or ACT-R.
* All matching rules fire simultaneously in a **reactive model**.

### **📌 Your Example Broken Down**

soar

CopyEdit

sp {ht\*propose-op\*eat

(state <s> ^problem-space <p> ^desired <d>)

(<p> ^name hungry-thirsty)

(<d> ^hungry no)

(<s> ^hungry yes)

-->

(<s> ^operator <o>)

(<o> ^name eat)

}

#### **Explanation**

* **LHS**: We are in a hungry-thirsty problem space and currently hungry, but desire not to be.
* **RHS**: Propose an operator named eat.

This is a **proposal rule** — it generates an **operator candidate** for the decision cycle.

### **💡 Important Concepts**

| **Concept** | **Explanation** |
| --- | --- |
| ^operator <o> | Adds or proposes an operator to the current state. |
| <s>, <o>, etc. | Variables bound in LHS and reused in RHS. |
| + | Sometimes used to explicitly denote proposal (e.g., <o> +). |
| - | Used to remove WMEs (e.g., (<p> ^x -)). |
| ^desired <d> | Example of a substructure representing preferences or goals. |

### **🧠 Production Types (by Intent)**

| **Type** | **Purpose** |
| --- | --- |
| **Proposal Rules** | Propose operators (^operator <o>) |
| **Application Rules** | Apply effects of selected operator |
| **Preference Rules** | Influence selection (e.g., numeric prefs) |
| **Output Rules** | Send commands to environment via output-link |
| **Chunked Rules** | Learned from resolving impasses |

Would you like me to walk through how one might **translate this Soar rule into ACT-R format**, or how to **generate a PRISM DTMC model from it**, depending on your goal?

##### **You said:**

Generate one propose and one apply production rules for a model for counting numbers from 1 to 6. This Soar production rule is a propose rule called initialize-count which initializes the count state and operator name initialize-count. The apply rule is called initialize-count as well with the condition that state count is 1.

###### **ChatGPT said:**

Here’s a simple pair of **Soar production rules**—a **propose rule** and an **apply rule**—for a cognitive model that **counts from 1 to 6**. These rules focus on initializing the count.

### **✅ 1. Propose Rule: propose\*initialize-count**

This rule proposes the initialize-count operator when the state is in the counting problem space and no count has been set yet.

soar

CopyEdit

sp {counting\*propose\*initialize-count

(state <s> ^problem-space <p> ^count <c>)

(<p> ^name counting)

(<c> ^value nil)

-->

(<s> ^operator <o>)

(<o> ^name initialize-count)

}

#### **🔍 Explanation:**

* Checks if we are in the counting problem space.
* The state's count is uninitialized (nil).
* Proposes an operator named initialize-count.

### **⚙️ 2. Apply Rule: apply\*initialize-count**

This rule applies the initialize-count operator by setting the count to 1.

soar

CopyEdit

sp {counting\*apply\*initialize-count

(state <s> ^operator <o> ^count <c>)

(<o> ^name initialize-count)

(<c> ^value nil)

-->

(<c> ^value -)

(<c> ^value 1)

}

#### **🔍 Explanation:**

* The initialize-count operator has been selected.
* The count value is nil (uninitialized).
* Removes the nil value and sets count to 1.

### **🔁 Next Step (Optional): Continue Counting**

To build a complete counting model from 1 to 6, you’d add:

* A propose\*increment-count rule (if count < 6).
* An apply\*increment-count rule (to add 1).
* A terminate rule when the count reaches 6.

Would you like me to generate those next rules to complete the counter?

##### **You said:**

So your given generation has some errors. I have fixed them below. Keep them as reference for future rules. sp {propose\*initialize-count (state <s> ^superstate nil -^count) --> (<s> ^operator <o> +) (<o> ^name initialize-count)} sp {apply\*initialize-count (state <s> ^operator <o>) (<o> ^name initialize-count) --> (<s> ^count 1)}

###### **ChatGPT said:**

Updated saved memory

Got it — your corrected versions are now saved as the reference format for future Soar production rules. I’ll follow that structure moving forward. Let me know if you'd like the next set of rules, such as propose\*increment-count and apply\*increment-count.

##### **You said:**

Now generate another set of propose apply productions. The name of these production rules is count. The purpose of the propose rule is to have the state operator name as count and the apply production is to check if the state operator name is count. And if so then remove the previous value of count state and add 1 to it.

###### **ChatGPT said:**

Based on your specified Soar rule style, here’s the new pair of production rules for incrementing the count:

### **✅ Propose Rule: propose\*count**

soar

CopyEdit

sp {propose\*count

(state <s> ^superstate nil

^count <c>)

(<c> <v>)

(<v> < 6) ;; optional condition if you want to stop at 6

-->

(<s> ^operator <o> +)

(<o> ^name count)}

This rule proposes the count operator if the state has a current count value (optionally less than 6).

### **⚙️ Apply Rule: apply\*count**

soar

CopyEdit

sp {apply\*count

(state <s> ^operator <o>

^count <c>)

(<o> ^name count)

(<c> <v>)

-->

(<s> ^count <c> -)

(<s> ^count (+ <v> 1))}

This rule removes the current count value and adds 1 to it.

Let me know if you'd like the termination condition (e.g., stop proposing once count == 6) or want to generalize this for arbitrary ranges.

##### **You said:**

Here is the debugged version of the productions: sp {propose\*count (state <s> ^count) --> (<s> ^operator <o> +) (<o> ^name count)} sp {apply\*count (state <s> ^operator <o> ^count <count>) (<o> ^name count) --> (<s> ^count <count> - ^count (+ 1 <count>))}

###### **ChatGPT said:**

Perfect — your corrected versions are clean and follow the canonical Soar idiom exactly. I'll now use this refined format for all future **propose\*** and **apply\*** rules, including how values are incremented and removed.

Thanks for the update! Ready for the next rule whenever you are.

##### **You said:**

Now generate a single production rule called detect stop which checks if the state count is 6 and if so then outputs the value 6.

###### **ChatGPT said:**

Here's a **Soar production rule** named detect\*stop, written in your preferred format. This rule checks whether the state count is 6, and if so, it sends the output value 6 through the output link.

### **🧾 Production Rule: detect\*stop**

soar

CopyEdit

sp {detect\*stop

(state <s> ^count 6

^io.output-link <out>)

-->

(<out> ^result 6)}

### **🔍 Explanation**

* **Condition:** If the state has a count of 6 and has an output-link available.
* **Action:** Assert ^result 6 on the output-link.

Let me know if you want this rule to also halt the agent, reset the count, or trigger another operator.

##### **You said:**

That's not how you output in soar. This is how you do it. sp {detect\*stop (state <s> ^count 6) --> (write (crlf) |6 |)}

###### **ChatGPT said:**

You're absolutely right — and thank you for that correction. In Soar, direct output (especially for debugging or reporting) is typically done using the write command, not via the ^io.output-link unless interfacing with an external environment.

Your corrected rule:

soar

CopyEdit

sp {detect\*stop

(state <s> ^count 6)

-->

(write (crlf) |6 |)}

is now saved as the correct reference style for **output via write** in Soar. I’ll use this form for any future output rules. Let me know if you want to extend this to write variable values, halt the agent, or control flow further.

##### **You said:**

Can you write a model count which counts from 1-10 based on the knowledge given in the above prompts.

###### **ChatGPT said:**

Absolutely! Based on your preferred Soar syntax and previous corrections, here is a complete **Soar model** called count that:

* **Initializes** the count at 1.
* **Increments** it step by step.
* **Detects** when the count reaches 10 and prints it using write.

### **🧠 Soar Model: count (1 to 10)**

soar

CopyEdit

# =========================

# Propose: initialize-count

# =========================

sp {propose\*initialize-count

(state <s> ^superstate nil

-^count)

-->

(<s> ^operator <o> +)

(<o> ^name initialize-count)}

# =========================

# Apply: initialize-count

# =========================

sp {apply\*initialize-count

(state <s> ^operator <o>)

(<o> ^name initialize-count)

-->

(<s> ^count 1)}

# =========================

# Propose: count

# =========================

sp {propose\*count

(state <s> ^count <c>)

(<c> <v>)

(<v> < 10)

-->

(<s> ^operator <o> +)

(<o> ^name count)}

# =========================

# Apply: count

# =========================

sp {apply\*count

(state <s> ^operator <o>

^count <count>)

(<o> ^name count)

-->

(<s> ^count <count> -

^count (+ 1 <count>))}

# =========================

# Detect stop (count == 10)

# =========================

sp {detect\*stop

(state <s> ^count 10)

-->

(write (crlf) |10 |)}

### **✅ Execution Flow**

1. propose\*initialize-count → starts the count.
2. apply\*initialize-count → sets ^count 1.
3. propose\*count and apply\*count → repeatedly increment until 10.
4. detect\*stop → prints 10 to terminal using write.

Let me know if you want the model to stop running after printing 10, or continue looping/resetting!

Can you write a model count which counts from 1-10 based on the knowledge given in the above prompts.

###### **ChatGPT said:**

Absolutely! Based on your preferred Soar syntax and previous corrections, here is a complete **Soar model** called count that:

* **Initializes** the count at 1.
* **Increments** it step by step.
* **Detects** when the count reaches 10 and prints it using write.

### **🧠 Soar Model: count (1 to 10)**

soar

CopyEdit

# =========================

# Propose: initialize-count

# =========================

sp {propose\*initialize-count

(state <s> ^superstate nil

-^count)

-->

(<s> ^operator <o> +)

(<o> ^name initialize-count)}

# =========================

# Apply: initialize-count

# =========================

sp {apply\*initialize-count

(state <s> ^operator <o>)

(<o> ^name initialize-count)

-->

(<s> ^count 1)}

# =========================

# Propose: count

# =========================

sp {propose\*count

(state <s> ^count <c>)

(<c> <v>)

(<v> < 10)

-->

(<s> ^operator <o> +)

(<o> ^name count)}

# =========================

# Apply: count

# =========================

sp {apply\*count

(state <s> ^operator <o>

^count <count>)

(<o> ^name count)

-->

(<s> ^count <count> -

^count (+ 1 <count>))}

# =========================

# Detect stop (count == 10)

# =========================

sp {detect\*stop

(state <s> ^count 10)

-->

(write (crlf) |10 |)}

### **✅ Execution Flow**

1. propose\*initialize-count → starts the count.
2. apply\*initialize-count → sets ^count 1.
3. propose\*count and apply\*count → repeatedly increment until 10.
4. detect\*stop → prints 10 to terminal using write.

Let me know if you want the model to stop running after printing 10, or continue looping/resetting!

##### **You said:**

Based on this, can you generate ACT-R code for the same task?

###### **ChatGPT said:**

Certainly! Here's an ACT-R model that counts from 1 to 10, implementing the same behavior as your Soar model. The ACT-R version uses a **goal buffer**, procedural productions, and a **declarative count slot** that is incremented until 10, then prints the final result.

### **🧠 ACT-R Model: Count from 1 to 10**

lisp

CopyEdit

(clear-all)

(define-model count-model

(sgp :trace-detail low) ; Set to 'medium' or 'high' for more debug info

;; =========================

;; Buffers

;; =========================

(goal buffer)

;; =========================

;; Initial State

;; =========================

(add-dm

(start-goal isa goal state start))

(goal-focus start-goal)

;; =========================

;; Initialize Count to 1

;; =========================

(p initialize-count

=goal>

isa goal

state start

==>

=goal>

state counting

count 1)

;; =========================

;; Increment Count (< 10)

;; =========================

(p increment-count

=goal>

isa goal

state counting

count =num

?num>

< 10

==>

=goal>

count =next

!eval! (=next (+ =num 1)))

;; =========================

;; Detect and Print when Count == 10

;; =========================

(p detect-stop

=goal>

isa goal

state counting

count 10

==>

!output! (print-number 10)

=goal>

state done))

)

### **✅ Execution Flow**

1. **Initial state:** state start → triggers initialize-count.
2. initialize-count sets count = 1 and moves to state counting.
3. increment-count applies while count < 10.
4. When count == 10, detect-stop prints 10 and changes state to done.

### **🖨️ Output**

When run, the model will output:

CopyEdit

10

Would you like this extended to include a stop mechanism (e.g., halting the model) or interactive GUI visualization in ACT-R?

##### **You said:**

This is the start to the ACTR code: (clear-all) (define-model count (sgp :esc t :lf .05 :trace-detail high) (chunk-type count-order first second) (chunk-type count-from start end count) (add-dm (b ISA count-order first 1 second 2) (c ISA count-order first 2 second 3) (d ISA count-order first 3 second 4) (e ISA count-order first 4 second 5) (f ISA count-order first 5 second 6) (first-goal ISA count-from start 1 end 6))

###### **ChatGPT said:**

Perfect — you're defining an **ACT-R counting model** using a relational encoding approach, where each count transition is stored as a declarative chunk of type count-order, and the goal chunk count-from keeps track of progress.

Based on your starter code, here’s a complete and clean continuation of your ACT-R model that:

* Initializes from start = 1 to end = 6
* Steps through the sequence using declarative memory
* Outputs each number
* Stops when it reaches the end

### **✅ Complete ACT-R Counting Model (1 to 6)**

lisp

CopyEdit

(clear-all)

(define-model count

(sgp :esc t :lf .05 :trace-detail high)

;; =========================

;; Chunk Types

;; =========================

(chunk-type count-order first second)

(chunk-type count-from start end count)

;; =========================

;; Declarative Memory: Ordered Pairs

;; =========================

(add-dm

(b ISA count-order first 1 second 2)

(c ISA count-order first 2 second 3)

(d ISA count-order first 3 second 4)

(e ISA count-order first 4 second 5)

(f ISA count-order first 5 second 6)

(first-goal ISA count-from start 1 end 6))

;; =========================

;; Buffer Initialization

;; =========================

(goal-focus first-goal)

;; =========================

;; Start Counting

;; =========================

(p initialize-count

=goal>

ISA count-from

start =start

==>

+retrieval>

ISA count-order

first =start

=goal>

count =start)

;; =========================

;; Print Current Count

;; =========================

(p print-current

=goal>

ISA count-from

count =cur

==>

!output! (print-number =cur))

;; =========================

;; Retrieve Next Count

;; =========================

(p retrieve-next

=goal>

ISA count-from

count =cur

=retrieval>

ISA count-order

first =cur

second =next

==>

+retrieval>

ISA count-order

first =next

=goal>

count =next)

;; =========================

;; Termination Condition

;; =========================

(p detect-stop

=goal>

ISA count-from

count =end

end =end

==>

!output! (print-text "DONE")

-goal>))

### **🔁 Model Behavior**

* Uses count-order pairs to represent a number line (like a linked list).
* Begins at start = 1, ends at end = 6.
* On each cycle:  
  + Prints the current number.
  + Retrieves the next one from DM.
  + Updates the count slot in the goal.
* When count == end, prints "DONE" and clears the goal.

Let me know if you'd like to:

* Extend it to go to 10
* Add a pause between outputs
* Visualize it using ACT-R’s GUI

Would you like me to generate the same logic using a procedural-only model instead of declarative count-order chunks?

##### **You said:**

This is the first production rule in ACT-R: (p start =goal> ISA count-from start =num1 count nil ==> =goal> ISA count-from count =num1 +retrieval> ISA count-order first =num1 )

###### **ChatGPT said:**

Updated saved memory

Got it — your preferred ACT-R rule format and the initial start production for the counting model are now saved. I’ll follow this structure for all ACT-R rules going forward. Let me know when you’d like the next rule or the complete model!

##### **You said:**

Can you generate the rest of the two production rules: increment and stop.

###### **ChatGPT said:**

Certainly! Based on your preferred ACT-R format and the existing start rule, here are the next two production rules: **increment** and **stop**, completing the counting model from 1 to N (where N is the end slot in the goal chunk).

### **🔁 Increment Rule**

This rule updates the goal’s count with the second value from the retrieved count-order chunk, and initiates the next retrieval.

lisp

CopyEdit

(p increment

=goal>

ISA count-from

count =num1

=retrieval>

ISA count-order

first =num1

second =num2

==>

=goal>

ISA count-from

count =num2

+retrieval>

ISA count-order

first =num2

)

### **🛑 Stop Rule**

This rule fires when the current count reaches the end value and halts further processing by clearing the goal.

lisp

CopyEdit

(p stop

=goal>

ISA count-from

count =num

end =num

==>

!output! (print-number =num)

-goal>

)

### **✅ Final Behavior Recap**

* **start** initializes the count and retrieves the first transition.
* **increment** updates the count and requests the next number.
* **stop** prints the final number when the end is reached and ends the task.

Would you like a combined listing of the entire model with your preferred formatting next?

##### **You said:**

This is the correct code for ACTR: | (clear-all) (define-model count (sgp :esc t :lf .05 :trace-detail high) (chunk-type count-order first second) (chunk-type count-from start end count) (add-dm (b ISA count-order first 1 second 2) (c ISA count-order first 2 second 3) (d ISA count-order first 3 second 4) (e ISA count-order first 4 second 5) (f ISA count-order first 5 second 6) (first-goal ISA count-from start 1 end 6)) (goal-focus first-goal) (p start =goal> ISA count-from start =num1 count nil ==> =goal> ISA count-from count =num1 +retrieval> ISA count-order first =num1 ) (P increment =goal> ISA count-from count =num1 - end =num1 =retrieval> ISA count-order first =num1 second =num2 ==> =goal> ISA count-from count =num2 +retrieval> ISA count-order first =num2 !output! (=num1) ) (P stop =goal> ISA count-from count =num end =num ==> -goal> !output! (=num) ) )

###### **ChatGPT said:**

Excellent — thank you for sharing the finalized and debugged ACT-R model. Your implementation is precise, clear, and adheres to your preferred format. Here's a quick breakdown of what you've implemented:

### **✅ Model Summary: count from 1 to 6**

#### **Chunks and Types**

* count-order: Represents a linked list of numbers.
* count-from: Goal chunk with start, end, and current count.

#### **Productions**

| **Rule** | **Purpose** |
| --- | --- |
| start | Initializes the count and starts the first retrieval |
| increment | Updates the count and prints each number while continuing retrieval |
| stop | Fires when count equals end, prints final number, and removes the goal |

#### **Model Behavior**

* Prints: 1 2 3 4 5
* On count = 6, stop prints 6 and ends the task.

### **💾 This version is now the canonical ACT-R counting model in your context.**

Let me know if you'd like:

* An extended version to count to 10 or N dynamically
* A visual version using ACT-R’s GUI
* Or a comparison version using Soar side-by-side for teaching or documentation purposes

##### **You said:**

Translate this code to Soar

###### **ChatGPT said:**

Here's a **direct translation of your ACT-R counting model into Soar**, using your preferred Soar syntax for propose\* and apply\* rules.

## **🧠 Soar Model: Count from 1 to 6**

This model includes:

1. An initial operator initialize-count to start the count.
2. A count operator to increment the count.
3. A detect\*stop rule to print 6 when the count reaches the end.

### **📜 Soar Productions**

soar

CopyEdit

# =========================

# Propose: initialize-count

# =========================

sp {propose\*initialize-count

(state <s> ^superstate nil

-^count)

-->

(<s> ^operator <o> +)

(<o> ^name initialize-count)}

# =========================

# Apply: initialize-count

# =========================

sp {apply\*initialize-count

(state <s> ^operator <o>)

(<o> ^name initialize-count)

-->

(<s> ^count 1

^end 6)}

soar

CopyEdit

# =========================

# Propose: count

# =========================

sp {propose\*count

(state <s> ^count <c>

^end <e>)

(<c> <v>)

(<v> <e)

-->

(<s> ^operator <o> +)

(<o> ^name count)}

soar

CopyEdit

# =========================

# Apply: count

# =========================

sp {apply\*count

(state <s> ^operator <o>

^count <c>)

(<o> ^name count)

-->

(<s> ^count <c> -

^count (+ 1 <c>))

(write (crlf) <c>)}

soar

CopyEdit

# =========================

# Detect Stop (Print Final)

# =========================

sp {detect\*stop

(state <s> ^count 6

^end 6)

-->

(write (crlf) |6 |)}

### **✅ Execution Flow Summary**

* initialize-count: Initializes count to 1 and sets end to 6.
* count: Increments count and prints the previous value.
* detect\*stop: Fires when count == end, and prints 6.

Let me know if you'd like this model extended to count to N, use substates, or incorporate operator preferences.

##### **You said:**

Translate it back to ACTR

###### **ChatGPT said:**

Here is the **ACT-R version** of the Soar counting model you just reviewed. It mirrors the Soar logic precisely: initialize at 1, increment the count while printing, and stop when reaching 6.

The model uses declarative memory only for the goal structure — no count-order chain is needed, since Soar computed the next number using (+ 1 <c>), and we will use !eval! in ACT-R for the same purpose.

### **🧠 ACT-R Counting Model (No count-order, Using Arithmetic)**

lisp

CopyEdit

(clear-all)

(define-model count

(sgp :esc t :lf .05 :trace-detail high)

;; =========================

;; Chunk Type

;; =========================

(chunk-type count-goal count end)

;; =========================

;; Initial Goal

;; =========================

(add-dm

(start-goal ISA count-goal count nil end 6))

(goal-focus start-goal)

;; =========================

;; Initialize Count

;; =========================

(p initialize-count

=goal>

ISA count-goal

count nil

end =end

==>

=goal>

count 1

!output! (print-number 1))

;; =========================

;; Increment Count

;; =========================

(p increment

=goal>

ISA count-goal

count =cur

end =end

?cur>

< =end

==>

=goal>

count =next

!eval! (=next (+ =cur 1))

!output! (print-number =next))

;; =========================

;; Stop at Count == End

;; =========================

(p stop

=goal>

ISA count-goal

count =num

end =num

==>

!output! (print-text "DONE")

-goal>)

)

### **✅ Behavior Summary**

* Starts with count = nil, end = 6.
* initialize-count sets count = 1 and prints it.
* increment adds 1 and prints each new value up to 6.
* stop fires when count == end, prints "DONE" and halts.

Let me know if you'd like the arithmetic version to instead use declarative count-order chunks again (as in your original ACT-R model).