How to: ACT-R / Building a cognitive model in Jess / Model Tracing

Vincent Aleven

5th Annual PSLC Summer School

Pittsburgh, July 13 - 17, 2009

CTAT Cognitive Tutor Authoring Tools

LearnLab Pittsburgh Science of Learning Center

Overview

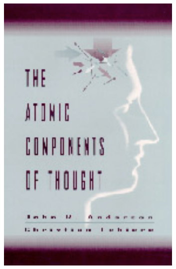
- ACT-R theory
  - Features of production rules and their predictions about learning
- How Production Systems Work
  - A simple example
  - A more complex example: multi-column addition
- Jess Production System Notation
  - Working memory: templates and facts
  - Production rule notation
- Model tracing with Jess
  - Algorithm
  - Special provisions needed when developing a model for model tracing

---

Overview

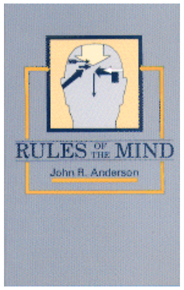
- **ACT-R theory**
  - **Features of production rules and their predictions about learning**
- How Production Systems Work
  - A simple example
  - A more complex example: multi-column addition
- Jess Production System Notation
  - Working memory: templates and facts
  - Production rule notation
- Model tracing with Jess
  - Algorithm
  - Special provisions needed when developing a model for model tracing

---



Anderson, J. R., & Lebiere, C. (1998). *The atomic components of thought*. Mahwah, NJ: Lawrence Erlbaum Associates.

http://act-r.psy.cmu.edu/book/



Anderson, J. R. (1993). *Rules of the mind.* Hillsdale, NJ: Lawrence Erlbaum Associates.

http://act-r.psy.cmu.edu/papers/ROM.html

ACT-R Theory

- Key Claim of Rules of the Mind (Anderson, 1993): "Cognitive skills are realized by production rules"
- What does this mean?
  - What predictions does it make about learning?
  - How does it help explain learning phenomena?

Main claims of ACT-R

1 There are two long-term memory stores, declarative memory and procedural memory.

2 The basic units in declarative memory are chunks.

3 The basic units in procedural memory are production rules.

Declarative-Procedural Distinction

- Declarative knowledge
  - Includes factual knowledge that people can report or describe, but can be non-verbal
  - Stores inputs of perception & includes visual memory
  - Is processed & transformed by procedural knowledge
  - Thus, it can be used *flexibly*, in multiple ways
- Procedural knowledge
  - Is only manifest in people's behavior, not open to inspection, cannot be directly verbalized
  - Is processed & transformed by fixed processes of the cognitive architecture
  - It is more specialized & *efficient*

Intuition for difference between declarative & procedural rules

- Although the rules for writing music (such as allowable chord structures and sequences) were often changed after a major composer had become a great influence, the actual rules by which composers shaped their compositions were often only known to later followers. When they first used them the composer was not consciously restricting himself/herself to the rules, but was rather using them subconsciously, leaving the collecting of the rules to later followers.

Production Rules Describe How People *Use* Declarative Rules in their Thinking

Declarative rule:

Side-side-side theorem

IF the 3 corresponding sides of two triangles are congruent (≅)
THEN the triangles are ≅

Production rules describe thinking patterns:

Special condition to aid search

IF two triangles *share a side* AND the other 2 corresponding sides are ≅
THEN the triangles are congruent (≅)

Using rule backward

IF *goal*: prove triangles ≅ AND 2 sets of corresponding sides are ≅
THEN *subgoal*: prove 3rd set of sides ≅

Using rule heuristically

IF two triangles look ≅
THEN try to prove any of the corresponding sides & angles ≅

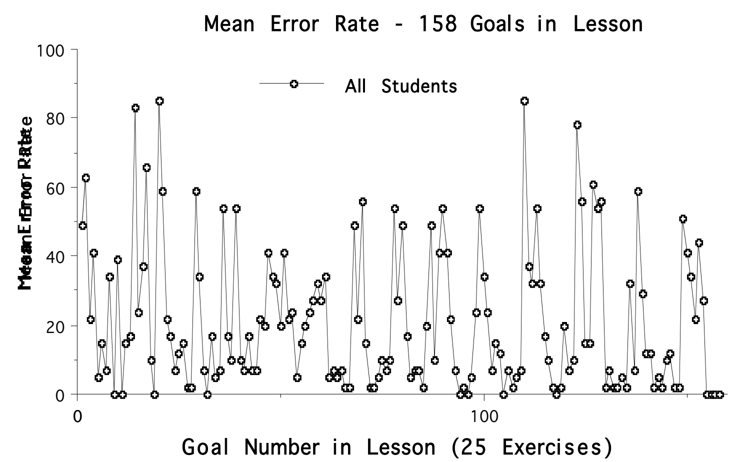
4 Critical Features of Production Rules

- Modular
  - Performance knowledge is learned in "pieces"
- Goal & context sensitive
  - Performance knowledge is tied to particular goals & contexts by the "if-part"
- Abstract
  - Productions apply in multiple situations
- Condition-Action Asymmetry
  - Productions work in one direction

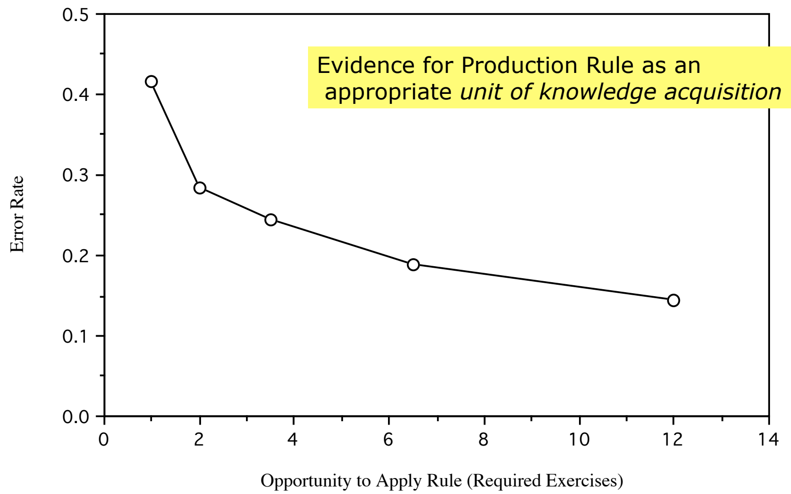
Features 1 & 2 of ACT-R Production Rules

1. Modularity
   - production rules are the units by which a complex skill is acquired
   - empirical evidence: data from the Lisp tutor
2. Abstract character
   - each production rule covers a range of situations, not a single situation
   - variables in the left-hand side of the rule can match different working memory elements

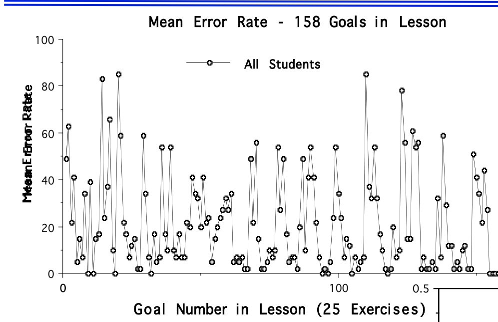
Student Performance As They Practice with the LISP Tutor



Production Rule Analysis



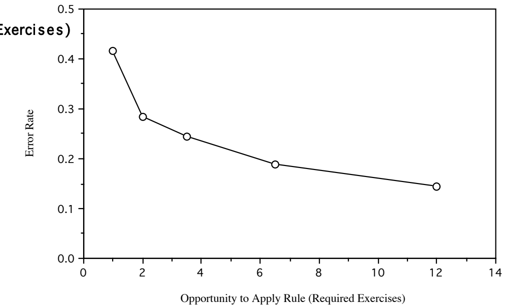
Production Rule Analysis "Cleans Up"



A surface level model does not explain/clarify learning process. Production rule model does.

Learning?

Yes! At the production rule level.



Features 3 & 4 of ACT-R Production Rules

3. Goal structuring
   - productions often include goals among their conditions - a new production rule must be learned when the same action is done *for a different purpose*
   - abstract character means that productions capture a range of generalization, goal structuring means that the *range is restricted to specific goals*
4. Condition-action asymmetry
   - For example, skill at writing Lisp code does not transfer (fully) to skill at evaluating Lisp code.

Production rules have limited generality -- depending on purpose & context of acquisition

Overly general

IF "Num1 + Num2" appears in an expression
THEN
replace it with the sum

Leads to order of operations error:
"x * 3 + 4" is rewritten as "x * 7"

Overly specific

IF "ax + bx" appears in an expression and c = a + b
THEN
replace it with "cx"

Works for "2x + 3x" but not for "x + 3x"

Not explicitly taught

IF you want to find Unknown and the final result is Known-Result and the last step was to apply Last-Op to Last-Num,
THEN
Work backwards by inverting Last-Op and applying it to Known-Result and Last-Num

In "3x + 48 = 63":
63
- 48
-----
15 / 3 = 5 (no use of equations!)

Production Rule Asymmetry Example

Declarative rule:

Side-side-side theorem

IF the 3 corresponding sides of two triangles are congruent (≅)

THEN

the triangles are ≅

Forward use of declarative rule

Backward uses of declarative rule

Productions are learned independently, so a student might be only able to use a rule in the forward direction.

Production rules describe thinking patterns:

Special condition to aid search

IF two triangles *share a side* AND
the other 2 corresponding sides are ≅

THEN the triangles are congruent (≅)

Using rule backward

IF *goal*: prove triangles ≅ AND
2 sets of corresponding sides are ≅

THEN *subgoal*: prove 3rd set of sides ≅

Using rule heuristically

IF two triangles look ≅

THEN try to prove any of the corresponding sides & angles ≅

The chunk in declarative memory

- Modular and of limited size
  -> limits how much new info can be processed
- Configural & hierarchical structure
  -> different parts of have different roles
  -> chunks can have subchunks
  - A fraction addition problem contains fractions, fractions contain a numerator & denominator
- Goal-independent & symmetric
  - Rules can be represented as declarative chunks
  - You can "think of" declarative rules but only "think with" procedural rules

Declarative Knowledge Terms

- Declarative Knowledge
  - Is the "Working Memory" of a production system
- A "chunk" is an element of declarative knowledge
  - Type indicates the "slots" or "attributes"
  - In Jess, the chunks are called "facts" and the chunk types are called "templates"

Summary

- Features of cognition explained by ACT-R production rules:
  - Procedural knowledge:
    - modular, limited generality, goal structured, asymmetric
  - Declarative knowledge:
    - flexible, verbal or visual, less efficient

Multiple Uses of Cognitive Model

- Summarizes results of analysis of data on student thinking
- Is the "intelligence" in the tutor
- Most importantly, provides guidance for all aspects of tutor development
  - Interface, tutorial assistance, problem selection and curriculum sequencing

Overview

- ACT-R theory
  - Features of production rules and their predictions about learning
- **How Production Systems Work**
  - **A simple example**
  - A more complex example: multi-column addition
- Jess Production System Notation
  - Working memory: templates and facts
  - Production rule notation
- Model tracing with Jess
  - Algorithm
  - Special provisions needed when developing a model for model tracing

Components of a production rule system

- Working memory -- the database
- Production rule memory
- Interpreter that repeats the following cycle:
  1. Match
     - Match "if-parts" of productions with working memory
     - Collect all applicable production rules
  2. Conflict resolution
     - Select one of these productions to "fire"
  3. Act
     - "Fire" production by making changes to working memory that are indicated in "then-part"

An example production system

- You want a program that can answer questions and make inferences about food items
- Like:
  - What is purple and perishable?
  - What is packed in small containers and gives you a buzz?
  - What is green and weighs 15 lbs?

A simple production rule system making inferences about food

WORKING MEMORY (WM)

Initially WM = (green, weighs-15-lbs)

RULE MEMORY

P1. **IF** green **THEN** produce
P2. **IF** packed-in-small-container **THEN** delicacy
P3. **IF** refrigerated **OR** produce **THEN** perishable
P4. **IF** weighs-15-lbs **AND** inexpensive **AND NOT** perishable **THEN** staple
P5. **IF** perishable **AND** weighs-15-lbs **THEN** turkey
P6. **IF** weighs-15-lbs **AND** produce **THEN** watermelon

INTERPRETER

1. Find all productions whose condition parts are true
2. Deactivate productions that would add a duplicate symbol
3. Execute the lowest numbered production (or quit)
4. Repeat until there is no rule to execute

Adapted from the Handbook of AI, Vol I, pp. 191

First cycle of execution

WORKING MEMORY

WM = (green, weighs-15-lbs)

CYCLE 1

1. Productions whose condition parts are true: ***P1***
2. No production would add duplicate symbol
3. Execute ***P1***.
   This gives: WM = (***produce,*** green, weighs-15-lbs)

RULE MEMORY

P1. **IF** green **THEN** produce
P2. **IF** packed-in-small-container **THEN** delicacy
P3. **IF** refrigerated **OR** produce **THEN** perishable
P4. **IF** weighs-15-lbs **AND** inexpensive **AND NOT** perishable **THEN** staple
P5. **IF** perishable **AND** weighs-15-lbs **THEN** turkey
P6. **IF** weighs-15-lbs **AND** produce **THEN** watermelon

INTERPRETER

1. Find all productions whose condition parts are true
2. Deactivate productions that would add a duplicate symbol
3. Execute the lowest numbered production (or quit)
4. Repeat

Adapted from the Handbook of AI, Vol I, pp. 191

Do this yourself before reading on!

- Hand simulate the execution of the production rule model.
- For each cycle, write down the following information:
  Activate rules:
  Deactivate rules:
  Execute rule:
  WM = (….)
- What is in working memory when the production rule model finishes?
- Are there any mistakes in the production rules?

Cycle 2

WORKING MEMORY

WM = (produce, green, weighs-15-lbs)

CYCLE 2

1. Productions whose condition parts are true: ***P1, P3, P6***
2. Production P1 would add duplicate symbol, so ***deactivate P1***
3. Execute ***P3*** because it is the lowest numbered production.
   This gives: WM = (***perishable,*** produce, green, weighs-15-lbs)

RULE MEMORY

P1. **IF** green **THEN** produce
P2. **IF** packed-in-small-container **THEN** delicacy
P3. **IF** refrigerated **OR** produce **THEN** perishable
P4. **IF** weighs-15-lbs **AND** inexpensive **AND NOT** perishable **THEN** staple
P5. **IF** perishable **AND** weighs-15-lbs **THEN** turkey
P6. **IF** weighs-15-lbs **AND** produce **THEN** watermelon

INTERPRETER

1. Find all productions whose condition parts are true
2. Deactivate productions that would add a duplicate symbol
3. Execute the lowest numbered production (or quit)
4. Repeat

Adapted from the Handbook of AI, Vol I, pp. 191

Cycle 3

WORKING MEMORY
WM = (perishable, produce, green, weighs-15-lbs)

CYCLE 3
1. Productions whose condition parts are true: ***P1, P3, P5, P6***
2. Productions P1 and P3 would add duplicate symbol, so ***deactivate P1 and P3***
3. Execute ***P5***. ***Incorrect rule!?***
   This gives: WM = (***turkey***, perishable, produce, green, weighs-15-lbs)

RULE MEMORY
P1. **IF** green **THEN** produce
P2. **IF** packed-in-small-container
 THEN delicacy
P3. **IF** refrigerated **OR** produce
 THEN perishable
P4. **IF** weighs-15-lbs **AND** inexpensive **AND**
 NOT perishable
 THEN staple
P5. **IF** perishable **AND** weighs-15-lbs **THEN**
 turkey
P6. **IF** weighs-15-lbs **AND** produce **THEN**
 watermelon

INTERPRETER
1. Find all productions whose condition parts are true
2. Deactivate productions that would add a duplicate symbol
3. Execute the lowest numbered production (or quit)
4. Repeat

Adapted from the Handbook of AI, Vol I, pp. 191

Cycle 4

WORKING MEMORY
WM = (turkey, perishable, produce, green, weighs-15-lbs)

CYCLE 4
1. Productions whose condition parts are true: ***P1, P3, P5, P6***
2. Productions ***P1, P3, P5*** would add duplicate symbol, so deactivate them
3. Execute ***P6***. This gives: WM = (***watermelon***, turkey, perishable, produce, green, weighs-15-lbs)

RULE MEMORY
P1. **IF** green **THEN** produce
P2. **IF** packed-in-small-container
 THEN delicacy
P3. **IF** refrigerated **OR** produce
 THEN perishable
P4. **IF** weighs-15-lbs **AND** inexpensive **AND**
 NOT perishable
 THEN staple
P5. **IF** perishable **AND** weighs-15-lbs **THEN**
 turkey
P6. **IF** weighs-15-lbs **AND** produce **THEN**
 watermelon

INTERPRETER
1. Find all productions whose condition parts are true
2. Deactivate productions that would add a duplicate symbol
3. Execute the lowest numbered production (or quit)
4. Repeat

Adapted from the Handbook of AI, Vol I, pp. 191

Cycle 5

WORKING MEMORY
WM = (watermelon, turkey, perishable, produce, green, weighs-15-lbs)

CYCLE 5
1. Productions whose condition parts are true: ***P1, P3, P5, P6***
2. Productions ***P1, P3, P5, P6*** would add duplicate symbol, so ***deactivate them***
3. ***Quit.***

RULE MEMORY
P1. **IF** green **THEN** produce
P2. **IF** packed-in-small-container
 THEN delicacy
P3. **IF** refrigerated **OR** produce
 THEN perishable
P4. **IF** weighs-15-lbs **AND** inexpensive **AND**
 NOT perishable
 THEN staple
P5. **IF** perishable **AND** weighs-15-lbs **THEN**
 turkey
P6. **IF** weighs-15-lbs **AND** produce **THEN**
 watermelon

INTERPRETER
1. Find all productions whose condition parts are true
2. Deactivate productions that would add a duplicate symbol
3. Execute the lowest numbered production (or quit)
4. Repeat

Adapted from the Handbook of AI, Vol I, pp. 191

Cycles 2-5

WM = (produce, green, weighs-15-lbs)
CYCLE 2
1. Activate: P1, P3, P6
2. Deactivate P1
3. Execute ***P3***. WM= (***perishable,*** produce, green, weighs-15-lbs)

CYCLE 3 ***Is this a bug in the rules?***
1. Activate: P1, P3, P5, P6
2. Deactivate: P1 and P3
3. Execute ***P5***. WM= (***turkey***, perishable, produce, green, weighs-15-lbs)

CYCLE 4
1. Activate: P1, P3, P5, P6
2. Deactivate: P1, P3, P5
3. Execute ***P6***. WM = (***watermelon***, turkey, perishable, produce, green, weighs-15-lbs)

CYCLE 5
1. Activate: P1, P3, P5, P6
2. Deactivate: P1, P3, P5, P6.
3. ***Quit.***

RULE MEMORY
P1. **IF** green **THEN** produce
P2. **IF** packed-in-small-container
 THEN delicacy
P3. **IF** refrigerated **OR** produce
 THEN perishable
P4. **IF** weighs-15-lbs **AND** inexpensive **AND NOT** perishable
 THEN staple
P5. **IF** perishable **AND** weighs-15-lbs
 THEN turkey
P6. **IF** weighs-15-lbs **AND** produce **THEN** watermelon

How ACT-R & Jess production systems are more complex

- Watermelon is simple example:
  - *Working memory elements*: a single word
  - *Production rules*: no variables in if-part
  - *Interpreter*: conflict resolution selects lowest numbered unused production
- In contrast, in ACT-R and Jess:
  - *Working memory elements*: database-like record structures with attributes and values
  - *Production rules*: includes variables & patterns
  - *Interpreter*: match must deal with variables and patterns, conflict resolution does *not* use rule order

Overview

- ACT-R theory
  - Features of production rules and their predictions about learning
- How Production Systems Work
  - A simple example
  - **A more complex example: multi-column addition**
- Jess Production System Notation
  - Working memory: templates and facts
  - Production rule notation
- Model tracing with Jess
  - Algorithm
  - Special provisions needed when developing a model for model tracing

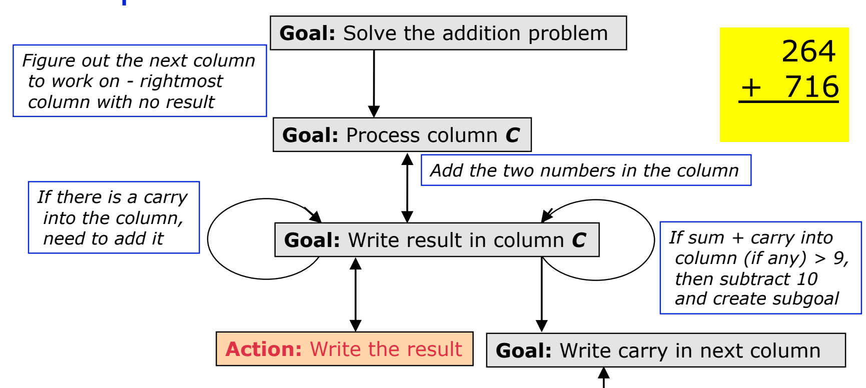
A second production rule model example

- Think about how you would write production rules to do multi-column addition?

$$\begin{array}{r} 264 \\ +\ 716 \\ \hline \end{array}$$

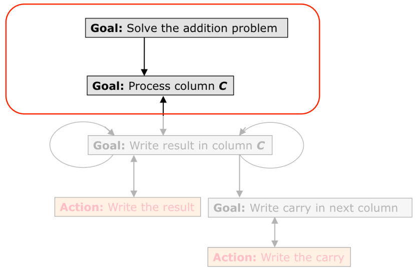
- What if-then rules would you write to perform this task in a step-by-step fashion?

Production rules set new goals and perform actions



Adapted from Anderson, J. R. 1993. *Rules of the Mind*. Hillsdale, NJ: LEA.

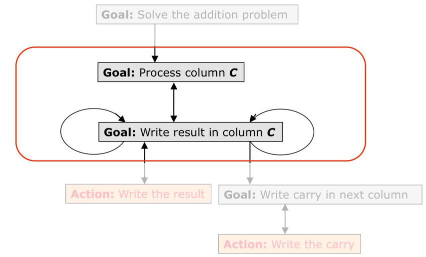
Production rules are tied to particular goals and particular context



FOCUS-ON-FIRST-COLUMN
IF The goal is to do an addition problem
And there is no pending subgoal
And there is no result yet in the rightmost column of the problem
THEN Set a subgoal to process the rightmost column

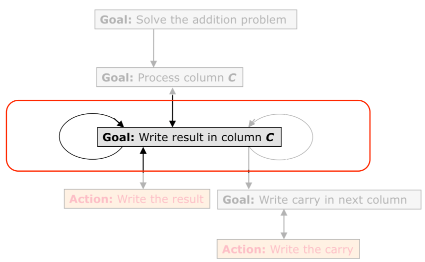
FOCUS-ON-NEXT-COLUMN
IF The goal is to do an addition problem
And there is no pending subgoal
And ***C*** is a column with numbers to add and no result
And the column to the right of ***C*** has a result
THEN Set a subgoal to process column ***C***

Production rules are tied to particular goals and particular context



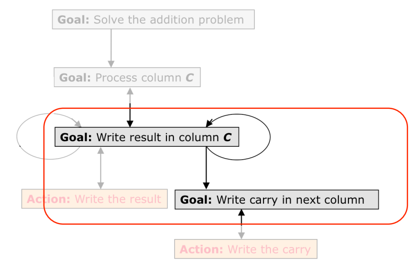
ADD-ADDENDS
IF There is a goal to process column ***C***
And there is no subgoal to write a result in column ***C***
THEN Set ***Sum*** to the sum of the addends in column ***C***
And set a subgoal to write ***Sum*** as the result in column ***C***
And remove the goal to process column ***C***

Production rules are tied to particular goals and particular context



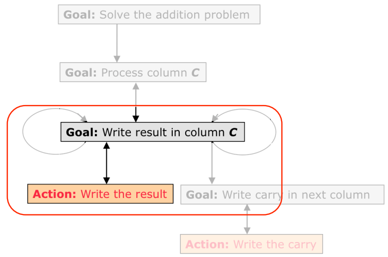
ADD-CARRY
IF There is a goal to write ***Sum*** as the result in column ***C***
And there is a carry into column ***C***
And the carry has not been added to ***Sum***
THEN Change the goal to write ***Sum*** + 1 as the result
And mark the carry as added

Production rules are tied to particular goals and particular context



MUST-CARRY
IF There is a goal to write ***Sum*** as the result in column ***C***
And the carry into column ***C*** (if any) has been added to ***Sum***
And ***Sum*** > 9
And ***Next*** is the column to the left of ***C***
THEN Change the goal to write ***Sum***-10 as the result in ***C***
Set a subgoal to write 1 as a carry in column ***Next***

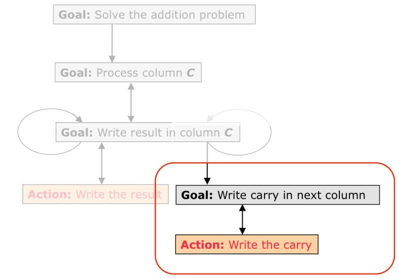
Production rules are tied to particular goals and particular context



WRITE-SUM

IF There is a goal to write ***Sum*** as the result in column ***C***
And ***Sum*** < 10
And the carry into column ***C*** (if any) has been added
THEN Write ***Sum*** as the result in column ***C***
And remove the goal

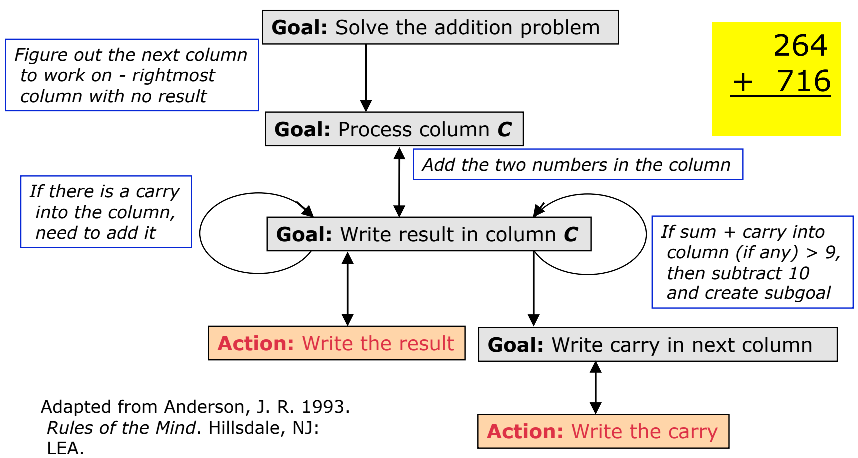
Production rules are tied to particular goals and particular context



WRITE-CARRY

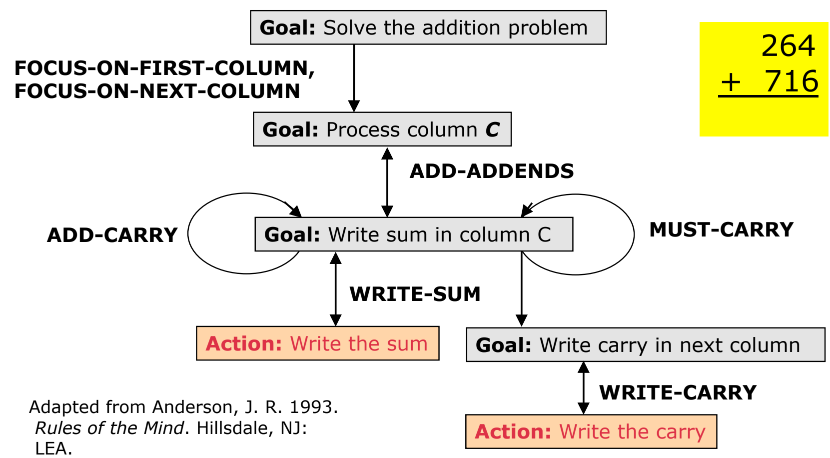
IF There is a goal to write a carry in column ***C***
THEN Write the carry in column ***C***
And remove the goal

Production rules set new goals and perform actions



Adapted from Anderson, J. R. 1993. *Rules of the Mind*. Hillsdale, NJ: LEA.

Production rules set new goals and perform actions



Adapted from Anderson, J. R. 1993. *Rules of the Mind*. Hillsdale, NJ: LEA.

Production rule model for addition

FOCUS-ON-FIRST-COLUMN
IF The goal is to do an addition problem
And there is no pending subgoal
And there is no result yet in the rightmost column of the problem
THEN Set a subgoal to process the rightmost column

FOCUS-ON-NEXT-COLUMN
IF The goal is to do an addition problem
And there is no pending subgoal
And ***C*** is a column with numbers to add and no result
And the column to the right of ***C*** has a result
THEN Set a subgoal to process column ***C***

ADD-ADDENDS
IF There is a goal to process column ***C***
And there is no subgoal to write a result in column ***C***
THEN Set ***Sum*** to the sum of the addends in column ***C***
And set a subgoal to write ***Sum*** as the result in column ***C***
And remove the goal to process column ***C***

ADD-CARRY
IF There is a goal to write ***Sum*** as the result in column ***C***
And there is a carry into column ***C***
And the carry has not been added to ***Sum***
THEN Change the goal to write ***Sum*** + 1 as the result
And mark the carry as added

MUST-CARRY
IF There is a goal to write ***Sum*** as the result in column ***C***
And the carry into column ***C*** (if any) has been added to ***Sum***
And ***Sum*** > 9
And ***Next*** is the column to the left of ***C***
THEN Change the goal to write ***Sum***-10 as the result in ***C***
Set a subgoal to write 1 as a carry in column ***Next***

WRITE-SUM
IF There is a goal to write ***Sum*** as the result in column ***C***
And ***Sum*** < 10
And the carry into column ***C*** (if any) has been added
THEN Write ***Sum*** as the result in column ***C***
And remove the goal

WRITE-CARRY
IF There is a goal to write a carry in column ***C***
THEN Write the carry in column ***C***
And remove the goal

DONE
IF The goal is to do an addition problem
And there is no incomplete subgoal to work on
And there is no column left with numbers to add (or a carry) and no result
THEN Mark the problem as done

A Trace of Production Rule Firings

column3 column2 column1

```
  264
+ 716
    0
```

Subgoals: Write carry in column2

Step 1

1. FOCUS-ON-FIRST-COLUMN
   → **Goal:** Process column1
2. ADD-ADDENDS
   C = column1
   Sum = 10
   ← **Goal:** Process column1
   → **Goal:** Write 10 as result in column1
3. MUST-CARRY
   C = column1
   Sum = 10
   Next = column2
   → **Goal:** Write carry in column2
   → **Goal:** Write 0 as result in column1
4. WRITE-SUM
   C = column1
   Sum = 0
   Action: Write 0 as result in column1
   ← **Goal:** Write 0 as result in column1

Q: *Could the carry have been written first?*
A: *Yes, the condition of WRITE-CARRY holds after step 1.3. The model is flexible w.r.t. the order of writing the carry and writing the result.*

Step 2

5. WRITE-CARRY
   C = column2
   Action: Write carry in column2
   ← **Goal:** Write carry in column2

Q: *Could we have moved on without writing the carry?*
A: *No, FOCUS-ON-NEXT-COLUMN can fire only if there is no pending goal. The model does NOT allow implicit carrying.*

A Trace of Production Rule Firings (ctnd.)

```
    1
  264
+ 716
  980
```

No pending goal

DONE

Step 3

6. FOCUS-ON-NEXT-COLUMN
   C = column2
   → **Goal:** Process column2
7. ADD-ADDENDS
   C = column2
   Sum = 7
   ← **Goal:** Process column2
   → **Goal:** Write 7 as result in column2
8. ADD-CARRY
   C = column2
   Sum = 7
   → **Goal:** Write 8 as result in column2
9. WRITE-SUM
   C = column2
   Sum = 8
   Action: Write 8 as result in column2
   ← **Goal:** Write 8 as result in column2

Q: *Good thing that WRITE-SUM did not fire instead after step 3.7. Why didn't it?*
A: *WRITE-SUM has condition that carry into the column must have been added.*

Step 4

10. FOCUS-ON-NEXT-COLUMN
    C = column3
    → **Goal:** Process column3
11. ADD-ADDENDS
    C = column3
    Sum = 9
    ← **Goal:** Process column3
    → **Goal:** Write 9 as result in column3
12. WRITE-SUM
    C = column3
    Sum = 9
    Action: Write 9 as result in column3
    ← **Goal:** Write 9 as result in column3

Step 5

13. DONE

How could you model students who don't carry?

- Instead of doing the addition correctly:

```
    1
  264
+ 716
  980
```

- Can you model a student who writes:

```
  264
+ 716
  970
```

- How can you change the production rule model?

Overview

- ACT-R theory
  - Features of production rules and their predictions about learning
- How Production Systems Work
  - A simple example
  - A more complex example: multi-column addition
- **Jess Production System Notation**
  - **Working memory: templates and facts**
  - **Production rule notation**
- Model tracing with Jess
  - Algorithm
  - Special provisions needed when developing a model for model tracing

Implementing a production rule model in Jess

- Simple example: a model for single-column addition without carrying!

4 + 3	→	4 + 3 7	→	4 + 3 7 Done

- How would you define:
  - Working memory representation for the problem states
  - Production rules that transform working memory

Design and implement working memory representation

A *template* defines a type of fact and the slots that belong to the type:

```
(deftemplate 1column-addition-problem
   (slot name)
   (slot first-addend)
   (slot second-addend)
   (slot result)
   (slot done))
```

CTAT will do this for you!

Asserting a fact puts it in working memory

```
(assert (1column-addition-problem
   (name add4+3)
   (first-addend 4)
   (second-addend 3)))
```


Working memory representing start state

```
Jess> (facts)
f-0   (initial-fact)
f-1   (1column-addition-problem
       (name add4+3)
       (first-addend 4)
       (second-addend 3)
       (result nil)
       (done nil))
For a total of 2 facts.
```

Somewhat unusual that there is only a single fact in working memory (WM). Typically, WM contains multiple facts.

→ We will focus on *unordered facts* only.

Plan production rule model

Problem states

4 + 3	→	4 + 3 7	→	4 + 3 7 Done

Planned production rules for each step

Production rule: ADD

Production rule: DONE

Planned working memory representation for each state

(single-column-addition -problem (name add4+3) (first-addend 4) (second-addend 3) (result nil) (done nil))	(single-column-addition -problem (name add4+3) (first-addend 4) (second-addend 3) **(result 7)** (done nil))	(single-column-addition -problem (name add4+3) (first-addend 4) (second-addend 3) (result 7) **(done True)**)

Jess production rule for first step

4 + 3	→	4 + 3 7	→	4 + 3 7 Done

English

ADD
IF
The goal is to do ***?problem***, a single-column addition problem
And no result has been found yet
And the first addend is ***?num1***
And the second added is ***?num2***
THEN
Set ***?sum*** to the sum of ***?num1*** and ***?num2***
Write ***?sum*** as the result

Jess

```
(defrule add
  ?problem <-
     (1column-addition-problem
       (result nil)
       (first-addend ?num1)
       (second-addend ?num2))
=>
  (bind ?sum (+ ?num1 ?num2))
  (modify ?problem (result ?sum)))
```


Jess production rule notation

Name of the rule

Variables ("bound" to values through matching)

If-part Specifies conditions

Pattern Specifies constraints on slot values

```
(defrule add
  ?problem <-
    (1column-addition-problem
      (result nil)
      (first-addend ?num1)
      (second-addend ?num2))
=>
  (bind ?sum (+ ?num1 ?num2))
  (modify ?problem (result ?sum))
)
```

Then-part Specifies actions

Function calls Perform computations, change facts in working memory

Matching a production rule against working memory -- Find values for each variable

Working Memory

```
f-1  (1column-addition-problem
       (name add4+3)
       (first-addend 4)
       (second-addend 3)
       (result nil)
       (done nil))
```

Match!

Match!

Production Rule

```
(defrule add
  ?problem <- (1column-addition-problem
                (result nil)
                (first-addend ?num1)
                (second-addend ?num2))
=>
  (bind ?sum (+ ?num1 ?num2))
  (modify ?problem (result ?sum)))
```

Find value for each variable

Variable	**Value**
?num1	4
?num2	3
?problem	<Fact-1>
?sum	7

What changes are made to working memory?

```
(1column-addition-problem
    (name add4+3)
    (first-addend 4)
    (second-addend 3)
    (result 7)
    (done nil))
```

More Jess production rule notation

4 + 3	→	4 + 3 7	→	4 + 3 7 Done

English

DONE
IF
The goal is to do a single-column addition problem
And the result has been written
And the problem has not been marked as done yet
THEN
Mark the problem as done

Jess

```
(defrule done
  ?problem <- (1column-addition-problem
                   (result ~nil)
                   (done nil))
=>
  (modify ?problem (done True)))
```

"~" means "not".
A constraint preceded by a tilde is satisfied exactly when the constraint without the tilde would not have been.

Matching the second production rule against working memory

Working Memory

```
f-1  (1column-addition-problem
        (name add4+3)
        (first-addend 4)
        (second-addend 3)
        (result 7)
        (done nil))
```

Match!
Match!
Match!

Production Rule

```
(defrule done
  ?problem <-
    (1column-addition-problem
      (result ~nil)
      (done nil))
=>
  (modify ?problem (done True)))
```

Find value for each variable

Variable	Value
?problem	<Fact-1>

What changes are made to working memory?

```
(1column-addition-problem
     (name add4+3)
     (first-addend 4)
     (second-addend 3)
     (result 7)
     (done True))
```


Rules model problem-solving steps as planned!

4 + 3	→	4 + 3 7	→	4 + 3 7 Done

Production rule: ADD

Production rule: DONE

(single-column-addition -problem	→	(single-column-addition -problem	→	(single-column-addition -problem
(name add4+3) (first-addend 4) (second-addend 3) (result nil) (done nil))		(name add4+3) (first-addend 4) (second-addend 3) **(result 7)** (done nil))		(name add4+3) (first-addend 4) (second-addend 3) (result 7) **(done True)**)

Why didn't the Done rule match in the initial state?

Working Memory

```
f-1  (1column-addition-problem
        (name add4+3)
        (first-addend 4)
        (second-addend 3)
        (result nil)
        (done nil))
```

No Match!

Production Rule

```
(defrule done
  ?problem <-
    (1column-addition-problem
      (result ~nil)
      (done nil))
=>
  (modify ?problem (done True)))
```

Summary—Jess production rule notation

- Working memory is a collection of facts

```
f-1   (1column-addition-problem
          (name add4+3)
          (first-addend 4)
          (second-addend 3)
          (result 7)
          (done nil))
```

- A *template* defines a type of fact and the slots of the type:

```
(deftemplate 1column-addition-problem
   (slot name)
   (slot first-addend)
   (slot second-addend)
   (slot result)
   (slot done))
```

- IF-part of production rules: patterns matched to working memory

```
?problem <- (1column-addition-problem
                (result nil)
                (first-addend ?num1)
                (second-addend ?num2))
```

- THEN-part: computations and changes to working memory

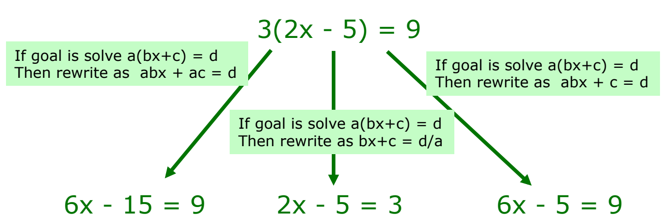
```
(bind ?sum (+ ?num1 ?num2))
(modify ?problem (result ?sum)))
```


Overview

- ACT-R theory
  - Features of production rules and their predictions about learning
- How Production Systems Work
  - A simple example
  - A more complex example: multi-column addition
- Jess Production System Notation
  - Working memory: templates and facts
  - Production rule notation
- **Model tracing with Jess**
  - **Algorithm**
  - **Special provisions needed when developing a model for model tracing**

Cognitive Tutor Technology:
Use ACT-R theory to individualize instruction

- Cognitive Model: A system that can solve problems in the various ways students can



- Model Tracing: Follows student through their individual approach to a problem -> context-sensitive instruction

Model tracing algorithm (main idea)

After a student action:

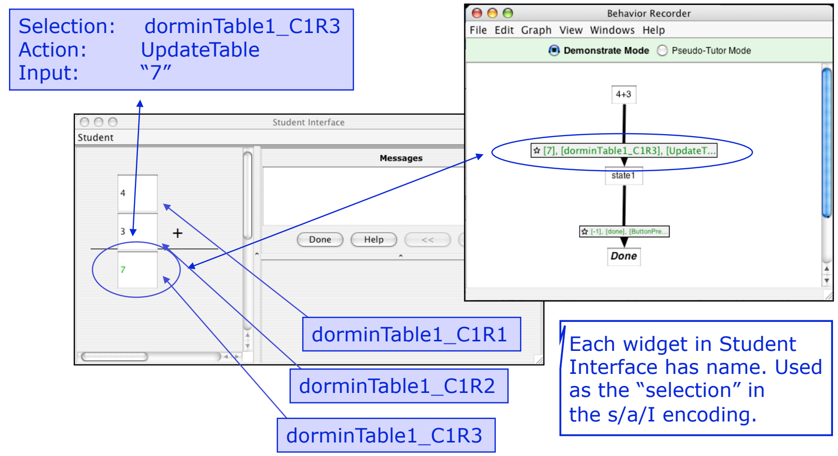
1. Use model to figure out all correct next steps
2. If student took one of these steps, then good!
3. Otherwise, error.

Model tracing algorithm (simplified)

After a student action:

1. Use model to figure out all correct next steps: Use production rule model in "exploratory mode" to generate all sequences of rule firings that produce an "observable action"
   (changes to working memory are undone)
2. If student took one of these steps, then good! If student action is among the actions generated by the model, provide positive feedback
   and update working memory by firing the rule activations that produce the observable actions
   (so working memory and interface stay in sync)
3. Otherwise, error. Provide negative feedback
   (and leave working memory unchanged)

Step 2: comparing student actions against model: actions are encoded as selection/action/input triples.



Production rule author must indicate which RHS actions correspond to observable actions.

```
(defrule add
  ?problem <- (1column-addition-problem
                        (result nil)
                        (first-addend ?num1)
                        (second-addend ?num2))
=>
    (bind ?sum (+ ?num1 ?num2))
    (predict-observable-action
         dorminTable1_C1R3          ; selection
         UpdateTable                ; action
         ?sum)                      ; input
    (modify ?problem (result ?sum))
  )
```

On the RHS, ***observable actions*** simulated by the model (i.e., actions that are "visible" in the interface) must be communicated to the model-tracing algorithm by means of a call to function predict-observable-action. This information enables the model-tracing algorithm to test what the student actually did against the actions predicted by the model.

Best to place the call to predict-observable-action before any actions that modify working memory (more efficient).

Cognitive model for addition: An observable action may involve multiple thinking steps

FOCUS-ON-FIRST-COLUMN
IF The goal is to do an addition problem
And there is no pending subgoal
And there is no result yet in the rightmost column of the problem
THEN Set a subgoal to process the rightmost column

FOCUS-ON-NEXT-COLUMN
IF The goal is to do an addition problem
And there is no pending subgoal
And ***C*** is the rightmost column with numbers to add and no result
THEN Set a subgoal to process column ***C***

ADD-ADDENDS
IF There is a goal to process column ***C***
THEN Set ***Sum*** to the sum of the addends in column ***C***
And set a subgoal to write ***Sum*** as the result in column ***C***
And remove the goal to process column ***C***

ADD-CARRY
IF There is a goal to write ***Sum*** as the result in column ***C***
And there is a carry into column ***C***
And the carry has not been added to ***Sum***
THEN Change the goal to write ***Sum*** + 1 as the result
And mark the carry as added

MUST-CARRY
IF There is a goal to write ***Sum*** as the result in column ***C***
And the carry into column ***C*** (if any) has been added to ***Sum***
And ***Sum*** > 9
And ***Next*** is the column to the left of ***C***
THEN Change the goal to write ***Sum***-10 as the result in ***C***
Set a subgoal to write 1 as a carry in column ***Next***

WRITE-SUM
IF There is a goal to write ***Sum*** as the result in column ***C***
And ***Sum*** < 10
And the carry into column ***C*** (if any) has been added
THEN Write ***Sum*** as the result in column ***C***
And remove the goal

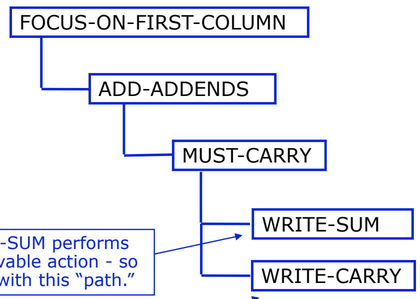
WRITE-CARRY
IF There is a goal to write a carry in column ***C***
THEN Write the carry in column ***C***
And remove the goal

DONE
IF The goal is to do an addition problem
And there is no incomplete subgoal to work on
And there is no column left with numbers to add (or a carry) and no result
THEN Mark the problem as done

Conflict Tree: shows "paths" of matching rules

1. FOCUS-ON-FIRST-COLUMN
   ➔ **Goal:** Process column1
2. ADD-ADDENDS
   C = column1
   Sum = 10
   ← **Goal:** Process column1
   ➔ **Goal:** Write 10 as result in column1
3. MUST-CARRY
   C = column1
   Sum = 10
   Next = column2
   ➔ **Goal:** Write carry in column2
   ➔ **Goal:** Write 0 as result in column1
4. WRITE-SUM
   C = column1
   Sum = 0
   Action: Write 0 as result in column1
   ← **Goal:** Write 0 as result in column1
4. **(alternative)** WRITE-CARRY
   C = column2
   Action: Write carry in column2
   ← **Goal:** Write carry in column2

```
  264
+ 716
```

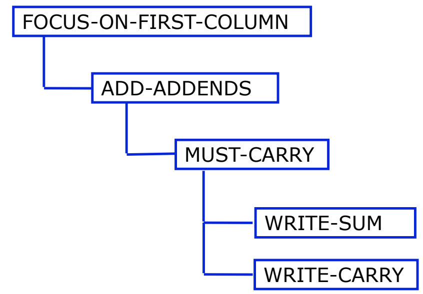


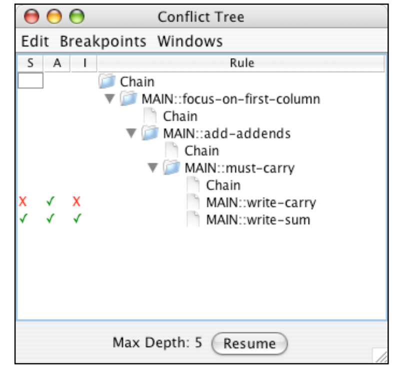
WRITE-SUM performs observable action - so done with this "path."

Likewise for WRITE-CARRY.

Each path (i.e., rule activation sequence) is explored up to the point where it leads to an observable action.

CTAT's Conflict Tree window: Debugging tool





Each path in the Conflict Tree leads to a single observable action.

Model tracing algorithm (simplified)

After a student action:

1. Use model to figure out all correct next steps: Use production rule model in "exploratory mode" to generate all sequences of rule firings that produce an "observable action"
   (changes to working memory are undone)
2. If student took one of these steps, then good! If student action is among the actions generated by the model, provide positive feedback
   and update working memory by firing the rule activations that produce the observable actions
   (so working memory and interface stay in sync)
3. Otherwise, if student made known error, provide error feedback message: if student action corresponds to path with "bug rule" Present (specific) error feedback message to student
   (and leave working memory unchanged)
4. Otherwise, error. Provide negative feedback
   (and leave working memory unchanged)

Production rule author must indicate which rules capture errors ("bug rules").

```
(defrule buggy-add-one-too-few
  ?problem <- (1column-addition-problem
                   (result nil)
                   (first-addend ?num1)
                   (second-addend ?num2))
=>
    (bind ?sum (- (+ ?num1 ?num2) 1)
    (predict-observable-action
         dorminTable1_C1R3     ; selection
         UpdateTable           ; action
         ?sum)                 ; input
    (modify ?problem (result ?sum))
    (construct-message [ You are one short. ])
 )
```

Rule name must contain the word "buggy".

On the RHS, compute sum incorrectly …

Observable action: enter incorrect sum

On RHS, use a call to function construct-message to communicate the error feedback message to the model-tracing algorithm. It will present this message to the student when the bug rule's predicted observable action corresponds to the student's action.

Attach hint templates to production rules

```
(defrule add
 ?problem <- (1column-addition-problem
                    (result nil)
                    (first-addend ?num1)
                    (second-addend ?num2))
=>
   (bind ?sum (+ ?num1 ?num2))
   (predict-observable-action
         dorminTable1_C1R3      ; selection
         UpdateTable            ; action
         ?sum)                  ; input
   (modify ?problem (result ?sum))
   (construct-message
       "[ What is the sum of " ?num1 " and " ?num2 "? ]"
       "[ What number do you get when start with " ?num1 " and " you count up "
            ?num2 " times? Use your fingers! ]"
       "[ Enter" ?sum ". ]"  )
 )
```

Add a hint template by calling function construct-message on the RHS of a rule.

- Each expression in [] is next hint level
- Can insert variables
- Put text (including []) in double quotes.

The model-tracing algorithm will present the hint messages when the student requests a hint, and the current rule is used to generate the next action.

CTAT CognitiveTutorAuthoringTools LearnLab Pittsburgh Science of Learning Center

What's not so smart about the way this rule encodes its observable action?

```
(defrule add
 ?problem <- (1column-addition-problem
                    (result nil)
                    (first-addend ?num1)
                    (second-addend ?num2))
=>
   (bind ?sum (+ ?num1 ?num2))
   (predict-observable-action
         dorminTable1_C1R3      ; selection
         UpdateTable            ; action
         ?sum)                  ; input
   (modify ?problem (result ?sum)))
```

The selection name is 'hard coded.'

The rule will work fine on single-column addition problems in the given interface.

But not in a slightly different interface (e.g., one with two single-column addition problems in two separate tables).

Also, consider multi-column addition. We don't want to write an "add" rule for each column separately!

CTAT CognitiveTutorAuthoringTools LearnLab Pittsburgh Science of Learning Center

To create more flexible model: represent interface in working memory

- Create a representation of the interface in working memory (e.g., a table)
- Write rules that "retrieve" (by matching) the fact in WM that represents the relevant interface element.
  - For example, "the bottom cell in rightmost column that has no result yet "
- Means more flexible rules (e.g., doesn't matter how many columns in the table) and greater re-use (same rule can work for different columns in the problem)
- Often provides a good representation for the problem!

CTAT CognitiveTutorAuthoringTools LearnLab Pittsburgh Science of Learning Center

New templates to represent a table, with columns, that have cells

```
(deftemplate problem
 (slot name)
 (multislot interface-elements)
 (multislot subgoals)
 (slot done))
(deftemplate table
 (slot name)
 (multislot columns))
(deftemplate column
 (slot name)
 (multislot cells))
(deftemplate cell
 (slot name)
 (slot value))
(deftemplate button
 (slot name))
```

CTAT Cognitive Tutor Authoring Tools — LearnLab Pittsburgh Science of Learning Center

Representing table, columns, and cells in working memory

```
;; Create three cell facts
?cell1 <- (assert (cell (name dorminTable1_C1R1) (value 4)))
?cell2 <- (assert (cell (name dorminTable1_C1R2) (value 3)))
?cell3 <- (assert (cell (name dorminTable1_C1R3) (value nil)))

;; Create a column fact
?column <- (assert (column (name dorminTable1_Column1)
                    (cells ?cell1 ?cell2 ?cell3)))

;; Create a table fact and two button facts
?table <- (assert (table (name dorminTable1))))
                   (columns ?column)))
?button1 <- (assert (button (name done))))
?button2 <- (assert (button (name hint))))

;; Create a problem fact
(assert (problem (name 4+3)
         (interface-elements ?button1 ?button2 ?table)))
```

Names of the interface elements (or widgets) are stored in corresponding fact in WM.

CTAT generates this representation for you!

CTAT Cognitive Tutor Authoring Tools — LearnLab Pittsburgh Science of Learning Center

Example: A more flexible way of having production rules generate predictions

```
(defrule add
   ?problem <- (problem (interface-elements $? ?table $?))
   ?table <- (table (columns $? ?first-column))
   ?first-column <- (column (cells $? ?first-addend ?second-addend ?result))
   ?result <- (cell (value nil) (name ?cell-name))
   ?first-addend <- (cell (value ?num1))
   ?second-addend <- (cell (value ?num2))
   (test (< (+ ?num1 ?num2) 10))
=>
   (bind ?sum (+ ?num1 ?num2))
   (predict-observable-action
         ?cell-name     ; selection
         UpdateTable    ; action
         ?sum)          ; input
   (modify ?result (value ?sum)))
```

On the LHS, match the fact that represents the relevant interface element (or widget) and bind its name to a variable (e.g., ?cell-name).

In the encoding of the observable action on the RHS, the selection is set to this variable.

CTAT Cognitive Tutor Authoring Tools — LearnLab Pittsburgh Science of Learning Center

More elaborate rule …

```
(defrule add
   ?problem <- (problem
               (interface-elements $? ?table $?))
   ?table <- (table (columns $? ?first-column))
   ?first-column <-
      (column
        (cells $? ?first-addend ?second-addend ?result))
   ?result <- (cell (value nil) (name ?cell-name))
   ?first-addend <- (cell (value ?num1))
   ?second-addend <- (cell (value ?num2))
   (test (< (+ ?num1 ?num2) 10))
=>
   (bind ?sum (+ ?num1 ?num2))
   (predict-observable-action
         ?cell-name
         UpdateTable
         ?sum)
   (modify ?result (value ?sum)))
```

If *?problem* is a problem, with ?*table* among its interface-elements
And *?table* is a table with *?column* as its last column
And *?column* is a column, with *?first-addend*, *?second-addend*, and *?result* as its last three cells
And *?result* is a cell with value nil and name *?cell-name*
And *?first-addend* is a cell with value *?num1*
And *?second-addend* is a cell with value *?num2*
And *?num1* + *?num2* < 10

Then set *?sum* to *?num1* + *?num2*
And predict as observable action:
selection: *?cell-name*
action: UpdateTable
input: *?sum*
And set the value of *?result* to *?sum*

CTAT Cognitive Tutor Authoring Tools — LearnLab Pittsburgh Science of Learning Center

Left-Hand Side - Example pattern constraints

• The two rightmost elements of a list	($? ?second-col ?first-col)
• The two rightmost elements in a list of 3	(? ?second-col ?first-col)
• Any adjacent pair of list elements	($?before ?x1 ?x2 $?after)
• Any ordered pair of list elements	($? ?x1 $? ?x2 $?)
• Pairs of duplicate elements of a list	($? ?x1 $? ?x1 $?)

More Jess notation: Constraining slot data on the left-hand side of rules

- Literal Constraints (cell (value 1))
- Variable Constraints (cell (value ?val))
- Connective Constraints (cell (value ?val&:(neq ?val nil)))
- Predicate Constraints (test (> (+ ?num1 ?num2) 9))
- Pattern Constraints (for multi-slots) ($? ?x1 $? ?x1 $?)

Right-Hand Side - Typical function calls

- Bind - Specify a new variable, e.g.
  - **(bind ?sum (+ ?num1 ?num2))**
- Modify - Update a variable, typically from LHS, e.g.,
  - **(modify ?result (value ?new-sum))**
- Assert - Create a new fact
  - **(assert (write-carry-goal**
    (carry 1)
    (column ?second-column))))
- Retract - Delete an existing fact
  - **(retract ?result)**

Summary- Model tracing with CTAT

- Model tracing: the way CTAT uses a cognitive model to individualize instruction
  - Jess inference engine modified: build Conflict Tree and choose “path” that performs action that student took
- Rule author must
  - indicate whether rule encodes correct or incorrect behavior
  - encode observable actions on RHS with function predict -observable-action
  - attach hints
- It is often a good idea to represent the interface in working memory