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

Vincent Aleven

7th Annual PSLC Summer School
Pittsburgh, July 25 - 29, 2011

Track overview

- The CTAT track will cover development of Cognitive Tutors and Example-Tracing Tutors
  - You could focus on Example-tracing Tutors only, but this is a good opportunity to learn about Cognitive Tutor with the help of an experienced mentor

- Activities
  - Lecture about grounding of Cognitive Tutor technology in ACT-R
  - Number of "how to" lectures about cognitive modeling and model tracing
  - Hands-on activities focused on building tutors, both on Example-Tracing Tutors and Cognitive Tutors (fraction addition)
  - Project

Materials

- Retrieve slides, instructions for hands-on work, and example tutors (needed for hands-on activities) from:
- For starters, download file “Fraction Addition Tutors 2011.zip”
- Additional files may become available during the week

Overview

- ACT-R theory
  - Features of production rules and their predictions about learning
- How Production Systems Work
  - A simple example
  - A more complex example: fraction 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: fraction 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

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

\[
\text{IF} \quad \text{the 3 corresponding sides of two triangles are congruent (\(\cong\))} \\
\text{THEN} \quad \text{the triangles are} \cong
\]

Production rules describe thinking patterns:

- Special condition to aid search
  - IF two triangles share a side AND the other 2 corresponding sides are \(\cong\)
  - THEN the triangles are congruent (\(\cong\))

- Using rule backward
  - IF goal: prove triangles \(\cong\) AND 2 sets of corresponding sides are \(\cong\)
  - THEN subgoal: prove 3rd set of sides \(\cong\)

- Using rule heuristically
  - IF two triangles look \(\cong\)
  - THEN try to prove any of the corresponding sides & angles \(\cong\)

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

Production Rule Analysis

Evidence for Production Rule as an appropriate unit of knowledge acquisition

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

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 rule asymmetry example

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 ≡

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 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 "a + b" 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!)

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: fraction 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

First cycle of execution

<table>
<thead>
<tr>
<th>WORKING MEMORY</th>
<th>CYCLE 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>WM = (green, weighs-15-lbs)</td>
<td></td>
</tr>
<tr>
<td>1. Productions whose condition parts are true: <strong>P1</strong></td>
<td></td>
</tr>
<tr>
<td>2. No production would add duplicate symbol</td>
<td></td>
</tr>
<tr>
<td>3. Execute <strong>P1</strong>.</td>
<td></td>
</tr>
<tr>
<td>This gives: WM = (produce, green, weighs-15-lbs)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RULE MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1. IF green THEN produce</td>
</tr>
<tr>
<td>P2. IF packed-in-small-container THEN delicacy</td>
</tr>
<tr>
<td>P3. IF refrigerated OR produce THEN perishable</td>
</tr>
<tr>
<td>P4. IF weighs-15-lbs AND inexpensive AND NOT perishable THEN staple</td>
</tr>
<tr>
<td>P5. IF perishable AND weighs-15-lbs THEN turkey</td>
</tr>
<tr>
<td>P6. IF weighs-15-lbs AND produce THEN watermelon</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INTERPRETER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Find all productions whose condition parts are true</td>
</tr>
<tr>
<td>2. Deactivate productions that would add a duplicate symbol</td>
</tr>
<tr>
<td>3. Execute the lowest numbered production (or quit)</td>
</tr>
<tr>
<td>4. Repeat</td>
</tr>
</tbody>
</table>

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

### 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

### 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

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 staple
P6. IF weighs-15-lbs AND produce THEN watermelon

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

CYCLE 3
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.

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

Some tutors for you to look at

- French culture
  - http://www.andrew.cmu.edu/user/aeo/tutor/ibrahim-exp.swf
  - http://www.andrew.cmu.edu/user/aeo/tutor/ibrahim-controlV2.swf

- Stoichiometry
Overview

• ACT-R theory
  – Features of production rules and their predictions about learning
• How Production Systems Work
  – A simple example
  – A more complex example: fraction 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 fraction addition?
  \[ \frac{1}{3} + \frac{2}{5} = \? \]

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

Solution steps for fraction addition

<table>
<thead>
<tr>
<th>Given Fractions</th>
<th>Converted Fractions</th>
<th>Simplified Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \frac{1}{6} ]</td>
<td>[ \frac{1}{6} ]</td>
<td>[ \frac{4}{15} ]</td>
</tr>
<tr>
<td>+ [ \frac{4}{15} ]</td>
<td>[ \frac{4}{15} ]</td>
<td>[ \frac{13}{30} ]</td>
</tr>
</tbody>
</table>
Production rule - skeleton

```xml
<RULE-NAME>
  IF
  There is a (sub)goal to ...
  In the current state of problem solving is such that ...
  THEN
  Perform the following (observable) actions: ...
  Set a subgoal to ...
  Remove the subgoal to ...
  "IF part" or "Condition" or "Left-hand side"
  "THEN part" or "Action" or "Right-hand side"

Note: Not all condition/actions types are present in all rules
```

Production rules set new goals and perform actions

**Goal:** Solve fraction addition problem
- **DETERMINE-LCD**
- **DETERMINE-REDUCTION-FACTOR**
- **ADD-NUMERATORS**
- **REDUCE-NUMERATOR**
- **REDUCE-DENOMINATOR**
- **COPY-ANSWER-DENOMINATOR**

**Goal:** Convert fraction to a common denominator
- **CONVERT-NUMERATOR**
- **CONVERT-DENOMINATOR**

**Goal:** Add converted fractions
- **Action:** Write the numerator of the sum fraction
- **Action:** Write the denominator of the sum fraction

**Goal:** Reduce the sum fraction
- **Action:** Write the converted numerator
- **Action:** Write the converted denominator

**Goal:** Add converted fractions
- **Action:** Write the converted numerator
- **Action:** Write the converted denominator

**Goal:** Convert fraction to a common denominator
- **Action:** Write the converted numerator
- **Action:** Write the converted denominator

**Goal:** Reduce the sum fraction
- **Action:** Write the converted numerator
- **Action:** Write the converted denominator

**Goal:** Add converted fractions
- **Action:** Write the converted numerator
- **Action:** Write the converted denominator

**Goal:** Reduce the sum fraction
- **Action:** Write the converted numerator
- **Action:** Write the converted denominator

**Goal:** Add converted fractions
- **Action:** Write the converted numerator
- **Action:** Write the converted denominator

**Goal:** Reduce the sum fraction
- **Action:** Write the converted numerator
- **Action:** Write the converted denominator

Names of variables are shown in bold+italics: e.g., $D1$, $LCD$
Rules for fraction addition (2)

**DETERMINE-PRODUCT**

*IF* there is a fraction addition problem and there are no subgoals and \( D_1 \) is the denominator of the first given fraction and \( D_2 \) is the denominator of the second given fraction

*THEN*

Set \( \text{PROD} \) to \( D_1 \) and \( D_2 \)
Set subgoals to convert the fractions to denominator \( \text{PROD} \)
Set a subgoal to add the converted fractions

Rules for fraction addition (3)

**DETERMINE-COMMON-MULT**

*IF* there is a fraction addition problem and there are no subgoals and \( D_1 \) is the denominator of the first given fraction and \( D_2 \) is the denominator of the second given fraction

*THEN*

Set \( \text{MULT} \) to \( D_1 \) and \( D_2 \)
Set subgoals to convert the fractions to denominator \( \text{MULT} \)
Set a subgoal to add the converted fractions

Rules for fraction addition (4)

**CONVERT-DENOMINATOR**

*IF* there is a subgoal to convert fraction \( F \) to denominator \( D \) and the denominator of \( F \) has not been converted yet

*THEN*

Write \( D \) as the denominator
Make a note that the denominator part of the subgoal is done

Rules for fraction addition (5)

**CONVERT-NUMERATOR**

*IF* there is a subgoal to convert fraction \( F \) to denominator \( D \) and the numerator of \( F \) has not been converted yet and the (original) numerator of \( F \) is \( \text{NUM} \) and the denominator of \( F \) is \( \text{DENOM} \)

*THEN*

Write \( \text{NUM} * (D / \text{DENOM}) \) as the numerator of the converted fraction
Mark the subgoal: numerator done
Can you complete these rules?

**ADD-NUMERATORS**
**IF** there is a subgoal to add two fractions
And __________________________
______________________________
______________________________
**THEN**
Write $N_1 + N_2$ as the numerator of the (unreduced) sum fraction
Mark the numerator part of the subgoal done

**COPY-ANSWER-DENOMINATOR**
**IF** there is a subgoal to add two fractions
And __________________________
______________________________
______________________________
**THEN**
Write $DENOM$ as the denominator of the sum fraction

More rules for you to work on ...

**DETERMINE-REDUCTION-FACTOR**
**IF** the fractions have been added
And __________________________
______________________________
______________________________
**THEN**
Set a subgoal to reduce the fraction, using the GCD of the denominator and numerator as the reduction factor

**REDUCE-NUMERATOR**
**IF** there is a subgoal to reduce the answer fraction, with reduction factor $F$
And __________________________
______________________________
______________________________
**THEN**
Write $N/F$ as the numerator of the reduced answer fraction

And yet more ...

**ADD-NUMERATORS**
**IF** there is a subgoal to add two fractions (the converted fractions if the original fractions need to be converted, the original fractions otherwise)
and the numerators have not been added yet
and the numerators are known (i.e., have been converted)
and $N_1$ and $N_2$ are the numerators of the two fractions
**THEN**
Write $N_1 + N_2$ as the numerator of the (unreduced) sum fraction
Mark the numerator part of the subgoal done

**COPY-ANSWER-DENOMINATOR**
**IF** there is a subgoal to add two fractions
and the denominator of the sum fraction has not been written yet
and $DENOM$ is the denominator of one of the converted fractions
**THEN**
write $DENOM$ as the denominator of the sum fraction

**REDUCE-DENOMINATOR**
**IF** there is a subgoal to reduce the answer fraction, with reduction factor $F$
And __________________________
______________________________
______________________________
**THEN**
Write $D/F$ as the denominator of the reduced answer fraction

**DONE**
**IF**
And __________________________
______________________________
______________________________
**THEN** Mark the problem as done
Production rules for fraction addition

**Production Rule Firings**

1. **DETERMINE-LCD**
   
   IF there is a goal to solve a fraction addition problem and $D_1$ is the denominator of the first given fraction and $D_2$ is the denominator of the second given fraction
   
   THEN
   
   Set $LCD$ to the least common denominator of $D_1$ and $D_2$

2. **CONVERT-DENOMINATOR**
   
   IF there is a subgoal to convert fraction $F$ to denominator $D$ and the denominator of $F$ has not been converted yet
   
   THEN
   
   Write $D$ as the denominator
   
   Make a note that the denominator part of the subgoal is done

3. **CONVERT-NUMERATOR**
   
   IF there is a subgoal to convert fraction $F$ to denominator $D$ and the numerator of $F$ has not been converted yet
   
   THEN
   
   Write $NUM = (D / DENOM)$ as the numerator of the converted fraction
   
   Mark the subgoal: numerator done

4. **ADD-NUMERATORS**
   
   IF there is a subgoal to add two fractions (the converted fractions if the original fractions need to be converted, the original fractions otherwise)
   
   and the numerators have not been added yet
   
   and the numerators are known (i.e., have been converted)
   
   and $N_1$ and $N_2$ are the numerators of the two fractions
   
   THEN
   
   Write $N_1 + N_2$ as the numerator of the (unreduced) sum fraction
   
   Mark the numerator part of the subgoal done

**Working memory**

Subgoals

1. Convert first given fraction
2. Convert second given fraction
3. Add converted fractions

**Working memory**

Subgoals

1. Convert first given fraction
2. Convert second given fraction
3. Add converted fractions

**Working memory**

Subgoals

1. Convert first given fraction
2. Convert second given fraction
3. Add converted fractions

**Working memory**

Subgoals

1. Convert first given fraction
2. Convert second given fraction
3. Add converted fractions

**Questions**

Q: With different conflict resolution, could the denominator of the second converted fraction have been written first?

A: Yes, after DETERMINE-LCD fires, there are two activations of CONVERT-DENOMINATOR, one for each given fraction. (i.e., in one, variable $F$ is bound to the first given fraction, in the other, to the second given fraction.)

Q: Could the numerator of one of the converted fractions have been written first?

A: Yes, after DETERMINE-LCD fires, there are two activations of CONVERT-DENOMINATOR, one for each given fraction.

**Production Rule Firings**

1. DETERMINE-LCD

   Variable bindings
   
   $D_1 = 3$
   
   $D_2 = 5$

   $LCD = 15$

   Changes to working memory
   
   Add subgoal: Convert first given fraction to denominator 15
   
   Add subgoal: Convert second given fraction to denominator 15

   **Action:**
   
   Write 15 as denominator of first given fraction
   
   Add converted fractions

   **Subgoals**

   1. Convert first given fraction
   2. Convert second given fraction
   3. Add converted fractions

2. CONVERT-DENOMINATOR

   **Action:**
   
   Changes to working memory
   
   **Subgoals**

   1. Convert first given fraction
   2. Convert second given fraction
   3. Add converted fractions

3. ADD-NUMERATORS

   **Variable bindings**
   
   $F$ = first given fraction
   
   $D = 15$

   Changes to working memory
   
   **Subgoals**

   1. Convert first given fraction
   2. Convert second given fraction
   3. Add converted fractions

4. REDUCE-DENOMINATOR

   **Variable bindings**
   
   $D/F$ = first given fraction
   
   **Subgoals**

   1. Convert first given fraction
   2. Convert second given fraction
   3. Add converted fractions

5. REDUCE-NUMERATOR

   **Variable bindings**
   
   $N/F$ = second given fraction
   
   **Subgoals**

   1. Convert first given fraction
   2. Convert second given fraction
   3. Add converted fractions

**Interpreter, Cycle 1**

**Match**

• DETERMINE-LCD (1x)

• DETERMINE-PRODUCT (1x)

• DETERMINE-COMMON-MULTIPLE (1x)

**Conflict Resolution**

• Let’s say DETERMINE-LCD is chosen to fire

**Questions**

Q: With different conflict resolution, could the denominator of the second converted fraction have been written first?

A: Yes, after DETERMINE-LCD fires, there are two activations of CONVERT-DENOMINATOR, one for each given fraction. (i.e., in one, variable $F$ is bound to the first given fraction, in the other, to the second given fraction.)

Q: Could the numerator of one of the converted fractions have been written first?

A: Yes, after DETERMINE-LCD fires, there are two activations of CONVERT-DENOMINATOR, one for each given fraction.
**Production Rule Firings**

1. DETERMINE-LCD
2. CONVERT-DENOMINATOR
3. CONVERT-NUMERATOR
4. CONVERT-DENOMINATOR
5. CONVERT-NUMERATOR

**Variable bindings**

- $F =$ first given fraction
- $D =$ denominator
- $NUM =$ numerator
- $DENOM =$ denominator

**Changes to working memory**

- **DENOM**
- **NUM**

**Questions**

**Q:** With different conflict resolution, could the numerator of the sum fraction be written at this point? The denominator?

**A:** No, the numerator of the sum fraction cannot be written yet. ADD-NUMERATORS will not fire until the numerators of the fractions to be added are known. However, the if-part of COPY-ANSWER-DENOMINATOR is satisfied, so the denominator of the sum fraction could be written.

---

**Production Rule Firings**

1. DETERMINE-LCD
2. CONVERT-DENOMINATOR
3. CONVERT-NUMERATOR
4. CONVERT-DENOMINATOR
5. CONVERT-NUMERATOR

**Variable bindings**

- $F =$ second given fraction
- $D =$ denominator
- $NUM =$ numerator
- $DENOM =$ denominator

**Changes to working memory**

- **DENOM**
- **NUM**

**Questions**

**Q:** With different conflict resolution, could the denominator of the second converted fraction have been written first (i.e., before the numerator)?

**A:** Yes, after DETERMINE-LCD fires in cycle 1, there are two activations of CONVERT-DENOMINATOR, one for each given fraction. Each could fire in cycle 2, 3, or 4.
### Production Rule Firings

1. **DETERMINE-LCD**
2. **CONVERT-DENOMINATOR**
3. **CONVERT-NUMERATOR**
4. **CONVERT-DENOMINATOR**
5. **CONVERT-DENOMINATOR**
6. **COPY-ANSWER-DENOMINATOR**
7. **ADD-NUMERATORS**

### Working memory

\[
\frac{1}{3} + \frac{2}{5} = \frac{11}{15}
\]

### Subgoals

1. Convert first given fraction
2. Convert second given fraction
3. Add converted fractions

### Interpreter, Cycle 7

**Match**

- ADD-NUMERATORS (1x)

**Conflict Resolution**

- Select activation of ADD-NUMERATORS

### Questions

Q: With different conflict resolution, could the answer have been copied to the simplified answer fraction?

A: No, the DETERMINE-REDUCTION-FACTOR rule requires that the GCD of the numerator and denominator is greater than 1.

---

### Production Rule Firings

1. **DETERMINE-LCD**
2. **CONVERT-DENOMINATOR**
3. **CONVERT-NUMERATOR**
4. **CONVERT-NUMERATOR**
5. **CONVERT-DENOMINATOR**
6. **COPY-ANSWER-DENOMINATOR**
7. **ADD-NUMERATORS**
8. **DONE**

### Working memory

\[
\frac{1}{3} + \frac{2}{5} = \frac{11}{15}
\]

### Subgoals

1. Convert first given fraction
2. Convert second given fraction
3. Add converted fractions

### Interpreter, Cycle 8

**Match**

- DONE (1x)

**Conflict Resolution**

- Select activation of DONE

### Summary

- **Components of Production Systems:**
  - Working memory, production memory, interpreter
  - Steps in the interpreter: Match, conflict resolution, fire

- **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
Overview

- ACT-R theory
  - Features of production rules and their predictions about learning
- How Production Systems Work
  - A simple example
  - A more complex example: fraction 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


Why do we have the textField facts (instead of storing the denominator and numerator values in the fraction facts)?

As we will see later, the textField facts help with model tracing (i.e., help when the model is used to provide tutoring). The values in the name slot of these facts correspond to the component values set in Flash or Netbeans.
Design and implement working memory representation

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

(deftemplate MAIN::problem
(slot name)
(multislot given-fractions)
(multislot converted-fractions)
(multislot answer-fractions)
(slot subgoals)
(slot done))

(deftemplate MAIN::fraction
(slot name)
(slot numerator)
(slot denominator)
(slot has-converted-form)
(slot is-converted-form-of))

(deftemplate MAIN::textField
(slot name)
(slot value))

Create the facts in two steps:
1. Create each fact
2. Fill in values and cross-references

Creating facts in working memory

*Assert* creates a fact and puts it in working memory

*Bind* creates a new variable and gives it a value

;; Fact representing the problem
(bind ?var21 (assert (problem (name 1-3plus2-5))))

;; One of the given fractions
(bind ?var22 (assert (fraction (name given-fraction1))))

;; One of the converted fractions
(bind ?var24 (assert (fraction (name converted-fraction1))))

;; Four facts representing text fields
(bind ?var8 (assert (textField (name givenNum1))))
(bind ?var9 (assert (textField (name givenDenom1))))
(bind ?var14 (assert (textField (name convertNum1))))
(bind ?var15 (assert (textField (name givenNum1)))))
Create the facts in two steps:
1. **Create each fact**
2. **Fill in values and cross-references**
Initial working memory representation ("start state")

Jess> (facts)
  f-0  (MAIN::initial-fact)
  ...
  f-2  (MAIN::textField (name givenNum1) (value 1))
  f-3  (MAIN::textField (name givenDenom1) (value 4))
  ...
  f-8  (MAIN::textField (name convertNum1) (value nil))
  f-9  (MAIN::textField (name convertDenom1) (value nil))
  ...
  f-15 (MAIN::problem (name 1-3plus2-5) (given-fractions <Fact-16> <Fact-17>) (converted-fractions <Fact-18> <Fact-19>) (answer-fractions <Fact-20> <Fact-21>) (subgoals) (more-subgoals) (done nil))
  f-16 (MAIN::fraction (name given-fraction1) (numerator <Fact-2>) (denominator <Fact-3>) (has-converted-form <Fact-18>) (is-converted-form-of nil))
  ...
  For a total of 22 facts in module MAIN.

Production rules will create subgoals in working memory
(need templates for subgoals, too)

(deftemplate MAIN::convert-fraction-goal
  (slot fraction)
  (slot new-denominator)
  (slot denom-done)
  (slot num-done))

(deftemplate MAIN::add-fractions-goal
  (multislot fractions)
  (slot denom-done)
  (slot num-done))

(deftemplate MAIN::reduce-fraction-goal
  (slot fraction)
  (slot factor)
  (slot denom-done)
  (slot num-done))

Jess production rule for first cycle

English
DETERMINE-LCD
IF there is a goal to solve a fraction addition problem
and there are no subgoals
and D1 is the denominator of the first
given fraction
and D2 is the denominator of the second given fraction
THEN
Set LCD to the least common
denominator of D1 and D2
Set subgoals to convert the fractions
to denominator LCD
Set a subgoal to add the converted fractions

Jess
(defrule determine-lcd
  ?problem <- (problem
    (subgoals)
    (given-fractions ?f1 ?f2)
    (converted-fractions ?f3 ?f4)
    (answer-fractions ?f5 ?f6)
    (subgoals) (more-subgoals) (done nil))
  ?f1 <- (fraction (denominator ?d1&~nil))
  ?f2 <- (fraction (denominator ?d2&~nil))
=>
  (bind ?lcd (lcm ?d1 ?d2))
  (bind ?sub1 (assert (convert-fraction-goal
    (fraction ?f1)
    (new-denominator ?lcd)))
  (bind ?sub2 (assert (convert-fraction-goal
    (fraction ?f2)
    (new-denominator ?lcd)))
  (bind ?sub3 (assert (add-fractions-goal
    (fractions ?f3 ?f4)))
  (modify ?problem (subgoals ?sub1 ?sub2 ?sub3))
  (printout t crlf "determine-lcd, ?lcd = " ?lcd))

Jess production rule notation

Name of the rule (defrule determine-lcd)
If-part Specifies conditions
| ?problem <- (problem
  (subgoals)
  (given-fractions ?f1 ?f2)
  (converted-fractions ?f3 ?f4)
  (answer-fractions ?f5 ?f6)
  (subgoals) (more-subgoals) (done nil))
| ?f1 <- (fraction (denominator ?d1&~nil))
| ?f2 <- (fraction (denominator ?d2&~nil))

Then-part Specifies actions
| (bind ?lcd (lcm ?d1 ?d2))
| (bind ?sub1 (assert (convert-fraction-goal
  (fraction ?f1)
  (new-denominator ?lcd)))
| (bind ?sub2 (assert (convert-fraction-goal
  (fraction ?f2)
  (new-denominator ?lcd)))
| (bind ?sub3 (assert (add-fractions-goal
  (fractions ?f3 ?f4)))
| (modify ?problem (subgoals ?sub1 ?sub2 ?sub3))
| (printout t crlf "determine-lcd, ?lcd = " ?lcd))

Variables ("bound" to values through matching)

Patterns Specify configurations of facts in working memory

Function calls Perform computations, change facts in working memory
Set a subgoal to convert
and there are no subgoals

IF
DETERMINE-LCD

THEN
Set LCD to the least common multiple
of D1 and D2
Set a subgoal to convert F1 to denominator LCD
Set a subgoal to convert F2 to denominator LCD
Set a subgoal to add the fractions F3 and F4

Declarative model

Jess production rule for first cycle

English

DETERMINE-LCD

IF there is a goal to solve a fraction addition problem
and there are no subgoals
and F1 is the first given fraction
and F2 is the second given fraction
and F3 is the first converted fraction
and F4 is the second converted fraction
and D1 is the denominator of F1
and D2 is the denominator of F2

THEN
Set LCD to the least common multiple of D1 and D2
Set a subgoal to convert F1 to denominator LCD
Set a subgoal to convert F2 to denominator LCD
Set a subgoal to add the fractions F3 and F4

Jess

(defrule determine-lcd
?problem <- (?problem
(subgoals
(given-fractions ?f1 ?f2)
(converted-fractions ?f3 ?f4)
?f1 <- (fraction (denominator ?denom1))
?f2 <- (fraction (denominator ?denom2))
?denom1 <- (textField value ?d1&:(neq ?d1 nil))
?denom2 <- (textField value ?d2&:(neq ?d2 nil))
=>
(bind ?new-den (lcm ?d1 ?d2))
(bind ?sub1 (assert (convert-fraction-goal
(fraction ?f1)
(new-denominator ?new-den))))
(bind ?sub2 (assert (convert-fraction-goal
(fraction ?f2)
(new-denominator ?new-den))))
(bind ?sub3 (assert (add-fractions-goal
(fractions ?f3 ?f4))))
(printout t crlf *determine-lcd, ?ldc = " ?ldc))

Additional conditions

Other conditions: Binding variables to values needed
on rhs. These conditions always succeed.

Jess production rule for second cycle

English

CONVERT-DENOMINATOR

IF there is a subgoal to convert fraction F to denominator D
and the denominator of F has not been converted yet

THEN
Write D as the denominator
Make a note that the denominator part of the subgoal is done

Jess

(defrule convert-denominator-1
?problem <- (?problem
(subgoals
(given-fractions ?f ?f)
(converted-fractions ?f ?f)
?f <- (fraction (denominator ?denom))
?denom <- (textField value ?d&:(neq ?d nil))
=>
(p predict-observable-action
convertDenom1 ; selection
UpdateTextField ; action
?new-den) ; input
(modify ?conv-f (denominator ?new-den))
(modify ?sub (denom-done T)))
**Jess production rule notation**

**Name of the rule**

(defrule convert-denominator-1

?problem <- (problem (subgoals $? ?sub $?)

(given-fractions ?f ?f))

?sub <- (convert-fraction-goal

(fraction ?f)

(new-denominator ?new-den)

(denom-done nil))

?f <- (fraction (has-converted-form ?conv-f))

=>

(predict-observable-action

convertDenom1 ; selection

UpdateTextField ; action

?new-den) ; input

(modify ?conv-f (denominator ?new-den))

(modify ?sub (denom-done T)))

**Variables**

("bind" to values through matching)

**Patterns**

Specify configurations of facts in working memory

**Function calls**

Perform computations, change facts in working memory

---

**Jess production rule for second cycle**

**Note:** the fact that in these two examples the "key conditions" come before the other conditions is coincidental.

**English (elaborated)**

**CONVERT-DENOMINATOR**

**If** there is a subgoal to convert fraction \( F \) to denominator \( \text{NEW-DEN} \) and the denominator part of this subgoal is not marked as done and \( \text{CONV-F} \) is the converted form of \( F \) and \( \text{FIELD-NAME} \) is the name of the text field \( \text{CONV-DENOM} \)

**Then**

Write \( \text{NEW-DEN} \) as the value in field \( \text{FIELD-NAME} \)

Set the value of \( \text{CONV-DENOM} \) to \( \text{NEW-DEN} \)

Make a note that the denominator part of the subgoal is done

**Jess**

(defrule convert-denominator

?problem <- (problem (subgoals $? ?sub $?)

(given-fractions ?f ?f))

?sub <- (convert-fraction-goal

(fraction ?f)

(new-denominator ?new-den)

(denom-done nil))

?f <- (fraction (has-converted-form ?conv-f))

=>

(predict-observable-action

?field-name UpdateTextField ?new-den)

(modify ?conv-denom (value ?new-den))

(modify ?sub (denom-done T)))

**Other conditions:** Binding variables to values needed on rhs. These conditions always succeed.

---

**Matching a production rule against working memory**

**Production Rule**

(defrule determine-lcd

?problem <- (problem (subgoals $? ?sub $?)

(given-fractions ?f1 ?f2))

(fraction ?f1)

(fraction ?f2)

(new-denominator ?lcd)

=>

(predict-observable-action

?sub (denom-done T)))

**Working Memory**

<table>
<thead>
<tr>
<th>F-1</th>
<th>(MAIN::problem (name 1-3plus2-5))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(given-fractions &lt;Fact-1&gt; &lt;Fact-2&gt;)</td>
</tr>
<tr>
<td></td>
<td>(converted-fractions &lt;Fact-3&gt; &lt;Fact-4&gt;)</td>
</tr>
<tr>
<td></td>
<td>(answer-fractions &lt;Fact-5&gt; &lt;Fact-6&gt;)</td>
</tr>
<tr>
<td></td>
<td>(fractions &lt;Fact-3&gt; &lt;Fact-4&gt;)</td>
</tr>
<tr>
<td></td>
<td>(given-denominator &lt;Fact-7&gt;)</td>
</tr>
<tr>
<td></td>
<td>(is-converted-form-of nil)</td>
</tr>
<tr>
<td>F-2</td>
<td>(MAIN::convert-fraction-goal (frac ?f1) (new-denominator ?new-den))</td>
</tr>
<tr>
<td></td>
<td>(is-converted-form-of nil)</td>
</tr>
<tr>
<td>F-3</td>
<td>(MAIN::add-fractions-goal (fractions &lt;Fact-3&gt; &lt;Fact-4&gt;))</td>
</tr>
<tr>
<td></td>
<td>(is-converted-form-of nil)</td>
</tr>
<tr>
<td>F-4</td>
<td>(MAIN::fraction (name conv2) (given-denominator nil))</td>
</tr>
<tr>
<td>F-5</td>
<td>(MAIN::fraction (name conv1))</td>
</tr>
<tr>
<td>F-6</td>
<td>(MAIN::fraction (name giv2))</td>
</tr>
<tr>
<td>F-7</td>
<td>(MAIN::problem (name 1-3plus2-5))</td>
</tr>
<tr>
<td></td>
<td>(given-numerator &lt;Fact-1&gt;)</td>
</tr>
<tr>
<td></td>
<td>(is-converted-form-of nil)</td>
</tr>
<tr>
<td></td>
<td>(is-converted-form-of nil)</td>
</tr>
</tbody>
</table>

**Find value for each variable**

<table>
<thead>
<tr>
<th>?f1</th>
<th>&lt;Fact-1&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>?f2</td>
<td>&lt;Fact-2&gt;</td>
</tr>
<tr>
<td>?f3</td>
<td>&lt;Fact-3&gt;</td>
</tr>
<tr>
<td>?f4</td>
<td>&lt;Fact-4&gt;</td>
</tr>
<tr>
<td>?problem</td>
<td>&lt;Fact-7&gt;</td>
</tr>
<tr>
<td>?d1</td>
<td>3</td>
</tr>
<tr>
<td>?d2</td>
<td>5</td>
</tr>
<tr>
<td>?n0</td>
<td>1.5</td>
</tr>
<tr>
<td>?sub1</td>
<td>&lt;Fact-8&gt;</td>
</tr>
<tr>
<td>?sub2</td>
<td>&lt;Fact-9&gt;</td>
</tr>
<tr>
<td>?sub3</td>
<td>&lt;Fact-10&gt;</td>
</tr>
</tbody>
</table>

**What changes are made to working memory?**

<table>
<thead>
<tr>
<th>F-8</th>
<th>(MAIN::convert-fraction-goal (fraction &lt;Fact-1&gt;) (new-denominator 15))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(denom-done nil)</td>
</tr>
<tr>
<td></td>
<td>(num-done nil)</td>
</tr>
<tr>
<td>F-9</td>
<td>(MAIN::convert-fraction-goal (fraction &lt;Fact-2&gt;) (new-denominator 15))</td>
</tr>
<tr>
<td></td>
<td>(denom-done nil)</td>
</tr>
<tr>
<td></td>
<td>(num-done nil)</td>
</tr>
<tr>
<td>F-10</td>
<td>(MAIN::add-fractions-goal (fractions &lt;Fact-3&gt; &lt;Fact-4&gt;))</td>
</tr>
<tr>
<td></td>
<td>(is-converted-form-of nil)</td>
</tr>
<tr>
<td></td>
<td>(is-converted-form-of nil)</td>
</tr>
<tr>
<td></td>
<td>(is-converted-form-of nil)</td>
</tr>
</tbody>
</table>
Summary—Jess production rule notation

- Working memory is a collection of facts
  F-7 (MAIN::problem (name 1-3plus2-5) (given-fractions <Fact-1> <Fact-2>) (converted-fractions <Fact-3> <Fact-4>) (answer-fractions <Fact-5> <Fact-6>) (subgoals ) (done nil))

- A template defines a type of fact and the slots of the type:
  (deftemplate MAIN::problem (slot name) (multislot given-fractions) (multislot converted-fractions) (multislot answer-fractions) (multislot subgoals) (slot done))

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

- THEN-part: computations and changes to working memory

Overview

- ACT-R theory
  - Features of production rules and their predictions about learning

- How Production Systems Work
  - A simple example
  - A more complex example: fraction 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

Model tracing:
Use cognitive model to individualize instruction

- Cognitive Model: A system that can solve problems in the various ways students can
  If goal is solve a(bx+c) = d
  Then rewrite as abx + ac = d
  3(2x - 5) = 9
  If goal is solve a(bx+c) = d
  Then rewrite as abx + c = d
  6x - 15 = 9
  2x - 5 = 3
  6x - 5 = 9

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

Hint message: "Distribute a across the parentheses."
Bug message: "You need to multiply c by a also."
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)

Need to generate sequences of rule activations that produce an observable action

Conflict Tree: systematic enumeration of model-generated next steps

- Search for all sequences of rule activations that start in the current state and end in an "observable action"
- Do depth-first search through the space of rule activations
- Fire rules to "see" their consequences (i.e., changes to working memory, or observable actions)
- Back up when a rule activation generates an observable action, or when there are no rule activations
- When backing up, undo changes made to working memory
- Space of rule activations can be depicted graphically (called "Conflict Tree")
Conflict Tree: shows sequences of rule activations that starting in the current state, end in an “observable action”

CTAT’s Conflict Tree window: Debugging tool

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

How are student actions compared against the actions predicted by the model?
Production rule author must indicate which RHS actions correspond to observable actions.

(defrule convert-denominator
  (?problem <- (problem (subgoals $? $?)
    (sub $?)
  )
  (?sub <- (convert-fraction-goal
    (denom-done nil)
    (fraction $?)
    (denominator-value
      ?new-den&(neq ?new-den nil)))
  )
  (?f1 <- (fraction (has-converted-form ?conv-f1))
    (?conv-f1 <- (fraction (denominator ?conv-denom))
    ?conv-denom <- (textField
      (name ?field-name)
      (value nil)))
  )
  )
  )
  =>
  (predict-observable-action
    ?field-name ; selection
    UpdateTextField ; action
    ?new-den ; input
    (modify ?conv-denom (value ?new-den))
    (modify ?sub (denom-done T)))
)

The LHS finds the fact in working memory that corresponds to the relevant interface component, and assigns it to a variable.

On the RHS, observable actions simulated by the model (i.e., actions that are "visible" in the interface) are communicated to the model-tracing algorithm by means of a call to function predict-observable-action. The model-tracing algorithm compares the model-generated actions against the student’s actions.

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

(defrule BUG-add-denominators
  (?problem <- (problem (subgoals $? $?)
    (sub $?)
  )
  (?f1 <- (fraction (denominator ?den1))
    ?den1 <- (textField (value ?d1)))
  (?f2 <- (fraction (denominator ?den2))
    ?den2 <- (textField (value ?d2)))
  (?answer <- (fraction (ans-denom))
    ?ans-denom <- (textField (value nil)(name ?name)))
  (bind ?sum (+ ?d1 ?d2))
  )
  =>
  (predict-observable-action
    ?name UpdateTextField
    ?sum?
    (construct-message
      [You added the denominators. Instead, you need to find a denominator that is a multiple of ?d1 and a multiple of ?d2.])
    )
)

Rule name must contain the word “bug”.

On the RHS, compute sum incorrectly ...

Observable action: enter incorrect sum

Attach hint templates to production rules

(defrule add-numerators
  (?problem <- (problem (subgoals $? $?)
    (sub $?)
  )
  )
  (?answer <- (answer-fractions ?answer ?))
  (?f1 <- (fraction (has-converted-form ?conv-f1))
    (?conv-f1 <- (fraction (denominator ?conv-denom))
    ?conv-denom <- (textField
      (name ?field-name)
      (value nil)))
  )
  )
  =>
  (predict-observable-action unreducedNum UpdateTextField
    ?sum?
    (modify ?ans (numerator ?ans))
    )
  (modify ?sub (num-done T))
  (construct-message
    [You have two fractions with the same denominator. You can add the numerators to find the numerator of the sum fraction.] )
  (construct-message
    [What is the sum of the numerators ?n1 and ?n2?] )
  (construct-message
    [The sum of the numerators is ?sum. Write ?sum as the numerator in the highlighted cell.] )
)

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.

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)
Back to the organization of working memory

- Why do we represent interface elements in working memory?

A better working memory representation for fraction addition?

True, a more compact working memory representation leads to more compact rules.

Instead of:

(defrule determine-lcd
 ?problem <= (problem
 (subgoals
 (given-fractions ?f1 ?f2)
 (converted-fractions ?f3 ?f4)))

=> (bind ?new-den (lcm ?d1 ?d2))
 (bind ?sub1 (assert (convert-fraction-goal (fraction ?f1) (denominator-value ?new-den))))
 (bind ?sub2 (assert (convert-fraction-goal (fraction ?f2) (denominator-value ?new-den))))
 (bind ?sub3 (assert (add-fractions-goal (fractions ?f3 ?f4))))
 (modify ?problem (subgoals ?sub1 ?sub2 ?sub3)))

Could have:

(defrule determine-lcd
 ?problem <= (problem
 (subgoals
 (given-fractions ?f1 ?f2)
 (converted-fractions ?f3 ?f4)))

=> (bind ?new-den (lcm ?d1 ?d2))
 (bind ?sub1 (assert (convert-fraction-goal (fraction ?f1) (denominator-value ?new-den))))
 (bind ?sub2 (assert (convert-fraction-goal (fraction ?f2) (denominator-value ?new-den))))
 (bind ?sub3 (assert (add-fractions-goal (fractions ?f3 ?f4))))
 (modify ?problem (subgoals ?sub1 ?sub2 ?sub3)))
But what would rules that generate observable actions look like?

(defrule convert-denominator
  ?problem <- (problem (subgoals $? ?sub $?))
  ?sub <- (convert-fraction-goal
    (denom-done nil)
    (fraction $?))
  ?f <- (fraction (has-converted-form ?conv-f))
  ?conv-denom <- (textField (name ?field-name) (value nil)))
=>
  (predict-observable-action
    ?field-name UpdateTextField ?new-den)
  (modify ?conv-denom (value ?new-den))
  (modify ?sub (denom-done T))

Rule relies on the fact that the names of the text fields are stored in working memory.

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

(defrule convert-denominator-1
  ?problem <- (problem (subgoals $? ?sub $?))
  (given-fractions $?))
  ?sub <- (convert-fraction-goal
    (fraction $?)
    (new-denominator ?new-den) (denom-done nil))
  ?f <- (fraction (has-converted-form ?conv-f))
=>
  (predict-observable-action
    convertDenom1 ; selection
    UpdateTextField ; action
    ?new-den) ; input
  (modify ?conv-f (denominator ?new-den))
  (modify ?sub (denom-done T)))

The selection name is 'hard coded.'
Will need two convert-denominator rules, one for each fraction. Would be identical except for the selection.
Also, consider multi-column addition. We don’t want to write an “add” rule for each column separately!

To summarize: 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!

Left-Hand Side - Example pattern constraints

• The two rightmost elements of a list
• 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 $)
• Pairs of duplicate elements ($? ?x1 $? ?x1 $) of a list
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 $?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