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

Vincent Aleven

7th Annual PSLC Summer School

Pittsburgh, July 25 - 29, 2011

CTAT CognitiveTutorAuthoringTools

LearnLab Pittsburgh Science of Learning Center

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

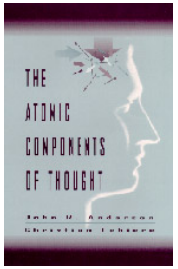
- Retrieve slides, instructions for hands-on work, and example tutors (needed for hands-on activities) from:
- http://ctat.pact.cs.cmu.edu/downloads/pslc2011/
- For starters, download file "Fraction Addition Tutors 2011.zip"
- Additional files may become available during the week

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

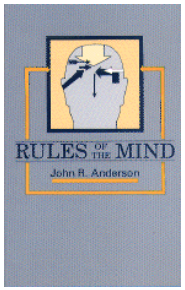
Overview

- **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



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 ≅

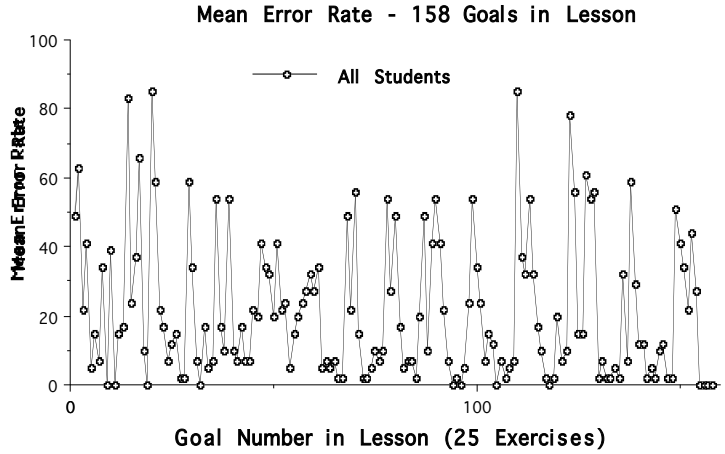
4 Critical Features of Production Rules

- 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

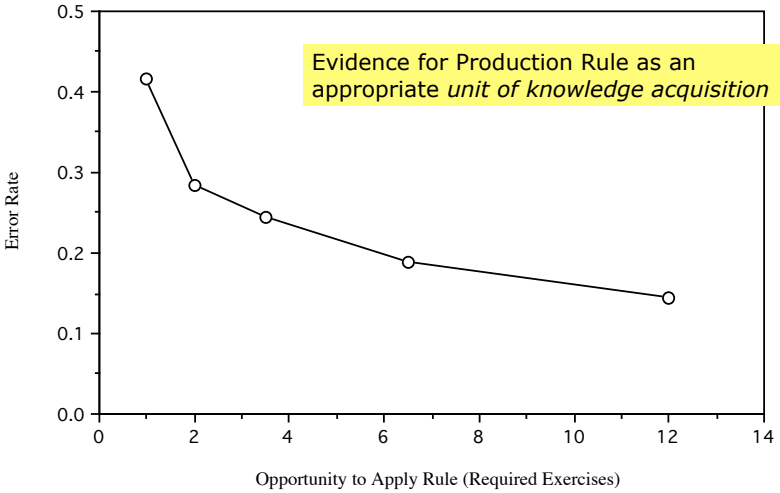
Features 1 & 2 of ACT-R Production Rules

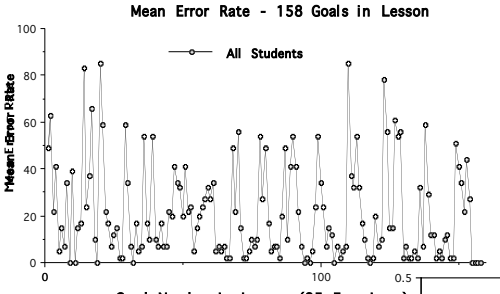
1. Modularity
   - production rules are the units by which a complex skill is acquired
   - empirical evidence: data from the Lisp tutor
2. Abstract character
   - each production rule covers a range of situations, not a single situation
   - variables in the left-hand side of the rule can match different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

Student Performance As They Practice with the LISP Tutor



Production Rule Analysis



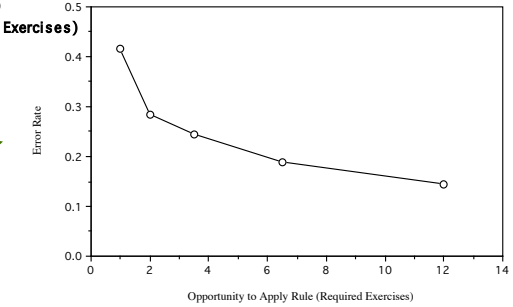


Production Rule Analysis "Cleans Up"

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 ≅

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.

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 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

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

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

Is this a bug in the rules?

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.***

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 turkey
P6. **IF** weighs-15-lbs **AND** produce
THEN watermelon

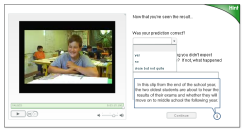
CTAT CognitiveTutorAuthoringTools LearnLab Pittsburgh Science of Learning Center

How ACT-R & Jess production systems are more complex

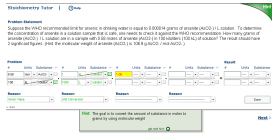
- Watermelon is simple example:
  - *Working memory elements*: a single word
  - *Production rules*: no variables in if-part
  - *Interpreter*: conflict resolution selects lowest numbered unused production
- In contrast, in ACT-R and Jess:
  - *Working memory elements*: database-like record structures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

CTAT CognitiveTutorAuthoringTools LearnLab Pittsburgh Science of Learning Center

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
  - http://pslc-qa.andrew.cmu.edu/~pact/tutors/chemstudy/reviewsite/tutors_study3/tutors_study3.htm

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

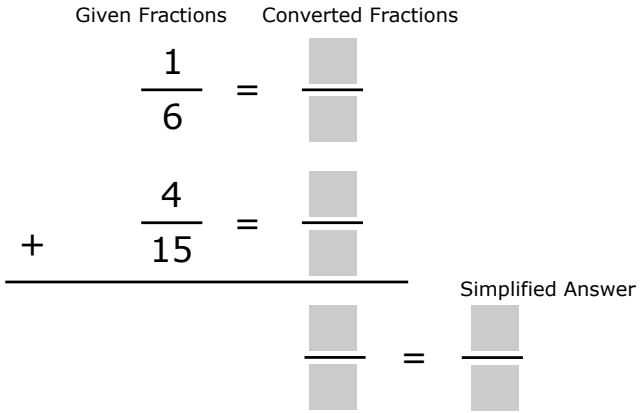
A second production rule model example

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

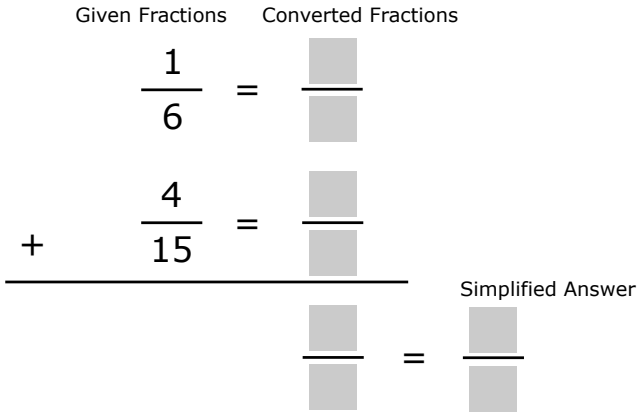
$$1/3 + 2/5 = ?$$

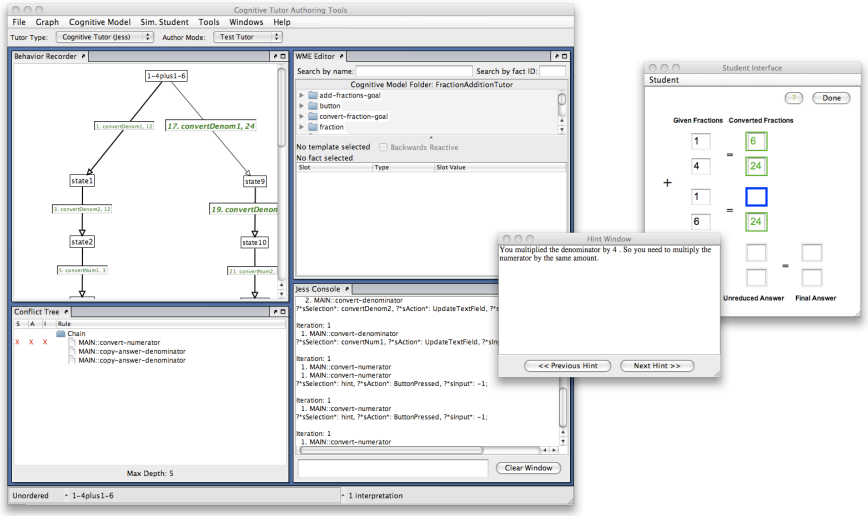
- What if-then rules would you write to perform this task in a step-by-step fashion?

Solution steps for fraction addition



Solution steps for fraction addition





Production rule - skeleton

<RULE-NAME>

"IF part" or "Condition" or "Left-hand side"

IF

There is a (sub)goal to …
In the current state of problem solving is such that …

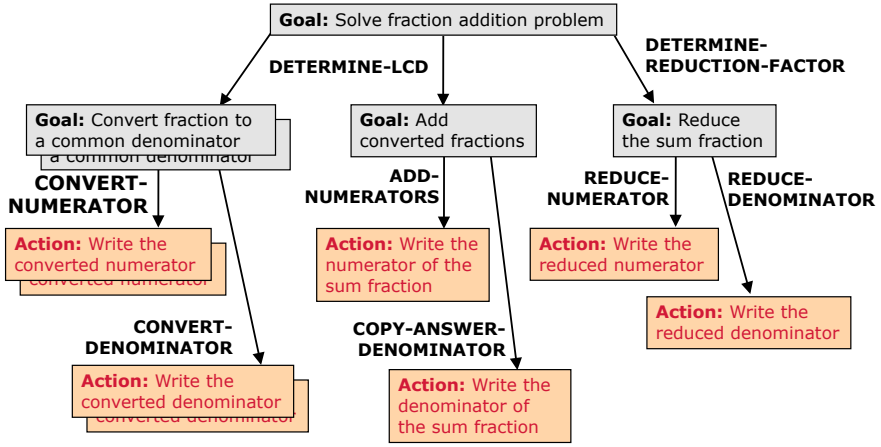
"THEN part" or "Action" or "Right-hand side"

THEN

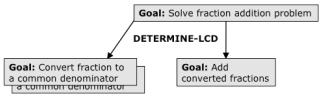
Perform the following (observable) actions: …
Set a subgoal to …
Remove the subgoal to …

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

Production rules set new goals and perform actions



Rules for fraction addition (1)



DETERMINE-LCD

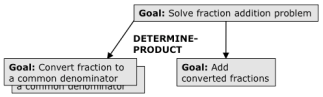
IF there is a fraction addition problem
and there are no subgoals
and ***D1*** is the denominator of the first given fraction
and ***D2*** is the denominator of the second given fraction

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

Names of variables are shown in bold+italics: e.g., ***D1, LCD***

Rules for fraction addition (2)



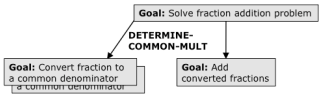
DETERMINE-PRODUCT

IF there is a fraction addition problem
and there are no subgoals
and ***D1*** is the denominator of the first given fraction
and ***D2*** is the denominator of the second given fraction

THEN

Set ***PROD*** to ***D1*** and ***D2***
Set subgoals to convert the fractions to denominator ***PROD***
Set a subgoal to add the converted fractions

Rules for fraction addition (3)



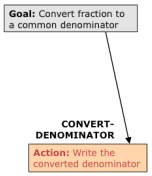
DETERMINE-COMMON-MULT

IF there is a fraction addition problem
and there are no subgoals
and ***D1*** is the denominator of the first given fraction
and ***D2*** is the denominator of the second given fraction

THEN

Set ***MULT*** to ***D1*** and ***D2***
Set subgoals to convert the fractions to denominator ***MULT***
Set a subgoal to add the converted fractions

Rules for fraction addition (4)



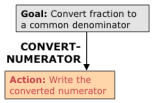
CONVERT-DENOMINATOR

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 ***NUM***
and the denominator of ***F*** is ***DENOM***

THEN

Write ***NUM*** * (***D*** / ***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 ***N1*** + ***N2*** 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

Can you complete these rules?

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 ***N1*** and ***N2*** are the numerators of the two fractions
THEN
Write ***N1*** + ***N2*** 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

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 …

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

DETERMINE-LCD
IF there is a goal to solve a fraction addition problem
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

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

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 ***NUM***
and the denominator of ***F*** is ***DENOM***
THEN
Write ***NUM*** * (***D*** / ***DENOM***) as the numerator of the converted fraction
Mark the subgoal: numerator done

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 ***N1*** and ***N2*** are the numerators of the two fractions
THEN
Write ***N1*** + ***N2*** 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

DETERMINE-REDUCTION-FACTOR
IF the fractions have been added
And there is no subgoal to reduce the fraction
And the answer numerator and denominator have a GCD greater than 1
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 the numerator of the answer fraction has been determined to be ***N***
And the numerator of the reduced answer fraction has not been determined
THEN
Write ***N/F*** as the numerator of the reduced answer fraction

REDUCE-DENOMINATOR
IF there is a subgoal to reduce the answer fraction, with reduction factor ***F***
And the denominator of the answer fraction has been determined to be ***D***
And the denominator of the reduced answer fraction has not been determined
THEN
Write ***D/F*** as the denominator of the reduced answer fraction

DONE
IF One of the answer fractions is filled in and cannot be reduced
THEN
Click Done

Production Rule Firings

1. DETERMINE-LCD

Variable bindings
D1 = 3
D2 = 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
Add subgoal: Add converted fractions

Working memory

$\frac{1}{3} = \frac{\square}{\square}$

$+ \quad \frac{2}{5} = \frac{\square}{\square}$

$\frac{\square}{\square} = \frac{\square}{\square}$

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

Production Rule Firings

1. DETERMINE-LCD

2. CONVERT-DENOMINATOR

Variable bindings
F = first given fraction
D = 15

Changes to working memory
Action: Write 15 as denominator of first converted fraction
Modify subgoal: Mark the denominator part of the subgoal done

Working memory

$\frac{1}{3} = \frac{\square}{15}$

$+ \quad \frac{2}{5} = \frac{\square}{\square}$

$\frac{\square}{\square} = \frac{\square}{\square}$

Subgoals

1. Convert first given fraction
   Denominator part is done
2. Convert second given fraction
3. Add converted fractions

Interpreter, Cycle 2

Match

- CONVERT-NUMERATOR (2x)
- CONVERT-DENOMINATOR (2x)

Conflict Resolution

- Let's say the system selects the *activation* of CONVERT-DENOMINATOR with ***F*** = first given fraction

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.)*

Production Rule Firings

1. DETERMINE-LCD

2. CONVERT-DENOMINATOR

Variable bindings
F = first given fraction
D = 15

Changes to working memory
Action: Write 15 as denominator of first converted fraction
Modify subgoal: Mark the denominator part of the subgoal done

Working memory

$\frac{1}{3} = \frac{\square}{15}$

$+ \quad \frac{2}{5} = \frac{\square}{\square}$

$\frac{\square}{\square} = \frac{\square}{\square}$

Subgoals

1. Convert first given fraction
   Denominator part is done
2. Convert second given fraction
3. Add converted fractions

Interpreter, Cycle 2

Match

- CONVERT-NUMERATOR (2x)
- CONVERT-DENOMINATOR (2x)

Conflict Resolution

- Let's say the system selects the *activation* of CONVERT-DENOMINATOR with ***F*** = first given fraction

Questions

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-NUMERATOR, one for each given fraction.*

Production Rule Firings

1. DETERMINE-LCD

2. CONVERT-DENOMINATOR

3. CONVERT-NUMERATOR

Variable bindings
F= first given fraction
D = 15
NUM = 1
DENOM = 3

Changes to working memory
Action: Write 5 as the numerator of the first converted fraction
Modify subgoal: Mark the numerator part of the subgoal done

Working memory

$\frac{1}{3} = \frac{5}{15}$

$+ \quad \frac{2}{5} = \frac{\square}{\square}$

$\frac{\square}{\square} = \frac{\square}{\square}$

Subgoals

1. Convert first given fraction
   - Denominator part is done
   - Numerator part is done
2. Convert second given fraction
3. Add converted fractions

Interpreter, Cycle 3

Match

- CONVERT-NUMERATOR (2x)
- CONVERT-DENOMINATOR (1x)
- COPY-ANSWER-DENOMINATOR (1x)

Conflict Resolution

- System selects the *activation* of CONVERT-NUMERATOR with ***F*** = first given fraction

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-NUMERATOR

Variable bindings
F= second given fraction
D = 15
NUM = 2
DENOM = 5

Changes to working memory
Action: Write 6 as the numerator of the second converted fraction
Modify subgoal: Mark the numerator part of the subgoal done

Working memory

$\frac{1}{3} = \frac{5}{15}$

$+ \quad \frac{2}{5} = \frac{6}{\square}$

$\frac{\square}{\square} = \frac{\square}{\square}$

Subgoals

1. Convert first given fraction
   - Denominator part is done
   - Numerator part is done
2. Convert second given fraction
   - Numerator part is done
3. Add converted fractions

Interpreter, Cycle 4

Match

- CONVERT-NUMERATOR (1x)
- CONVERT-DENOMINATOR (1x)
- COPY-ANSWER-DENOMINATOR (1x)

Conflict Resolution

- Select *activation* of CONVERT-NUMERATOR

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-NUMERATOR

5. CONVERT-DENOMINATOR

Variable bindings
F = second given fraction
D = 15

Changes to working memory
Action: Write 15 as denominator of second converted fraction
Modify subgoal: Mark the denominator part of the subgoal done

Working memory

$\frac{1}{3} = \frac{5}{15}$

$+ \quad \frac{2}{5} = \frac{6}{15}$

$\frac{\square}{\square} = \frac{\square}{\square}$

Subgoals

1. Convert first given fraction
   - Denominator part is done
   - Numerator part is done
2. Convert second given fraction
   - Numerator part is done
   - Denominator part is done
3. Add converted fractions

Interpreter, Cycle 5

Match

- CONVERT-DENOMINATOR (1x)
- ADD-NUMERATORS (1x)
- COPY-ANSWER-DENOMINATOR (1x)

Conflict Resolution

- Select *activation* of CONVERT-DENOMINATOR

Questions

Production Rule Firings

1. DETERMINE-LCD

2. CONVERT-DENOMINATOR

3. CONVERT-NUMERATOR

4. CONVERT-NUMERATOR

5. CONVERT-DENOMINATOR

6. COPY-ANSWER-DENOMINATOR

Variable bindings
DENOM = 15

Changes to working memory
Action: Write 15 as denominator of the answer fraction
Modify subgoal: Mark the denominator part of the subgoal done

Working memory

$\frac{1}{3} = \frac{5}{15}$

$+ \quad \frac{2}{5} = \frac{6}{15}$

$\frac{\square}{15} = \frac{\square}{\square}$

Subgoals

1. Convert first given fraction
   - Denominator part is done
   - Numerator part is done
2. Convert second given fraction
   - Numerator part is done
   - Denominator part is done
3. Add converted fractions
   - Denominator part is done

Interpreter, Cycle 6

Match

- ADD-NUMERATORS (1x)
- COPY-ANSWER-DENOMINATOR (2x)

Conflict Resolution

- Select one of the *activations* of COPY-ANSWER-DENOMINATOR (does not matter which one, the effect is the same)

Questions

Q: *With different conflict resolution, could the DONE rule fire at this point?*

A: *No, the DONE rule requires that one of the two answer fractions is fully filled in.*

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

Variable bindings
N1 = 5
N2 = 6

Changes to working memory
Action: Write 11 as numerator of the answer fraction
Modify subgoal: Mark the numerator part of the subgoal done

Working memory

$\frac{1}{3} = \frac{5}{15}$

$+ \quad \frac{2}{5} = \frac{6}{15}$

$\frac{11}{15} = \frac{\square}{\square}$

Subgoals

1. Convert first given fraction
   Denominator part is done
   Numerator part is done
2. Convert second given fraction
   Numerator part is done
   Denominator part is done
3. Add converted fractions
   Denominator part is done
   Numerator part is done

Interpreter, Cycle 7

Match
- ADD-NUMERATORS (1x)

Conflict Resolution
- Select *activation* of ADD-NUMERATORS

Questions

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

Variable bindings
N/A

Changes to working memory
Action: Click Done

Working memory

$\frac{1}{3} = \frac{5}{15}$

$+ \quad \frac{2}{5} = \frac{6}{15}$ ✓

$\frac{11}{15} = \frac{\square}{\square}$

Subgoals

1. Convert first given fraction
   Denominator part is done
   Numerator part is done
2. Convert second given fraction
   Numerator part is done
   Denominator part is done
3. Add converted fractions
   Denominator part is done
   Numerator part is done

Interpreter, Cycle 8

Match
- DONE (1x)

Conflict Resolution
- Select *activation* of DONE

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

$\frac{1}{3} = \frac{5}{15}$

$+ \quad \frac{2}{5} = \frac{6}{15}$ ✓

$\frac{11}{15} = \frac{\square}{\square}$

Subgoals

1. Convert first given fraction
   Denominator part is done
   Numerator part is done
2. Convert second given fraction
   Numerator part is done
   Denominator part is done
3. Add converted fractions
   Denominator part is done
   Numerator part is done

Interpreter, Cycle 9

Match
- No rules match

Conflict Resolution

Questions

CTAT CognitiveTutorAuthoringTools

LearnLab Pittsburgh Science of Learning Center

Summary

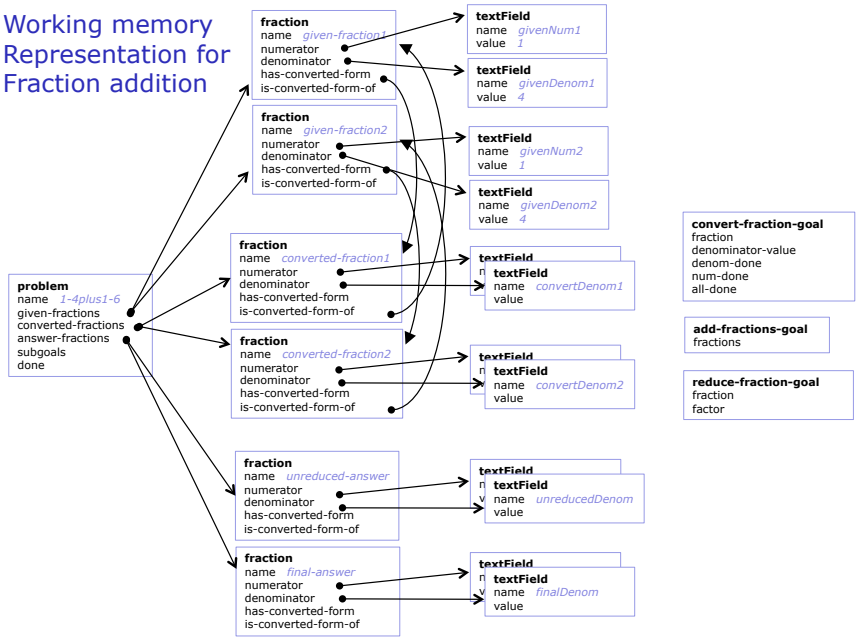
- Components of Production Systems:
  - Working memory, production memory, interpreter
  - Steps in the interpreter: Match, conflict resolution, fire
- Features of cognition explained by ACT-R production rules:
  - Procedural knowledge:
    - modular, limited generality, goal structured, asymmetric
  - Declarative knowledge:
    - flexible, verbal or visual, less efficient

Overview

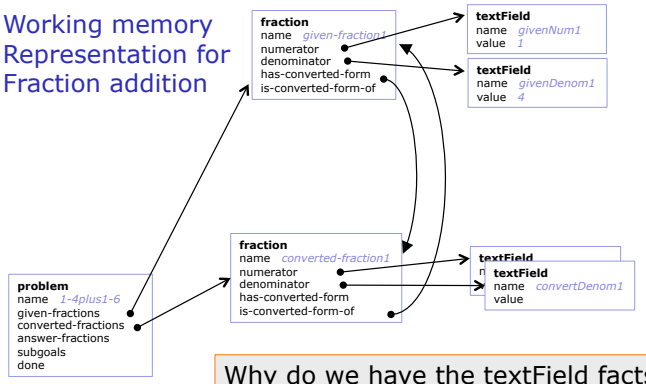
- ACT-R theory
  - Features of production rules and their predictions about learning
- How Production Systems Work
  - A simple example
  - A more complex example: fraction addition
- **Jess Production System Notation**
  - **Working memory: templates and facts**
  - **Production rule notation**
- Model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

- Associated reading for Lecture 4: Friedman-Hil, E. (2003). *Jess in Action: Java Rule-based Systems.* Greenwich, CT: Manning Publications. Focus on: Chapter 2 (skip if familiar with rule-based systems), 3, 4-4.2, 6-6.4, 7-7.3, 7.5.

Working memory Representation for Fraction addition



Working memory Representation for Fraction addition



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

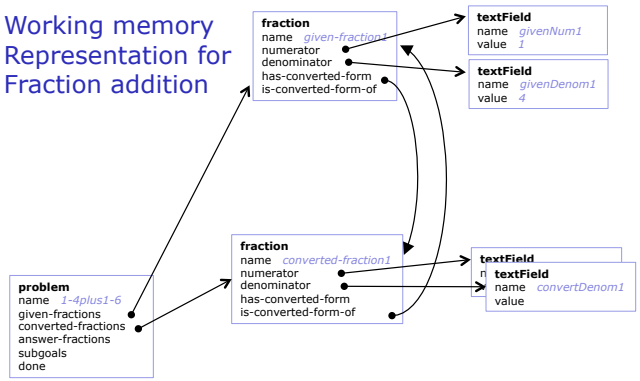
A *template* defines a type of fact and the slots that belong to the type:

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

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

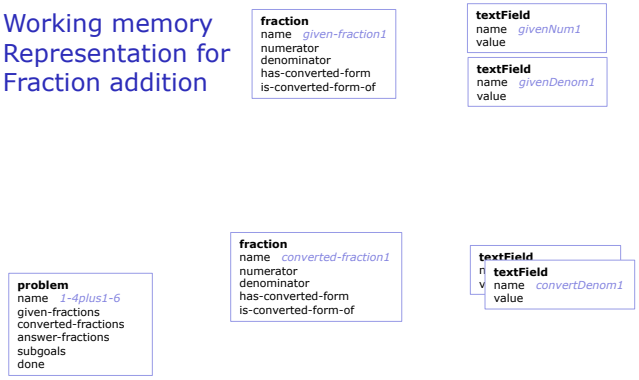
```
(deftemplate MAIN::textField
   (slot name)
   (slot value))
```


Working memory Representation for Fraction addition



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

Working memory Representation for Fraction addition



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

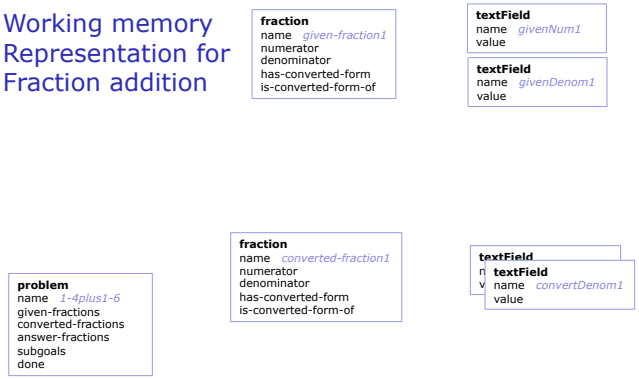
Bind creates a new variable and gives it a value

```
;; 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))))
```

All facts have a *name* slot. Names are useful if they help the author. Otherwise, they are not needed.

Exception: the facts that represent the interface should have the same name as the corresponding component/widget.

Working memory Representation for Fraction addition



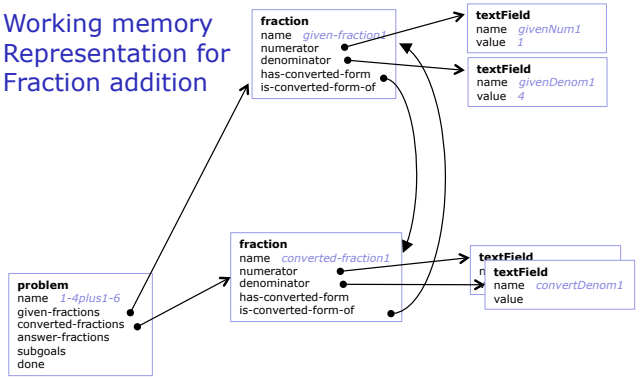
Create the facts in two steps:
1. Create each fact
2. Fill in values and cross-references

Creating facts in working memory

Modify changes a fact in working memory (but you need to be able to reference the fact – hence the use of variables)

```
;; Fraction facts point to each other and to the textField facts
(modify ?var22    ;; given-fraction1
    (numerator ?var8)
    (denominator ?var9)
    (has-converted-form ?var24))

(modify ?var24    ;;  converted-fraction1
    (numerator ?var14)
    (denominator ?var15)
    (is-converted-form-of ?var22))
```


Working memory Representation for Fraction addition



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

Creating facts in working memory

```
;; Fill in given values in the relevant TextField facts
(modify ?var8   ;;  givenNum1
   (value 1))

(modify ?var9   ;;  givenDenom1
   (value 4))


;; Problem fact, with links to all six fraction facts
(modify ?var21
        (given-fractions ?var22 ?var23)
        (converted-fractions …)
        (answer-fractions ... ))
```

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

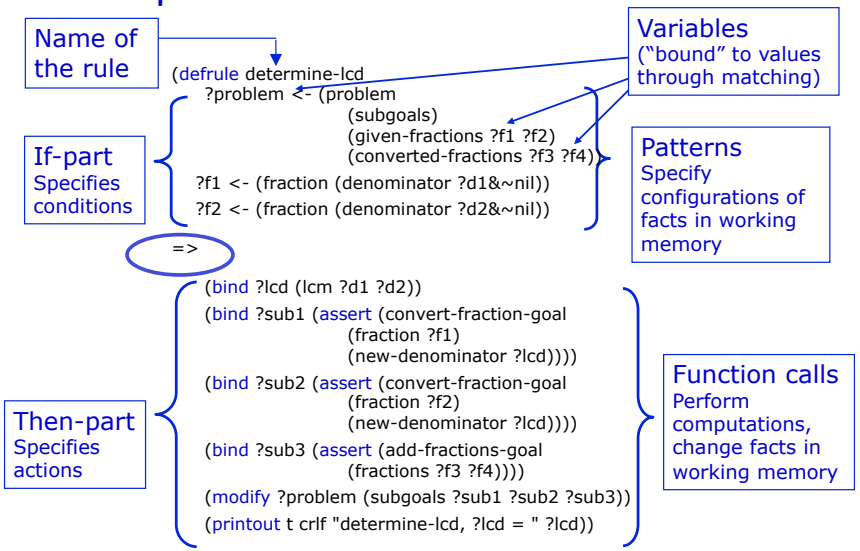
DETERMINE-LCD

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))
   ?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



Jess production rule for first cycle

English (elaborated)

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 ?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 for first cycle

English (elaborated)

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 ***DENOM1*** is the denominator of ***F1***
and ***DENOM2*** is the denominator of ***F2***
and ***D1*** is the value of ***DENOM1*** and is not *nil*
and ***D2*** is the value of ***DENOM2*** and is not *nil*
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

Key condition: no subgoals

```
(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)
                  (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)))
```


Jess production rule for first cycle

English (elaborated)

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 ***DENOM1*** is the denominator of ***F1***
and ***DENOM2*** is the denominator of ***F2***
and ***D1*** is the value of ***DENOM1*** and is not *nil*
and ***D2*** is the value of ***DENOM2*** and is not *nil*
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

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

```
(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)
                  (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))))
```


Jess production rule for second cycle

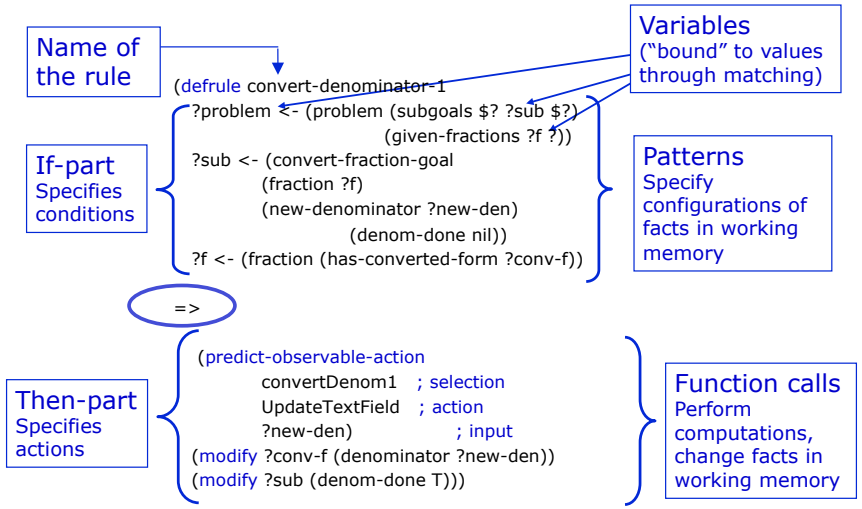
English

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 $? ?sub $?)
                       (given-fractions ?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)))
```

Jess production rule notation



Jess production rule for second cycle

Key condition: convert fraction subgoal whose denominator part is not done

English (elaborated)

CONVERT-DENOMINATOR

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

THEN
Write ***NEW-DEN*** as the value in field ***FIELD-NAME***
Set the value of ***CONV-DENOM*** to ***NEW-DENOM***
Make a note that the denominator part of the subgoal is done

Jess

```
(defrule convert-denominator
  ?problem <- (problem (subgoals $? ?sub $?))
  ?sub <- (convert-fraction-goal
            (denom-done nil)
            (fraction ?f)
            (denominator-value
                  ?new-den&:(neq ?new-den nil)))
  ?f <- (fraction (has-converted-form ?conv-f))
  ?conv-f <- (fraction (denominator ?conv-denom))
  ?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)))
```


Jess production rule for second cycle

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

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

English (elaborated)

CONVERT-DENOMINATOR

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

THEN
Write ***NEW-DEN*** as the value in field ***FIELD-NAME***
Set the value of ***CONV-DENOM*** to ***NEW-DENOM***
Make a note that the denominator part of the subgoal is done

Jess

```
(defrule convert-denominator
  ?problem <- (problem (subgoals $? ?sub $?))
  ?sub <- (convert-fraction-goal
            (denom-done nil)
            (fraction ?f)
            (denominator-value
                  ?new-den&:(neq ?new-den nil)))
  ?f <- (fraction (has-converted-form ?conv-f))
  ?conv-f <- (fraction (denominator ?conv-denom))
  ?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)))
```


Matching a production rule against working memory

Production Rule

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

Working Memory

```
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))
f-1  (MAIN::fraction (name giv1)
                     (numerator 1)
                     (denominator 3)
                     (has-converted-form <Fact-3>)
                     (is-converted-form-of nil))
f-2  (MAIN::fraction (name giv2)
                     (numerator 2)
                     (denominator 5)
                     (has-converted-form <Fact-4>)
                     (is-converted-form-of nil))
f-3  (MAIN::fraction (name conv1) … )
f-4  (MAIN::fraction (name conv2) … )
f-5  (MAIN::fraction (name sum) … )
f-6  (MAIN::fraction (name final) … )
```

Find value for each variable

?f1	<Fact-1>
?f2	<Fact-2>
?f3	<Fact-3>
?f4	<Fact-4>
?problem	<Fact-7>
?d1	3
?d2	5
?lcd	15
?sub1	<Fact-8>
?sub2	<Fact-9>
?sub3	<Fact-10>

What changes are made to working memory?

```
==> f-8 (MAIN::convert-fraction-goal
assert        (fraction <Fact-1>)
              (new-denominator 15)
              (denom-done nil)
              (num-done nil))

==> f-9 (MAIN::convert-fraction-goal
assert        (fraction <Fact-2>)
              (new-denominator 15)
              (denom-done nil)
              (num-done nil))

==> f-10 (MAIN::add-fractions-goal
assert         (fractions <Fact-3> <Fact-4>)
               (denom-done nil)
               (num-done nil))

<== f-7 (MAIN::problem (name 1-3plus2-5)
modify         (given-fractions <Fact-1> <Fact-2>)
               (converted-fractions <Fact-3> <Fact-4>)
               (answer-fractions <Fact-5> <Fact-6>)
               (subgoals <Fact-8> <Fact-9> <Fact-10>)
               (done nil))
```

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

```
?problem <- (problem
                (subgoals)
                (given-fractions ?f1 ?f2)
                (converted-fractions ?f3 ?f4))
?f1 <- (fraction (denominator ?d1&~nil))
```

- THEN-part: computations and changes to working memory

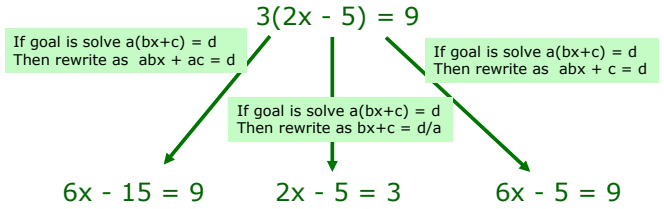
```
(bind ?lcd (lcm ?d1 ?d2))
(bind ?sub3 (assert (add-fractions-goal (fractions ?f3 ?f4))))
(modify ?problem (subgoals ?sub1 ?sub2 ?sub3))
```


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

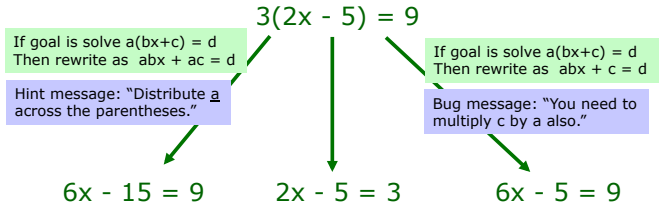
- Cognitive Model: A system that can solve problems in the various ways students can



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

Model tracing:
Use cognitive model 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.

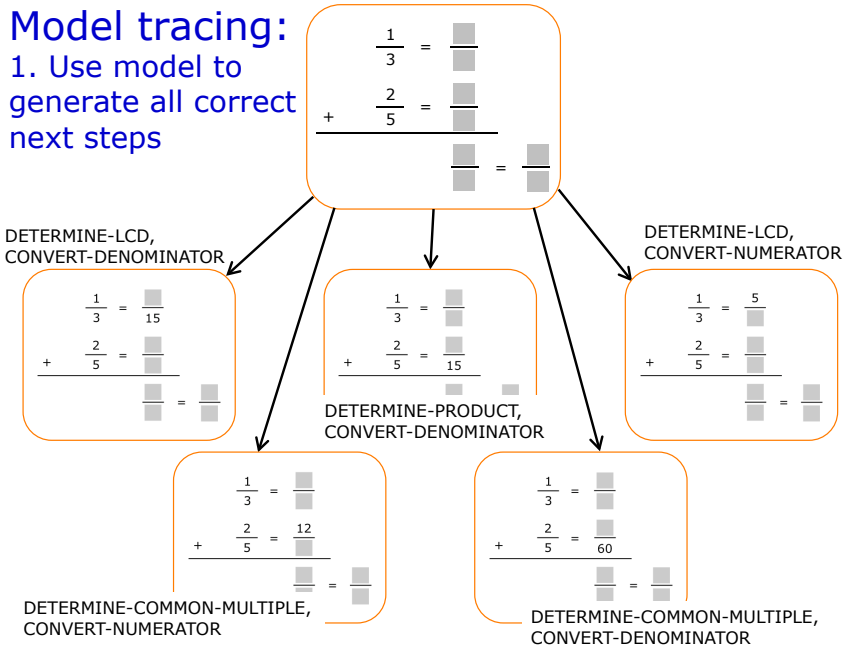
Model tracing algorithm (simplified)

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 in sync)
3. Otherwise, error. Provide negative feedback (working memory unchanged)

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

Model tracing:
1. Use model to generate all correct next steps



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"

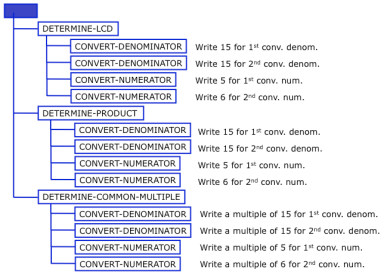
- DETERMINE-LCD
  - CONVERT-DENOMINATOR — Write 15 for 1st conv. denom.
  - CONVERT-DENOMINATOR — Write 15 for 2nd conv. denom.
  - CONVERT-NUMERATOR — Write 5 for 1st conv. num.
  - CONVERT-NUMERATOR — Write 6 for 2nd conv. num.
- DETERMINE-PRODUCT
  - CONVERT-DENOMINATOR — Write 15 for 1st conv. denom.
  - CONVERT-DENOMINATOR — Write 15 for 2nd conv. denom.
  - CONVERT-NUMERATOR — Write 5 for 1st conv. num.
  - CONVERT-NUMERATOR — Write 6 for 2nd conv. num.
- DETERMINE-COMMON-MULTIPLE
  - CONVERT-DENOMINATOR — Write a multiple of 15 for 1st conv. denom.
  - CONVERT-DENOMINATOR — Write a multiple of 15 for 2nd conv. denom.
  - CONVERT-NUMERATOR — Write a multiple of 5 for 1st conv. num.
  - CONVERT-NUMERATOR — Write a multiple of 6 for 2nd conv. num.

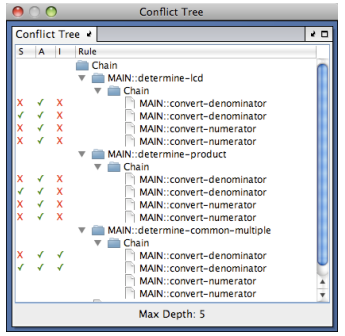
$$\frac{1}{3} = \frac{\square}{\square}$$

$$+ \quad \frac{2}{5} = \frac{\square}{\square}$$

$$\frac{\square}{\square} = \frac{\square}{\square}$$

CTAT's Conflict Tree window: Debugging tool





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

Model tracing algorithm (simplified)

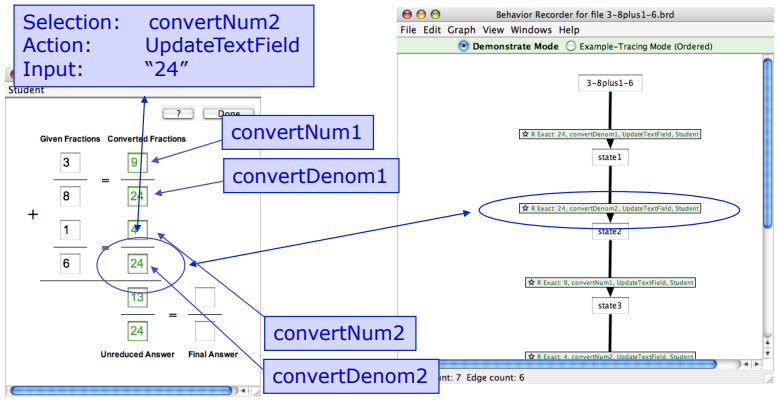
After a student action:

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, error. Provide negative feedback and leave working memory unchanged)

How are student actions compared against the actions predicted by the model?

In CTAT, student actions are encoded as Selection/Action/Input triples

Selection: convertNum2
Action: UpdateTextField
Input: "24"



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 ?f)
                (denominator-value
                        ?new-den&:(neq ?new-den nil)))
        ?f <- (fraction (has-converted-form ?conv-f))
        ?conv-f <- (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.

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 BUG-add-denominators
  ?problem <- (problem
                (given-fractions ?f1 ?f2)
                (answer-fractions ?answer ?))
  ?f1 <- (fraction (denominator ?den1))
  ?f2 <- (fraction (denominator ?den2))
  ?den1 <- (textField (value ?d1))
  ?den2 <- (textField (value ?d2))
  ?answer <- (fraction (denominator ?ans-denom))
  ?ans-denom <- (textField (value nil)(name ?name))
=>
  (bind ?sum (+ ?d1 ?d2))
  (predict-observable-action
                ?name               ; selection
                UpdateTextField     ; action
                ?sum)               ; input
  (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

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-numerators
  ?problem <- (problem
                (subgoals $? ?sub $?)
                (answer-fractions ?answer ?))
  ?sub <- (add-fractions-goal
            (fractions ?f1 ?f2)(num-done nil))
  ?f1 <- (fraction (numerator ?n1&~nil))
  ?f2 <- (fraction (numerator ?n2&~nil))
=>
  (bind ?sum (+ ?n1 ?n2))
  (predict-observable-action unreducedNum UpdateTextField ?sum)
  (modify ?answer (numerator ?sum))
  (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. ]"
     "[What is the sum of the numerators " ?n1 " and " ?n2 "?]"
     "[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.

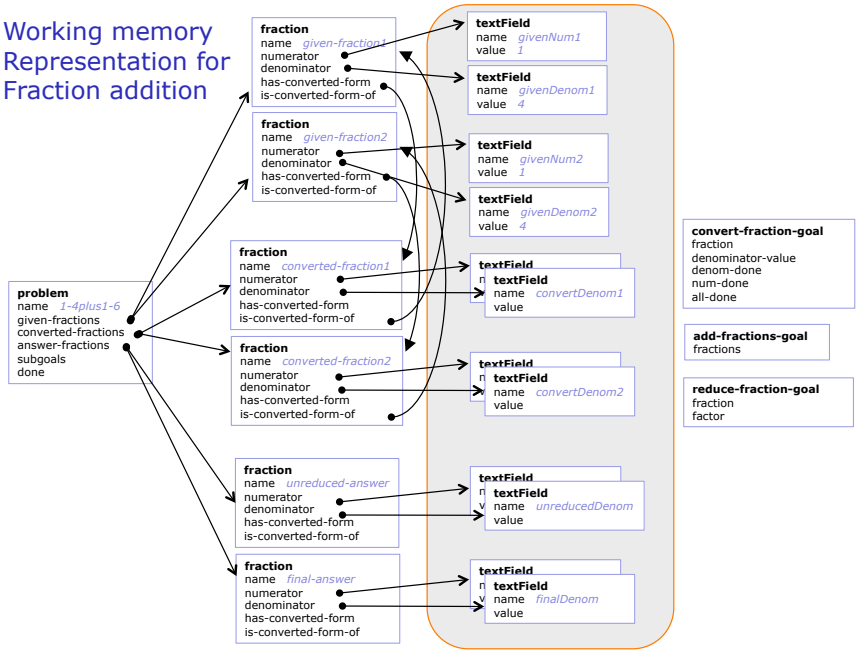
- Each expression in [] is next hint level
- Can insert variables
- Put text (including []) in double quotes.

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.

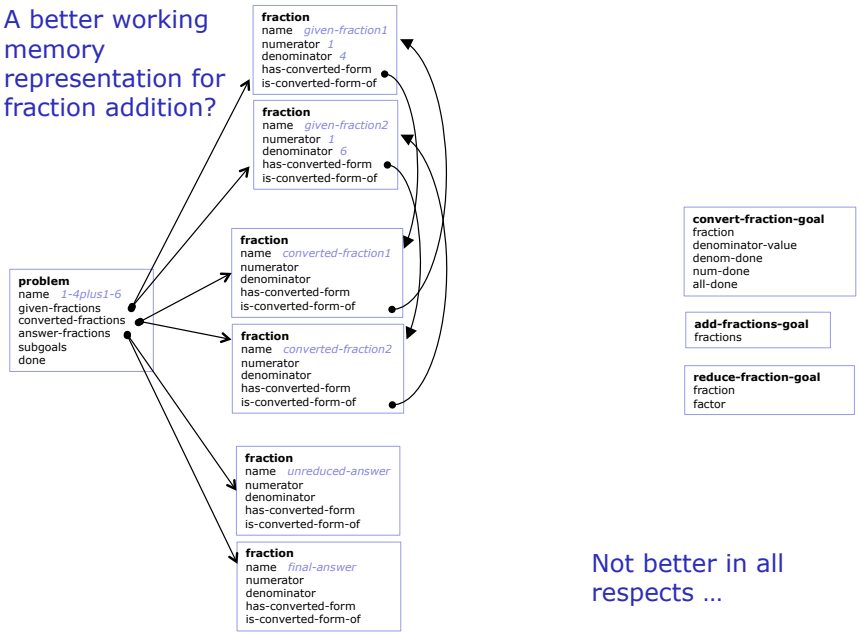
Back to the organization of working memory

- Why do we represent interface elements in working memory?

Working memory Representation for Fraction addition



A better working memory representation for fraction addition?



Not better in all respects …

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))
   ?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)
                    (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))
   ?f1 <- (fraction (denominator ?d1))
   ?f2 <- (fraction (denominator ?d2))
=>
   (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)
         (denominator-value ?new-den&:(neq ?new-den nil)))
   ?f <- (fraction (has-converted-form ?conv-f))
   ?conv-f <- (fraction (denominator ?conv-denom))
   ?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 ?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)))
```

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 — ($? ?second-col ?first-col)
- The two rightmost elements in a list of 3 — (? ?second-col ?first-col)
- Any adjacent pair of list elements — ($?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