Cognitive tutors
Recap: Intelligent tutoring systems
Cognitive tutors
LISP
The Lisp tutor
  What is LISP?
  Components
  Structured editor
  Practical example
  Conclusions in the article
Limitations of cognitive tutors
Intelligent tutoring systems

- Domain model
- Student model
- Tutoring model
- Interface component
Cognitive tutor

- Intelligent tutoring system
- Uses cognitive model (model tracing, knowledge tracing)
- Step-by-step guidance
- Monitors student's performance and learning
- Context-specific hints and problem-solving activities
- Blended learning
Cognitive tutors support:

- problem-solving environment
- step-by-step feedback on student performance
- feedback messages specific to errors
- context-specific next-step hints at student’s request
- individualized problem selection
LISP ( list processing )

- Expression oriented programming language
- Was one of the main programming languages of AI
- No distinction between expressions and statements
- Expressions as lists, using prefix notation
  - (+ a (* b c)) stands for a + b*c
- Symbolic nature
- Use of recursion
- Abstract AI programming
LISP example

(defun m-length (list)
  (if (null list)
      0
      (+ 1 (m-length (cdr list)))))

(defun print-list (list)
  (if (not (null list))
      (progn
        (princ (car list))
        (print-list (cdr list))))))

(((5 x o 1) a) 7 2 3)

(defun sum-numbers (list)
  (if
LISP tutor

- **Goal**
- **Components:**
  - Domain model (programming knowledge and misconceptions)
  - Planning mode with examples
  - Coding mode
  - **Mistake -> immediate feedback**
  - **Hints**
    - On request
    - Maximum allowed errors
  - **Structured editor**
Structured editor

- Balanced parentheses
- Placeholders for functions
- Highlighted areas
- Students focus on context not syntax

```
(defun <NAME> <PARAMETERS>
  <PROCESS>
)
```
Listing 1: The dialogue between student and computer tutor to define the function rightp.

Define rightp. It takes 3 arguments, side1, side2, and side3, which represent the lengths of the three sides of a given triangle, where side3 is the longest side. rightp tests whether the three sides form a right triangle. In a right triangle, the sum of the squares of side1 and side2 is equal to the square of side3. However, your function should allow for a measurement error of 2 percent. In other words, your function should return t if the sum of the squares of the first two sides is within 2 percent of the square of the third side.

For example (rightp 3 4 5.01) = t, because 3 squared + 4 squared = 25, which is within 2 percent of 5.01 squared.

1. (defun rightp (side1 side2 side3) (cond))
   Yes, you could use cond here, but there is a much easier way to test if the sum of the squares of the first two sides is within 2 percent of the square of the third. You can use a predicate, since we only have one test to do.

2. (defun rightp (side1 side2 side3))
   (lesss (difference (square <NUM2>))
   You want to code the amount of difference between the sum of the squares of the first two sides and the square of the third side, and you do not care which of those two numbers is larger. So we want to take the absolute value of their difference.

3. (defun rightp (side1 side2 side3))
   (lesss (abs (difference <NUM3> <NUM4>)) <NUM2>)
   Which do you want to do first?
   PRESS: IF YOU WANT TO:
   1 Write code to get the sum of the squares of the first two sides.
   2 Write code to get the square of the third side.
   Menu choice: 1

4. (defun rightp (side1 side2 side3)
   (lesss (abs (difference (plus) <NUM4>))
   <NUM2>)
   (continued)
LISP tutoring
Listing 2: The dialogue involved in defining the function fact, which computes the factorial of a number given.

Define the function fact. It takes a number argument, n, and computes the factorial of n, which is the result of multiplying n x n-1 x n-2 ... x 1. For instance, (fact 3) = 3 x 2 x 1 = 6. By definition, the factorial of 0 is 1.

1. (defun fact (n) (cond (equal n 0) (return 1) (recurse-case) (return (* n (fact (- n 1))))))

I assume that you are trying to test if the number argument is equal to the number zero. There is a special function for testing equivalence to zero. Try to use that function instead of equal.

What code can you write to test if the condition for terminating case is true? Try writing a predicate.

2. (defun fact (n) (cond (zero n) (return 1) (recurse-case) (return (* n (fact (- n 1))))))

Did you mean zero? (y or n)? y

3. (defun fact (n) (cond (zerop n) (return 1) (recurse-case) (return (* n (fact (- n 1))))))

What is the value of fact in the recursive cases calculated by:

Press: IF YOU WANT TO:
1. Calling fact recursively with n.
2. Multiplying n by fact of n minus 1.
3. Getting 1 x 2 x ... n and then multiplying the results.
4. Have the tutor choose.

Menu choice: 3

That is a reasonable way to think of doing factorials but it is not a plan for a recursive function. Since you seem to be having trouble with the recursive cases let us work through some examples and figure out the conditions and actions for each of these cases.

The recursive relationship is the relationship between fact called with n and fact called with:

Press: IF YOU WANT TO:
1. n minus 1
2. n equal to 1
3. n plus 1
4. Have the tutor choose.

Menu choice: 1

We will consider some concrete examples. We will characterize the recursive relationship between fact called with 1 and fact called with 3.

What is the value of (fact 1)?
Answer: 1

What is the value of (fact 3)?
Answer: 1

Examples:

fact (1) = 1 (fact 0) = 1

We will consider some more concrete examples. We will characterize the recursive relationship between fact called with 3 and fact called with 2.

What is the value of (fact 2)?
Answer: 2

What is the value of (fact 2)?
Answer: 2

Examples:

fact (3) = 3 x 2 x 1 = 6 

Since there is only one recursive case characterized by the examples you have just worked out, the condition in this case can be just coded as 1. We will focus on characterizing the action in this case.

In examples A and B what do you have to do to get the result of fact called with n?

Press: IF YOU WANT TO:
1. Multiply it by one less than n.
2. Multiply it by fact of one less than n.
3. Add n to the result of fact called with one less than n.
4. Have the tutor choose.

Menu choice: 2

We have now worked through a series of examples that illustrate the recursive relation. We will now return to the goal of trying to produce LISP code that specifies the recursive relation.

4. (defun fact (n) (cond (zerop n) (return 1) (return (* n (fact (- n 1))))))

You will have to use fact at some point but right now you want to multiply.

5. (defun fact (n) (cond (zerop n) (return 1) (return (* n (fact (- n 1))))))

What do you want to do first?
Press: IF YOU WANT TO:
1. Write code to get the number argument.
2. Write code to get the recursive part.

Menu choice: 1

6. (defun fact (n) (cond (zerop n) (return 1) (times n (fact (1)))))

You cannot just code n at this point. What you want to do is subtract one from n.

7. (defun fact (n) (cond (zerop n) (return 1) (times n (fact (1)))))

The difference would work in this case but it is more appropriate to use the function sub when you are subtracting 1 from a number.

8. (defun fact (n) (cond (zerop n) (return 1) (times n (fact (sub n)))))

(continued)
Conclusions in the article

- To many menus
- Sometimes slow
- A lot of computer power for that time
- Students more appreciative of the guidance once the difficulty increases
- Human tutor > LISP tutor > learning on your own
Limitations of Cognitive tutors

- Curriculum
  - Mostly limited to algebra, computer programming, geometry
  - Creating them for other subjects is not practical or economical

- Design
  - Complex
  - Time consuming to create
  - Limited choices of methods, prompts and hints
  - Hints used prematurely

- Conclusion
  - Human tutors still outperform them (more responsive, better feedback, scaffolding)
Thank you for your attention

Any questions?