UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Rule-Based Systems

Alfons Juan
Albert Sanchis
Jorge Civera

DSIC

Departament de Sistemes Informàtics i Computació

Objectives

- To know the basics of rule-based systems (RBS).
- To run a trace of a RBS based on CLIPS.

Contents

1 Introduction 3

2 Rule-Based Systems with CLIPS 4

3 The 8-puzzle problem 6

4 Facts and rules 8

5 Inference Engine 16

1 Introduction

- RBS can represent the expert knowledge on a problem
- States of the problem represented by ***facts***
- Possible actions on states represented as ***rules***
- Rule:

```
if conditions x, y, z are satisfied,
then actions a, b, c can be executed
```

- ***Inference engine***:
  - checks which rules are ready to be executed on available states
  - selects which one of the ready rules will be finally executed
  - executes the selected rule generating new state(s)
- Eventually the goal state is generated

2 Rule-Based Systems with CLIPS

CLIPS is a tool to build RBS with three components:

Facts:

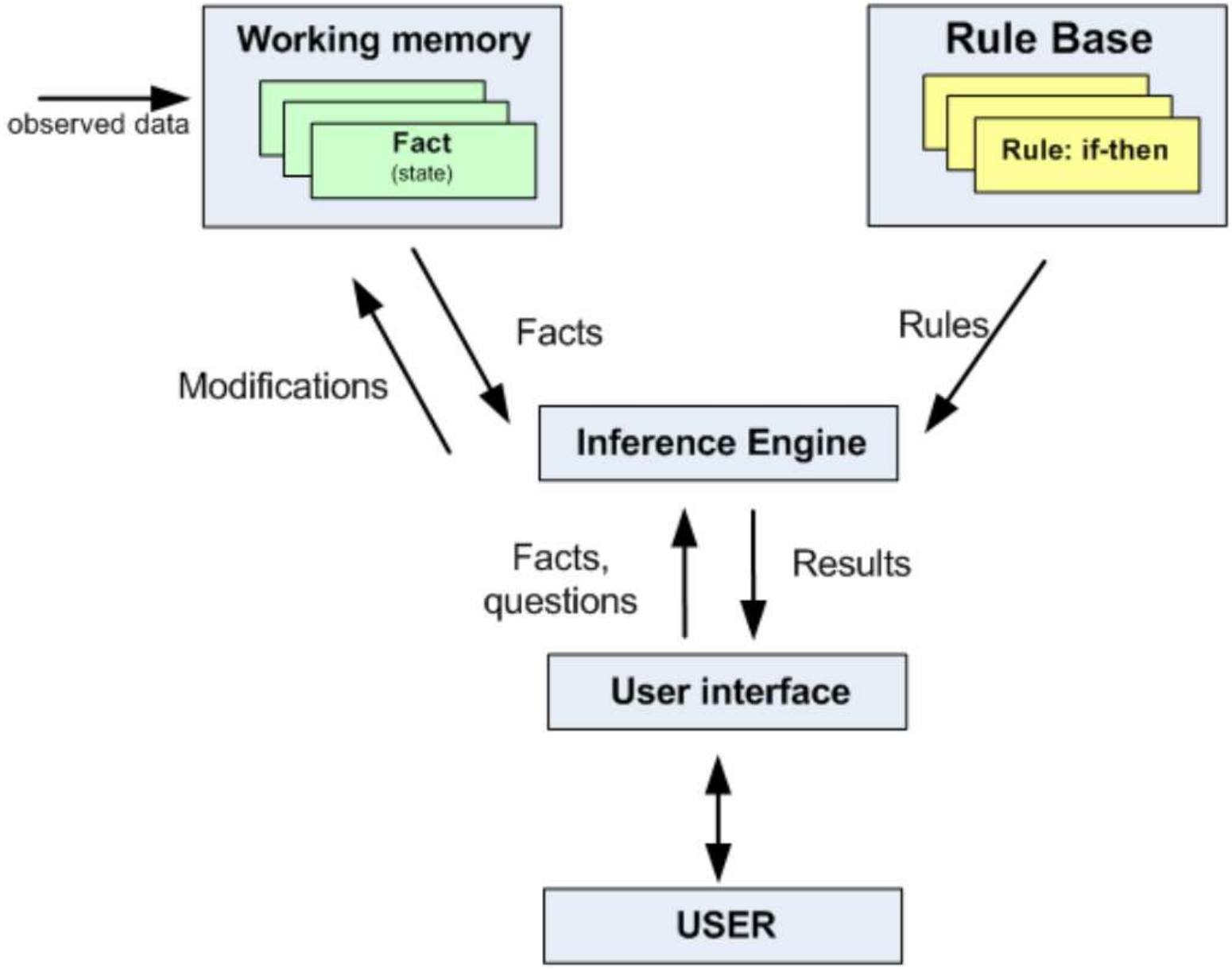
- Every state of the problem is defined as a single fact according to a fact pattern previously defined.
- After each step execution, facts represent states of the problem already explored or to be explored.
- The rest of facts are ***static*** information of the problem.

Rules:

- Every applicable action to one or more states of the problem. It is usually represented by a single rule `left => right`.
- The left-hand side (LHS) selects the set of elegible states.
- The right-hand side (RHS) usually adds new facts.

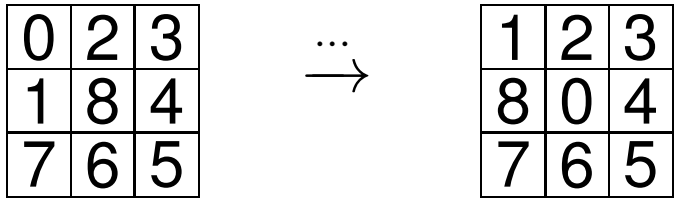
Inference engine: Rule instantiation, selection and execution.

Rule-Based Systems with CLIPS



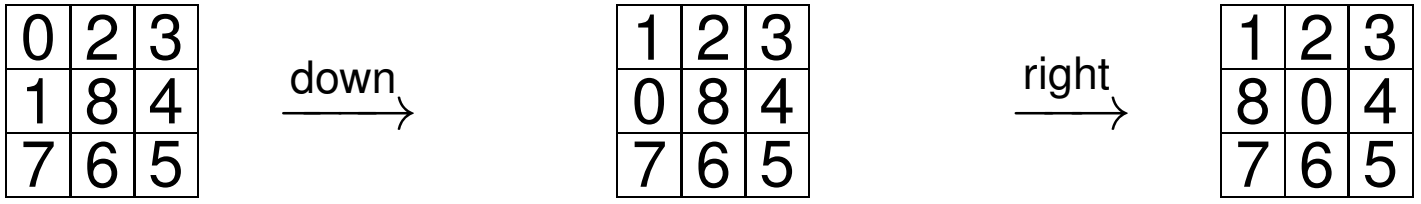
3 The 8-puzzle problem

Provided an initial state, a goal state has to be reached by movements of the blank tile (tile 0): right, left, up and down.



Initial state Goal

Solution for the previous initial and goal states:



Initial state Intermediate state Goal state

A simple RBS for the 8-puzzle problem

```
(deffacts bhini (puzzle 0 2 3 1 8 4 7 6 5))

(defrule left
  (puzzle $?x ?y 0 $?z)
  (test (<> (length$ $?x) 2))
  (test (<> (length$ $?x) 5)) =>
  (assert (puzzle $?x 0 ?y $?z)))

(defrule right
  (puzzle $?x 0 ?y $?z)
  (test (<> (length$ $?x) 2))
  (test (<> (length$ $?x) 5)) =>
  (assert (puzzle $?x ?y 0 $?z)))

(defrule up
  (puzzle $?x ?a ?b ?c 0 $?y) =>
  (assert (puzzle $?x 0 ?b ?c ?a $?y)))

(defrule down
  (puzzle $?x 0 ?a ?b ?c $?z) =>
  (assert (puzzle $?x ?c ?a ?b 0 $?z)))

(defrule goal
  (puzzle 1 2 3 8 0 4 7 6 5) =>
  (printout t "Solution found!" crlf)
  (halt))
```

4 Facts and rules

- Facts can be statically defined as:

```
(deffacts <name> [<comment>] <fact>*)
```

  where the syntax of each fact is:

```
<fact> ::= (<symbol> <constant>*)
```

- Example:

```
(deffacts bhini (puzzle 0 2 3 1 8 4 7 6 5))
```

- Rule syntax is:

```
(defrule <name> <LHS> => <RHS>)
```

  where the Left-Hand Side ***(LHS)*** of a rule is:

```
LHS ::== <conditional-element>*
```

  and the Right-Hand Side ***(RHS)*** of a rule is:

```
RHS ::= <action>*
```

LHS syntax

- ***Conditional elements (CEs)*** must be all satisfied (evaluated to True), so that the rule instance is ready to be executed.

```
<conditional-element> ::=
  <ordered-pattern> | <assigned-pattern> | <test>
  | <or CE> | <and CE> | <not CE>
```

- Most common types of ***<conditional-element>***:
  - ***<ordered-pattern>:*** Selection of facts that ***match*** the pattern
  - ***<assigned-pattern>:*** Assigns matching fact index to variable
  - ***<test>:*** Evaluation of variables instantiated in pattern matching
  - ***<or CE>:*** CEs separated by ***or*** logic operators
  - ***<and CE>:*** CEs separated by ***and*** logic operators
  - ***<not CE>:*** CE prefixed by the ***not*** operator

Ordered pattern

- ***<ordered-pattern>:*** list starting with an initial symbol, followed by constants, variables and/or wildcards

```
<ordered-pattern> ::= (<symbol> <constraint>*)
<constraint> ::= <constant> | <variable> | ? | $?
<constant> ::= <symbol> | <string> | <integer> | <float>
<variable> ::=  <single-vble> | <multi-vble>
<single-vble> ::= ?<symbol>
<multi-vble> ::= $?<symbol>
```

- Example:

```
(puzzle $?x ?y 0 $?z)
```

  where

  - ***$?x*** is a multi-valued variable matching zero o more elements
  - ***?y*** is a single-valued variable matching exactly one element
  - ***0*** is a constant

- ***wildcards ?*** and ***$?*** matching behavior is the same as variables, but matched values are not stored

Pattern matching

- Possible matchings between patterns and facts

pattern	fact	Single match
pattern	fact 1 fact 2	Same pattern matches once with different facts
pattern 1 pattern 2	fact	Different patterns match the same fact
pattern	fact	Several matches due to multi-valued variables

- *`<ordered-pattern>`: (list $?x c $?y)*
- *fact: (list a a b a c b b a c d c a b c)*

Match#	*$?x*	*$?y*
1	(a a b a c b b a c d c a b)	()
2	(a a b a c b b a c d)	(a b c)
3	(a a b a c b b a)	(d c a b c)
4	(a a b a)	(b b a c d c a b c)

Assigned pattern

- ***<assigned-pattern>:*** Assignment of matching fact index to a variable, so that it could be delete it as an action on the RHS.

```
<assigned-pattern> ::= <single-vble> <- <ordered-pattern>
```

- Example:

```
(defrule left
  ?f <- (puzzle $?x ?y 0 $?z)
  (test (<> (length$ $?x) 2))
  (test (<> (length$ $?x) 5))
  =>
  (retract ?f)
  (assert (puzzle $?x 0 ?y $?z)))
```

Test

- *<test>* is satisfied if *<function-call>* does not return ***False***

```
<test> ::= (test <function-call>)
<function-call> ::=  (<function-name> <expression>*)
<function-name> ::=  > | < | = | <> | eq | neq | <member>
<expression> ::= <constant> | <variable> | <function-call>
```

- Example:

```
(test (<> (length$ $?x) 2))
```

  where

  - *<>* is the not equal operator for *<integer>* or *<float>*
  - ***length$*** is a pre-defined function to compute the length of a list

- Mind that the operator is followed by operands

Member

- ***<member>*** is a pre-defined function in CLIPS:

```
(member$ <expression> <multifield-expression>)
```

- If ***<expression>*** is a single value and is ***member*** of second argument, then return the position in the field
- If the first argument is a multifield value and it is a substring of the 2nd argument, return range of positions in the second argument.
- If neither of these situations is satisfied, then FALSE is returned.
- Examples:

```
(member$ blue (create$ red 3 "text" 8.7 blue))
5
(member$ (create$ b c) (create$ a b c d))
(2 3)
(member$ 4 (create$ red 3 "text" 8.7 blue))
FALSE
```

RHS syntax

- Actions on the RHS allow to insert or delete facts, etc.:

```
<action> ::=
(assert <fact>+) |
(retract <fact-index>+)  |
(printout t <string> crlf) |
(halt)

<fact-index> ::= <integer> | <single-vble>
```

- Examples:

```
(defrule down
  (puzzle $?x 0 ?a ?b ?c $?z) =>
  (assert (puzzle $?x ?c ?a ?b 0 $?z)))

(defrule goal
  (puzzle 1 2 3 8 0 4 7 6 5) =>
  (printout t "Solution found!" crlf)
  (halt))
```

5 Inference Engine

- ***Input:*** Initial facts (F) and rules (R)
- ***Output:*** Final facts
- ***Procedure:***

$A = \emptyset$ // Agenda with rule instances

repeat

//Adding new rule instances to A using new facts:
$A =$ Instance(F, R, A)

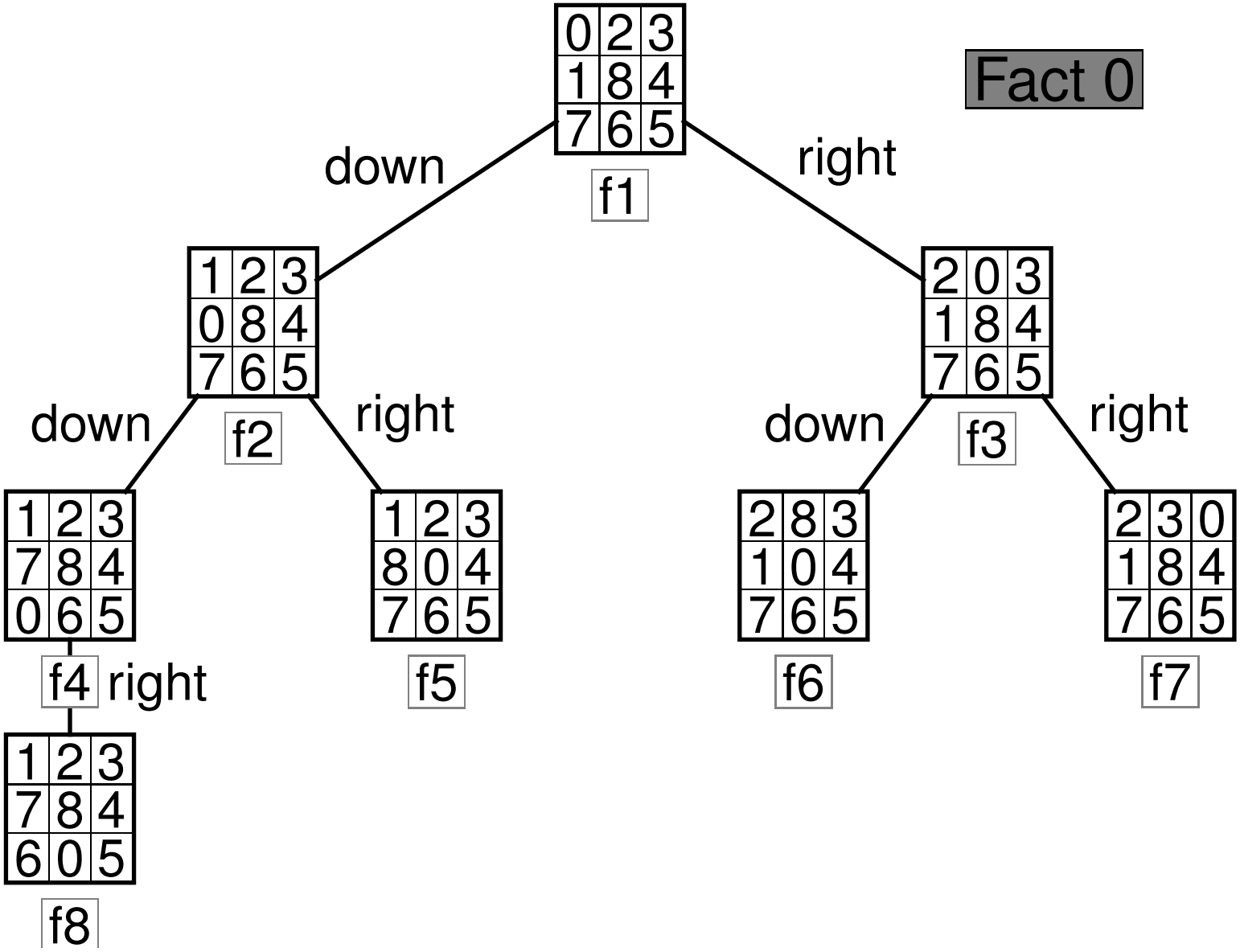
if $A == \emptyset$: **break** // Goal not reached

// Select a rule instance according to a search strategy
$InstRule =$ Select(A)

// execute $InstRule$ and update F and A:
$(F, A) =$ Execute$(F, A, InstRule)$

until Goal reached

Trace of BFS with CLIPS in 8-puzzle



Inference engine

- Facts in CLIPS are not duplicated in F by default
- ***Refraction:***

  A rule can only be instantiated once with the same fact and with the same value assignment to variables
- No duplication and refraction prevent from infinite rule activation
- A new fact may facilitate the activation of any rule

Search strategies

- Better known as conflict resolution strategies in CLIPS
- How rule instances are selected from the agenda
- Strategies: BFS, DFS, random, prioritized
- Example of explicit rule-based priority:

```
(defrule <name>
  (declare (salience <integer>))
  <LHS> => <RHS>)
```

- The higher the *<integer>* value, the higher the priority
- The default priority is zero

Sorting numbers in a list

```
(deffacts data (list 4 5 3 46 12 10))
(defrule sort
  ?f <- (list $?x ?y ?z $?w)
  (test (< ?z ?y))
  =>
  (retract ?f)
  (assert (list $?x ?z ?y $?w)))
(set-strategy breadth)
(watch facts)
(watch activations)
(reset)
(run)
(exit)
```

Run a trace showing the facts and the agenda.

Exercise (exam 2/11/2015, Question 1)

```
(deffacts F (list a b a b a))
(defrule R1
  ?f <- (list ?x $?y ?x $?z) =>
  (retract ?f)
  (assert (list $?y ?x $?z))
  (printout t "The list was modified" crlf))
(set-strategy breadth)
(watch facts)
(watch activations)
(reset)
(run)
(exit)
```

How many times the message is shown?

```
         CLIPS (V6.24 06/15/06)
CLIPS> (reset)
==> f-0     (initial-fact)
==> f-1     (list a b a b a)
==> Activation 0      R1: f-1
==> Activation 0      R1: f-1
CLIPS> (run)
<== f-1     (list a b a b a)
<== Activation 0      R1: f-1
==> f-2     (list b a b a)
==> Activation 0      R1: f-2
The list was modified
<== f-2     (list b a b a)
==> f-3     (list a b a)
==> Activation 0      R1: f-3
The list was modified
<== f-3     (list a b a)
==> f-4     (list b a)
The list was modified
CLIPS> (exit)
```

Exercise (2014-15, Question 4)

```
(deffacts F (list 1 2 3 4))
(defrule R1
?f <- (list ?x $?z) =>
(retract ?f)
(assert (list ?z))
(assert (element ?x)))
(defrule R2
?f <- (element ?x)
(element ?y)
(test (< ?x ?y)) =>
(retract ?f)
(assert (list-new ?x ?y)))
```

a) What is the final state of `F`? Run a trace.

b) What if `(deffacts F2 (list 1 2 2 4))` instead of `F`?

c) And if `(retract ?f)` is removed from `R1` with `F`?

c) And if `(retract ?f)` is removed from `R2` with `F`?

```
CLIPS (V6.24 06/15/06)

CLIPS> (facts)
f-0     (initial-fact)
f-8     (lista-new 1 2)
f-9     (lista)
f-10    (element 4)
f-11    (lista-new 2 3)
f-12    (lista-new 3 4)
For a total of 6 facts.
```

Exercise: Factorial

```
(deffacts F (number 5) (fact 1))
(defrule factorial
  ?f1 <- (number ?n1)
  ?f2 <- (fact ?n2)
  (test (> ?n1 1))
  =>
  (retract ?f1 ?f2)
  (assert (number (- ?n1 1)))
  (assert (fact (* ?n2 ?n1 ))))
(set-strategy breadth)
(watch facts)
(watch activations)
(reset)
(run)
(exit)
```

What is the final content of facts?

```
         CLIPS (V6.24 06/15/06)
<== f-1     (number 5)
<== f-2     (fact 1)
==> f-3     (number 4)
==> f-4     (fact 5)
==> Activation 0      factorial: f-3,f-4
<== f-3     (number 4)
<== f-4     (fact 5)
==> f-5     (number 3)
==> f-6     (fact 20)
==> Activation 0      factorial: f-5,f-6
<== f-5     (number 3)
<== f-6     (fact 20)
==> f-7     (number 2)
==> f-8     (fact 60)
==> Activation 0      factorial: f-7,f-8
<== f-7     (number 2)
<== f-8     (fact 60)
==> f-9     (number 1)
==> f-10    (fact 120)
```

Exercise: Member

```
(deffacts F (list A B C A B C C B A C B A))
(defrule R1
?f1 <- (list $?x1 ?y $?x2 ?y $?x3)
(test (> (length $?x2) 0))
(test (not (member ?y $?x2)))
=>
(retract ?f1)
(assert (list $?x1 ?y ?y $?x3)))
(set-strategy breadth)
(watch facts)
(watch activations)
(reset)
(run)
(exit)
```

What is the final content of facts?

```
         CLIPS (V6.24 06/15/06)
<== f-1     (list A B C A B C C B A C B A)
<== Activation 0       R1: f-1
<== Activation 0       R1: f-1
<== Activation 0       R1: f-1
<== Activation 0       R1: f-1
<== Activation 0       R1: f-1
<== Activation 0       R1: f-1
<== Activation 0       R1: f-1
==> f-2     (list A B C A B C C B A A)
==> Activation 0       R1: f-2
==> Activation 0       R1: f-2
==> Activation 0       R1: f-2
==> Activation 0       R1: f-2
==> Activation 0       R1: f-2
<== f-2     (list A B C A B C C B A A)
<== Activation 0       R1: f-2
<== Activation 0       R1: f-2
<== Activation 0       R1: f-2
<== Activation 0       R1: f-2
==> f-3     (list A B C A B B A A)
==> Activation 0       R1: f-3
==> Activation 0       R1: f-3
==> Activation 0       R1: f-3
<== f-3     (list A B C A B B A A)
<== Activation 0       R1: f-3
<== Activation 0       R1: f-3
==> f-4     (list A B C A A A)
==> Activation 0       R1: f-4
<== f-4     (list A B C A A A)
==> f-5     (list A A A A)
```