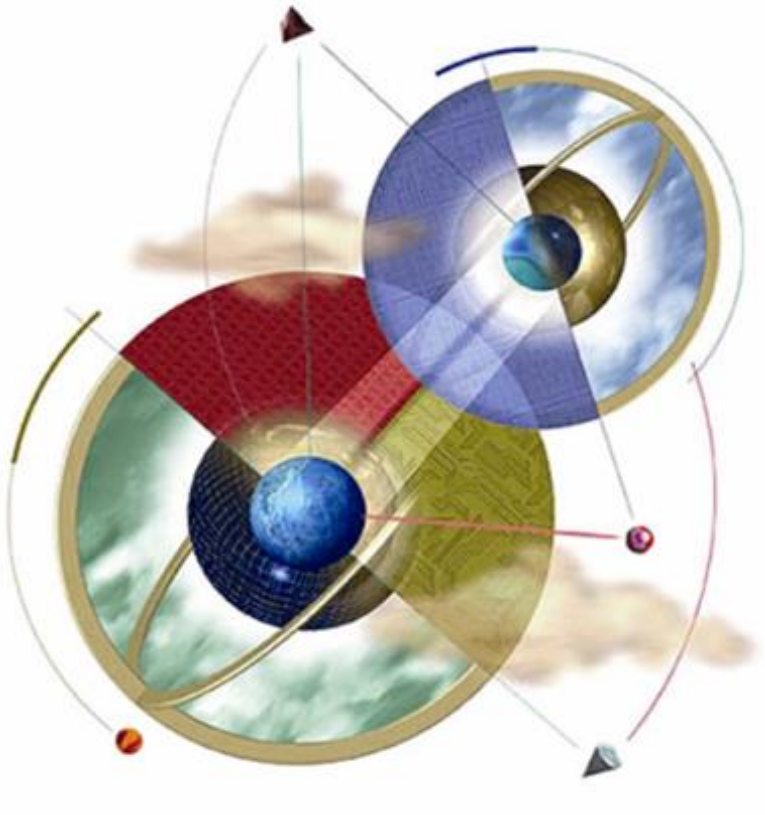
Expert Systems

- 上課用書
  - Expert Systems: Principles and Programming (4th Edition), Giarratano and Riley, ISBN 0534384471 (開發圖書代理)
- 上課內容
  1. Introduction to Expert Systems
  2. The Representation of Knowledge
  3. Methods of Inference
  7. Introduction to CLIPS
  8. Advanced Pattern Matching
  9. Modular Design, Execution Control, and Rule Efficiency
  A. FuzzyCLIPS
- 評分方式
  - 期末專題程式 20% (沒有期中考試和期末考試)
  - 小考作業程式 80%
    - 當場完成並測試成功為100分，隔週補交分數為70分，隔週未交為零分
    - 病假或事假一律隔週補交分數為80分，公假隔週補交不扣分仍為100分



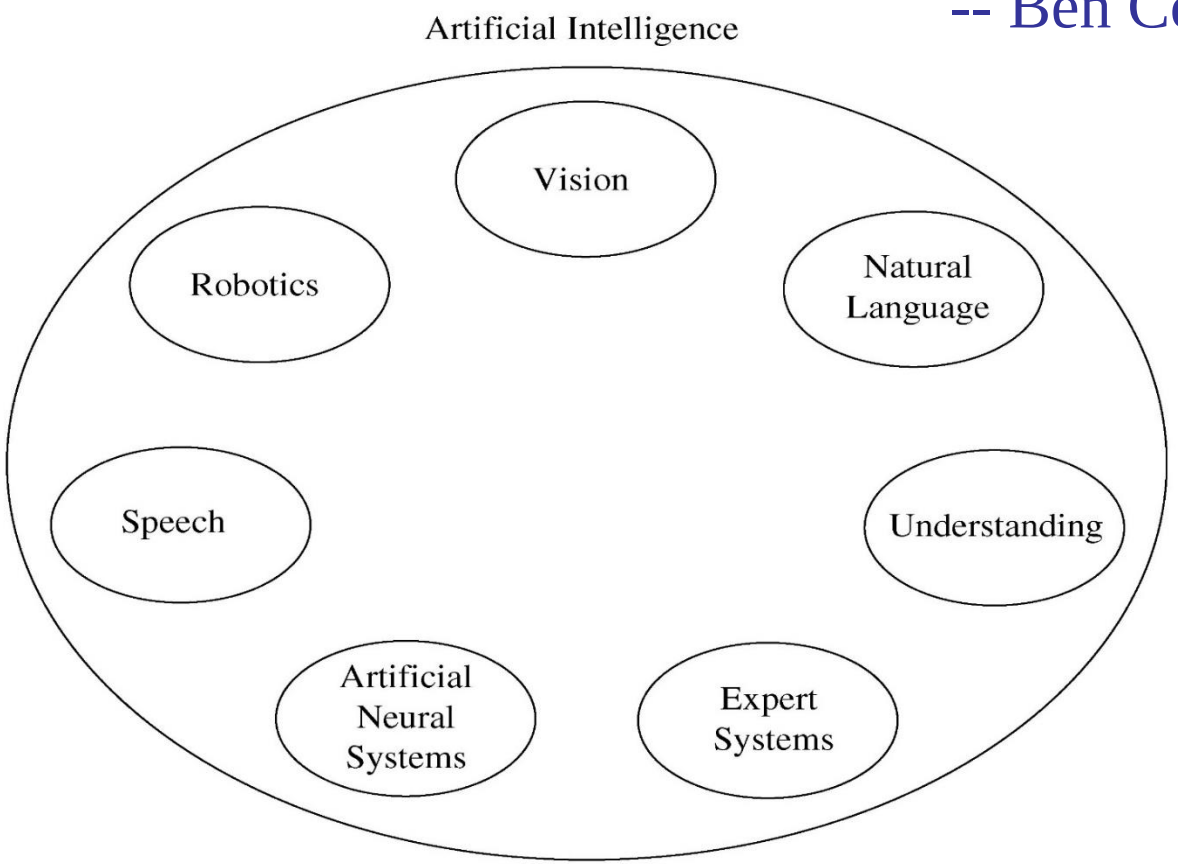
Chapter 1: Introduction to Expert Systems

Expert Systems: Principles and Programming, Fourth Edition

Artificial Intelligence (AI)

- Using methods based on the intelligent behavior of humans and other animals to solve complex problems.

-- Ben Coppin



What is an expert system?

"An expert system is a computer system that emulates with the decision-making capabilities of a human expert."

"An intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solutions".

Professor Edward Feigenbaum
Stanford University

The Role of AI

- An algorithm is an ideal solution guaranteed to yield a solution in a finite amount of time.
- When an algorithm is not available or is insufficient, we rely on artificial intelligence (AI).
- Expert system relies on inference – we accept a "reasonable solution" or "feasible solution"

Famous Expert Systems

- DENDRAL – used in chemical mass spectroscopy to identify chemical constituents
- MYCIN – medical diagnosis of illness
- DIPMETER – geological data analysis for oil
- PROSPECTOR – geological data analysis for minerals
- XCON/R1 – configuring computer systems

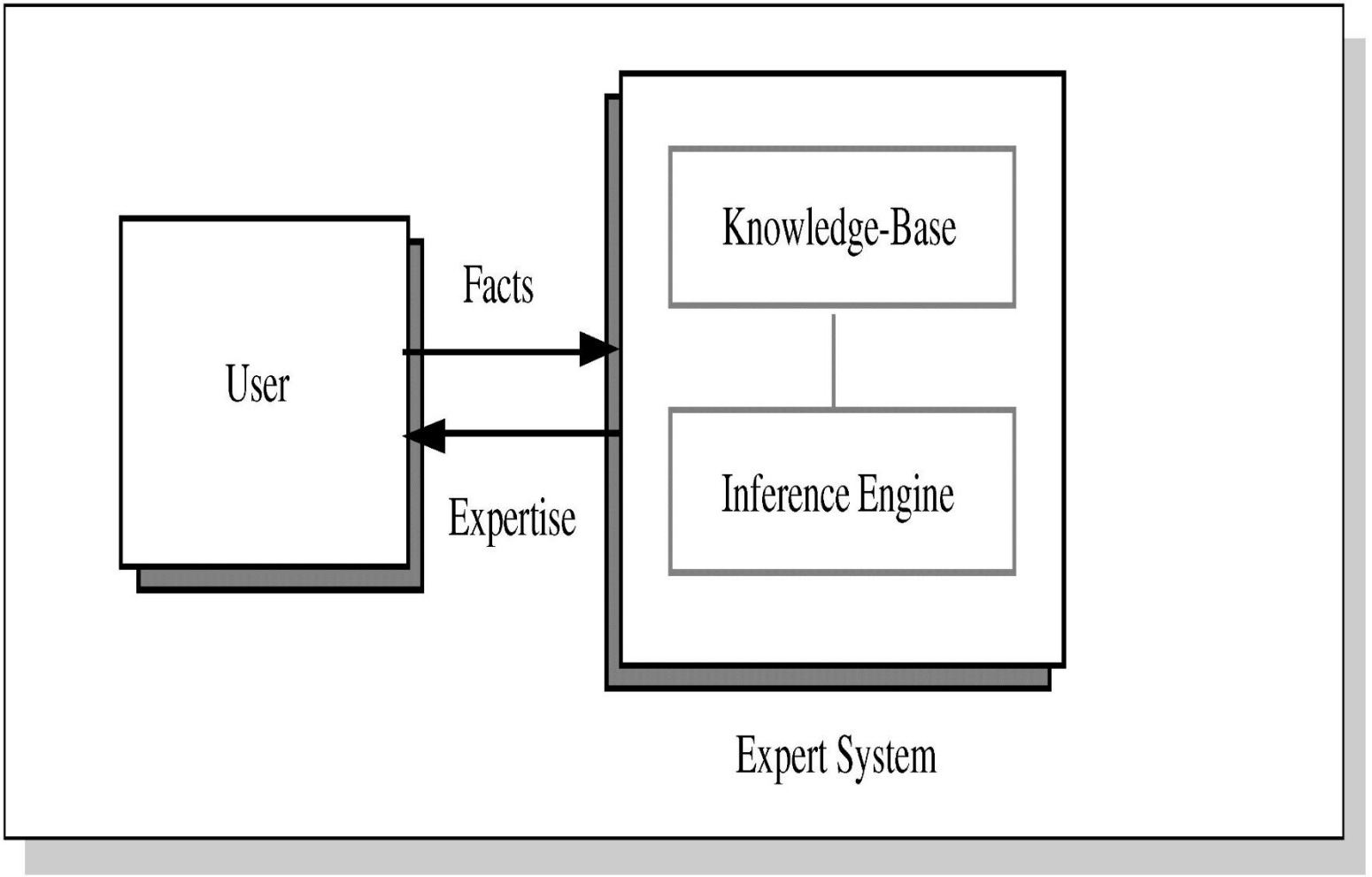
Expert system participants

- Users
- Experts
  - An expert's knowledge is specific to one problem domain, as opposed to knowledge about general problem-solving techniques
- Knowledge Engineers
  - Eliciting knowledge from experts and building expert systems

Expert System Main Components

- Knowledge base – obtainable from books, magazines, knowledgeable persons, etc.
- Inference engine – draws conclusions from the knowledge base

Basic Functions of Expert Systems

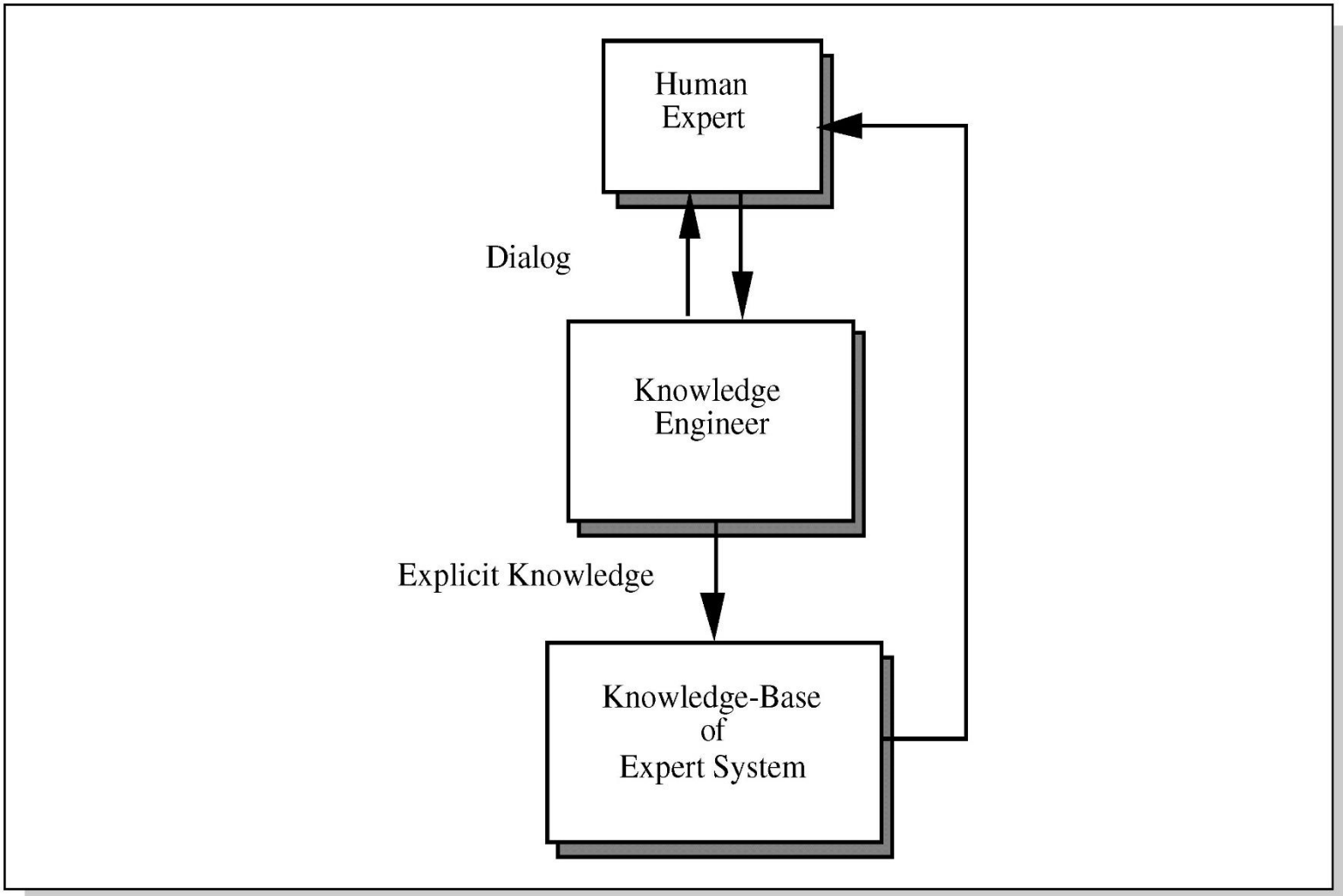


Knowledge Engineering

The process of building an expert system:

1. The knowledge engineer establishes a dialog with the human expert to elicit knowledge.
2. The knowledge engineer codes the knowledge explicitly in the knowledge base.
3. The expert evaluates the expert system and gives a critique to the knowledge engineer.

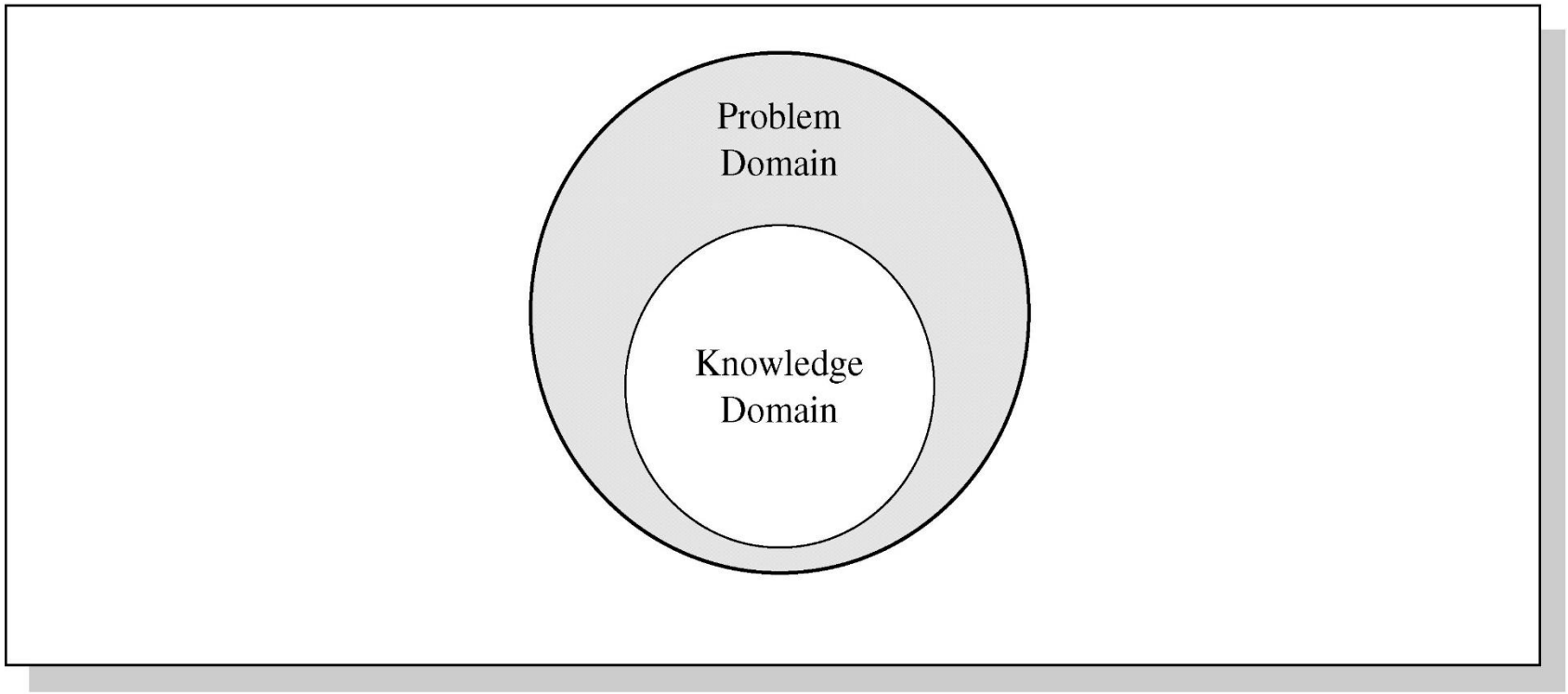
Development of an Expert System



Problem Domain vs. Knowledge Domain

- An expert's knowledge is specific to one problem domain – medicine, finance, science, engineering, etc.
- The expert's knowledge about solving specific problems is called the knowledge domain.
- The problem domain is always a superset of the knowledge domain.

Problem and Knowledge Domain Relationship

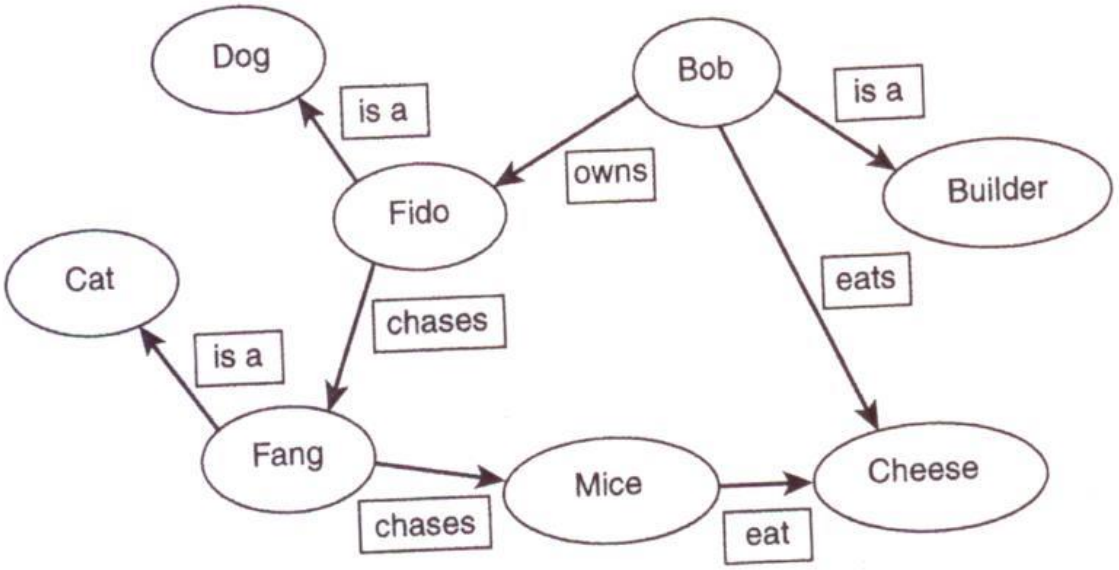


Developing expert systems

- Knowledge representations
  - representing the real world knowledge within computers
  - logic vs. graphical notation vs. frame-based notation
- Inference on knowledge
  - making inferences to solve problems
  - forward chaining vs. backward chaining

Knowledge Representation

- Predicate Logic
  - Builder(Bob) ∧ Dog(Fido) ∧ Cat(Fang) ∧ owns(Bob,Fido) ∧ eats(Bob,Cheese) ∧ chases(Fido,Fang) ∧ chases(Fang,Mise) ∧ eat(Mice,Cheese)
- Semantic Nets

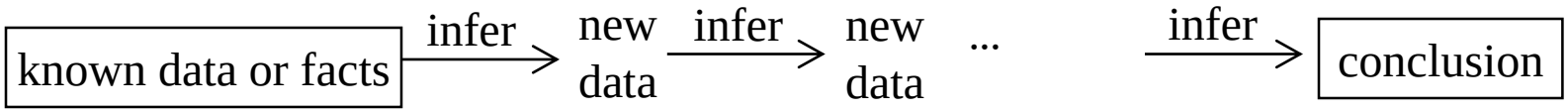


- Frame

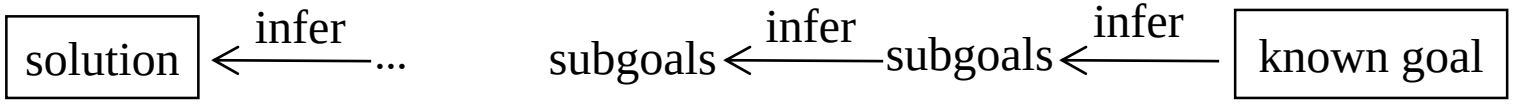
Frame Name	Slot	Slot Value
Bob	is a	Builder
	owns	Fido
	eats	Cheese
Fido	is a	Dog
	chases	Fang
Fang	is a	Cat
	chases	Mice
Mice	eat	Cheese

Inference

- Forward chaining
  - data-driven



- Backward chaining
  - goal-driven



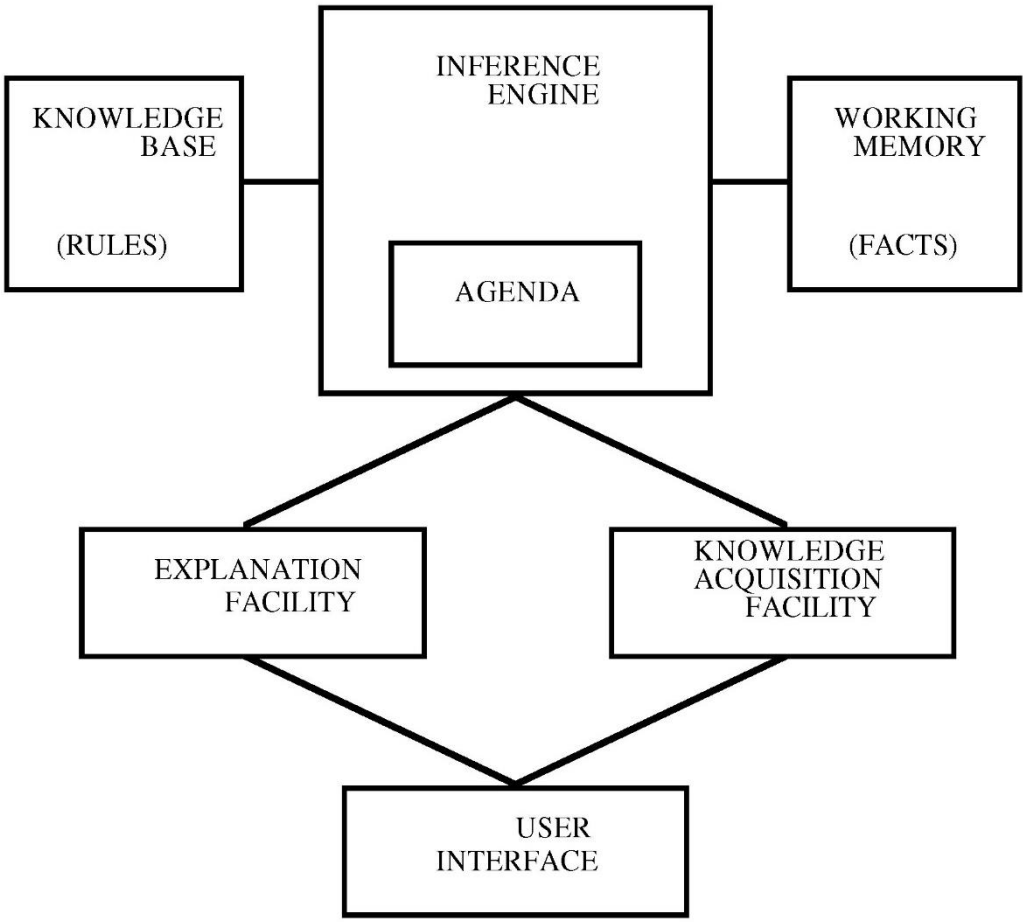
AI Programming Languages

- Prolog (PROgramming in LOGic)
  - Build a database of facts and rules (in logic)
  - Answer questions by a process of logical deduction
  - Backward chaining (backtracking)
- LISP (LISt Programming)
  - Use lists to represent programs and data
  - Provide list manipulation functions to handle lists
  - Forward chaining
- CLIPS (C Language Integrated Production System)
  - Build a Knowledge base (rules)
  - Match known facts and the rules (inference engine)
  - Forward chaining

Elements of an Expert System

- Knowledge base – rules
- Working memory –facts used by rules
- Inference engine – makes inferences deciding which rules are satisfied and prioritizing.
- Agenda – a prioritized list of rules in the inference engine
- Explanation facility – explains reasoning of expert system
- Knowledge acquisition facility – the user to enter knowledge in the system bypassing the knowledge engineer
- User interface – mechanism by which user and system communicate.

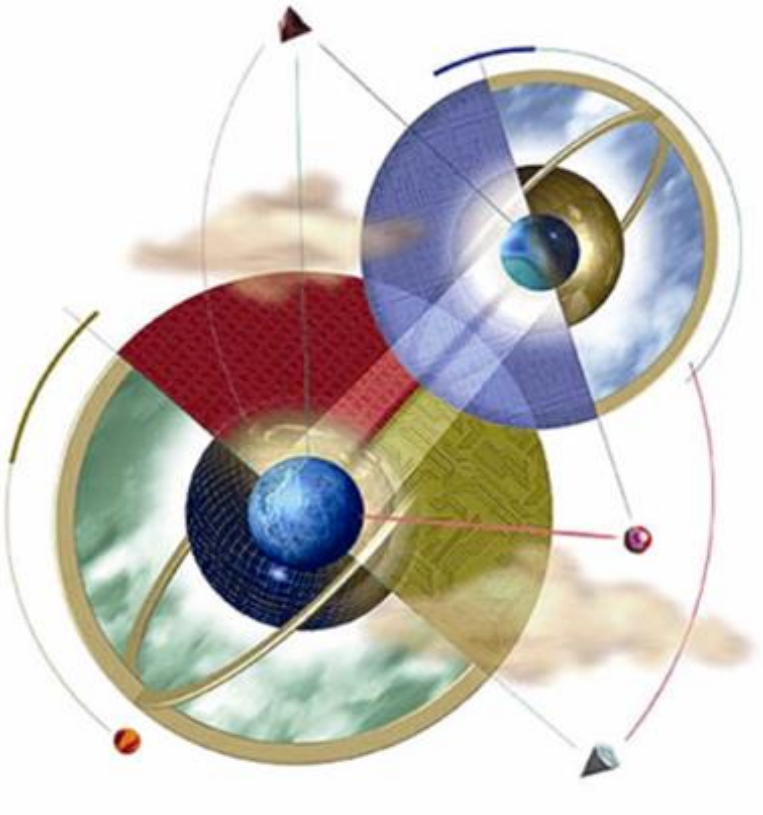
Structure of a Rule-Based Expert System



Rules of Inference

- Knowledge base contains production rules
  - Production rules can be expressed in IF-THEN format
- Inference engine determines which rule antecedents (IF parts) are satisfied by the facts
- Modus Ponens
  - Inference on rule-based expert systems

$p \rightarrow q$	If there is power, the computer will work
p	There is power
$\therefore q$	$\therefore$ The computer will work



Chapter 2: The Representation of Knowledge

Expert Systems: Principles and Programming, Fourth Edition

Knowledge in Expert Systems

- Knowledge refers to rules that are activated by facts or other rules.
- Activated rules produce new facts or conclusions.
- Conclusions are the end-product of inferences when done according to formal rules.

Metaknowledge

- Metaknowledge is knowledge about knowledge and expertise.
- In an expert system, an ontology is the metaknowledge that describes everything known about the problem domain.
- Most successful expert systems are restricted to as small a domain as possible.

Knowledge Representation Techniques

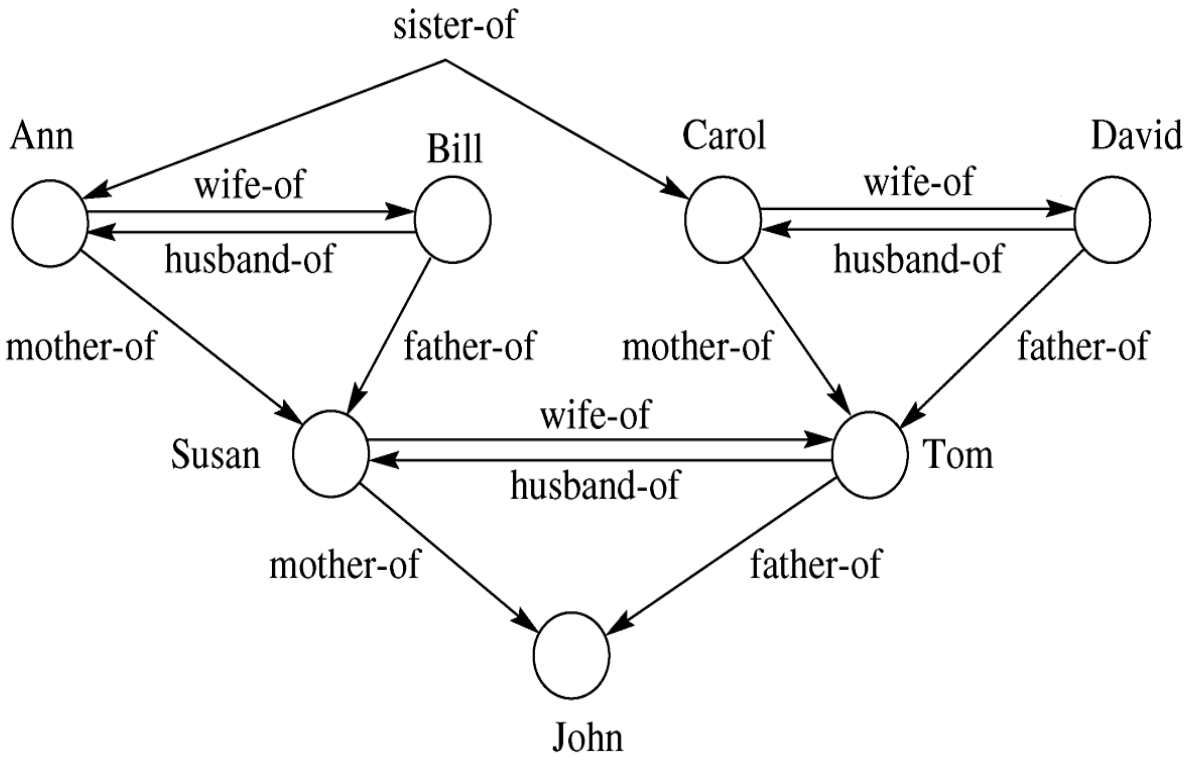
A number of knowledge-representation techniques have been devised:

- Rules
- Semantic nets
- Frames
- Scripts
- Logic
- Conceptual graphs

Semantic Nets

- A classic representation technique for propositional information
- Propositions – a form of declarative knowledge, stating facts (true/false)
- Propositions are called “atoms” – cannot be further subdivided.
- Semantic nets consist of nodes (objects, concepts, situations) and arcs (relationships between them).

A Semantic Nets



(b) A Semantic Net

PROLOG and Semantic Nets

- In PROLOG, predicate expressions consist of the predicate name, followed by zero or more arguments enclosed in parentheses, separated by commas.
- Example:

  mother(Ann,Susan)

  means that Ann is the mother of Susan

PROLOG

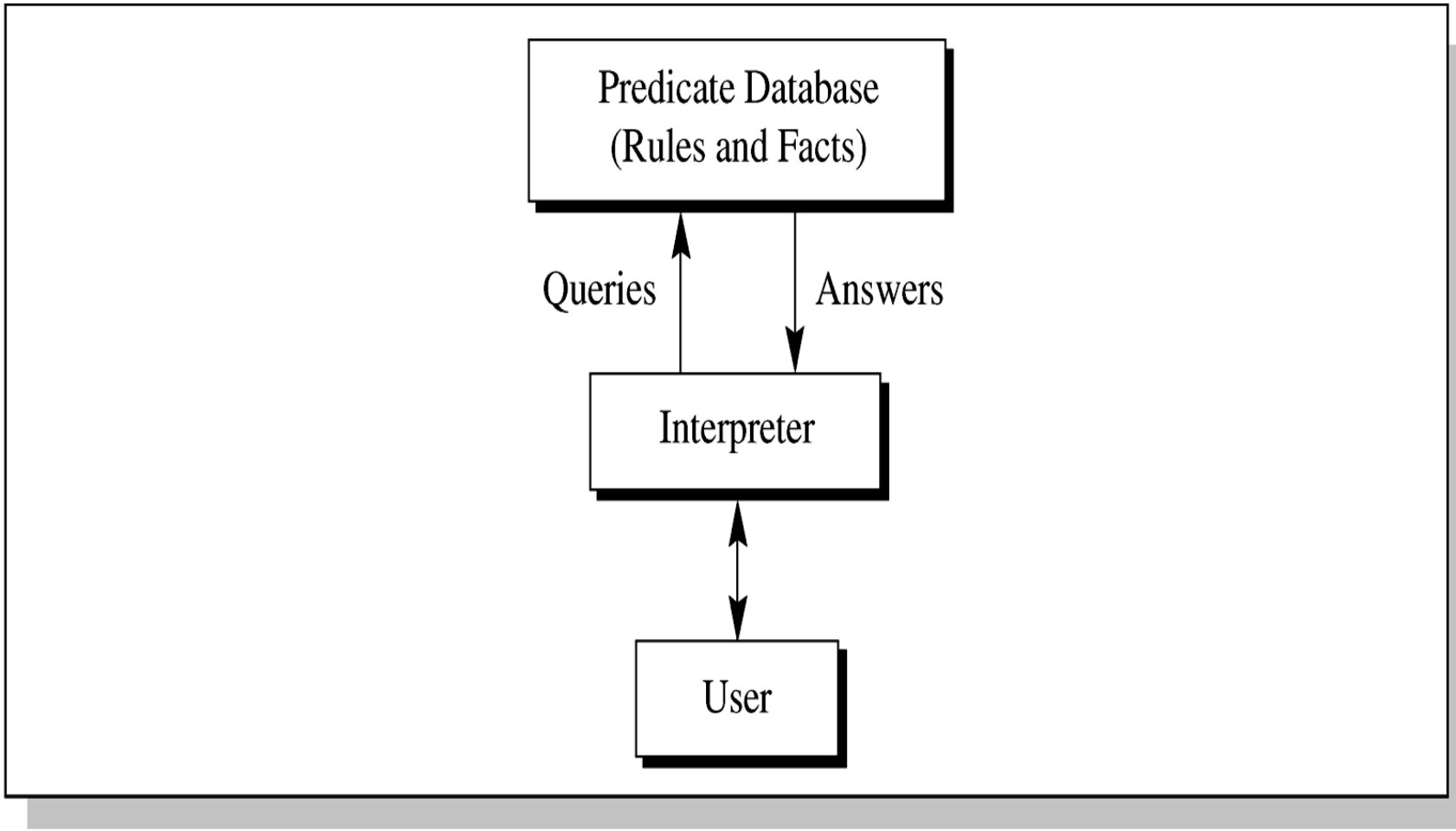
- Programs consist of facts and rules in the general form of goals.
- General form: `p:- p1, p2, …, pN`

  `p` is called the rule's head and the p_i represents the subgoals
- Example:

  grandparent(x,y) :- parent(x,z) , parent(z,y)

  grandmother(x,y) :- grandparent(x,y) , female(x)

General Organization of a PROLOG System



Problems with Semantic Nets

- inability to define knowledge in the same way that logic can, the link and node names must be rigorously defined.
  - A solution to this is extensible markup language (XML) and ontologies.
- combinatorial explosion of searching nodes
- heuristic inadequacy
  - there is no way to embed heuristic information in the net on how to efficiently search the net

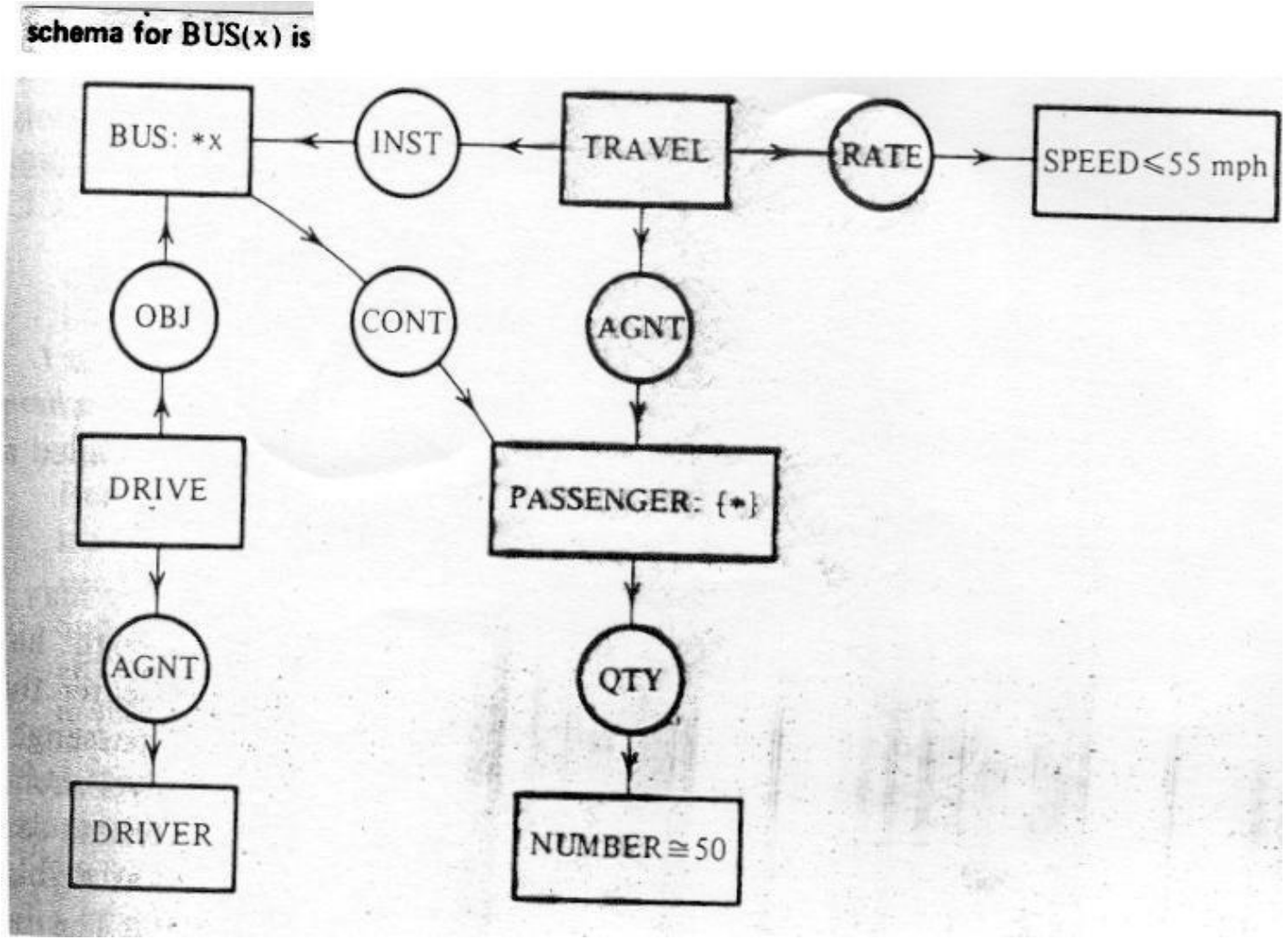
Schemata

- Knowledge Structure is used to represent an ordered collection of knowledge rather than just data
- A knowledge structure can be either shallow or deep
  - Semantic Nets are shallow knowledge structures, because all knowledge is contained in nodes and links (based on syntax)
    - For example, IF a person has a fever THEN take an aspirin
  - A deep knowledge structure has causal knowledge that explains why something occurs (based on semantics)
    - Doctors have causal knowledge. If a treatment is not working right, doctors can use reasoning powers to find an alternative
- Many types of real-world knowledge cannot be represented by the simple structure of a semantic net, that is, more complex structures are needed.

Schemata (Cont.)

- Schema is a more complex knowledge structure than a semantic net.
- A concept schema is used when we represent concepts
  - an abstraction (general properties) of an object
  - the stereotype (a typical example) in our minds of concepts
- Schemas have an internal structure to their nodes while semantic nets do not.
  - In a semantic net, nodes are represented by data alone, but in a schema, a node is like a record, which may contain, in addition to simply data, but also data, records, or pointers to other nodes.

A Schema in Conceptual Graphs



Frames

- The frame is one type of schema
- Another type of schema is the script – time-ordered sequence of frames
- Frames provide a convenient structure for representing objects that are typical to a given situation
  - Frames are useful for simulating commonsense knowledge (knowledge that is generally known)
- Semantic nets provide 2-dimensional representation of knowledge (links and nodes); frames provide 3-dimensional by allowing nodes to have structures
- Frames represent related knowledge about narrow subjects having much default knowledge.

Frames (Cont.)

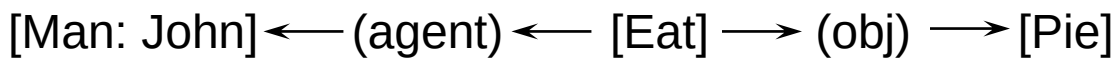
- A frame is a group of slots and fillers (attributes and values) that defines a stereotypical object that is used to represent generic/specific knowledge.
- Problems with frames
  - allowing unrestrained alteration/cancellation of slots
  - it is impossible to make universal statements (definitions)

Figure 2.8 A Car Frame

Slots	Fillers
manufacturer	General Motors
model	Chevrolet Caprice
year	1979
transmission	automatic
engine	gasoline
tires	4
color	blue

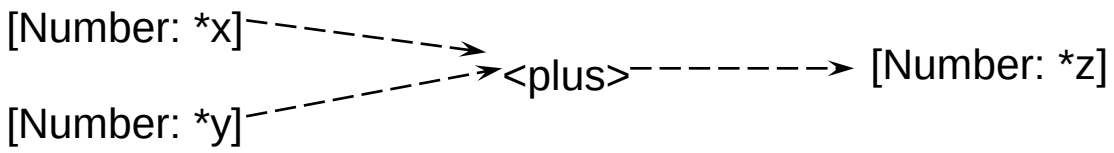
Conceptual Graphs

- Conceptual graphs consists of several components:
  - [Concept: referent] and (relation) :



$\exists x \exists y(Man(John) \wedge agent(x,John) \wedge Eat(x) \wedge obj(x,y) \wedge Pie(y))$

  - <actor> :



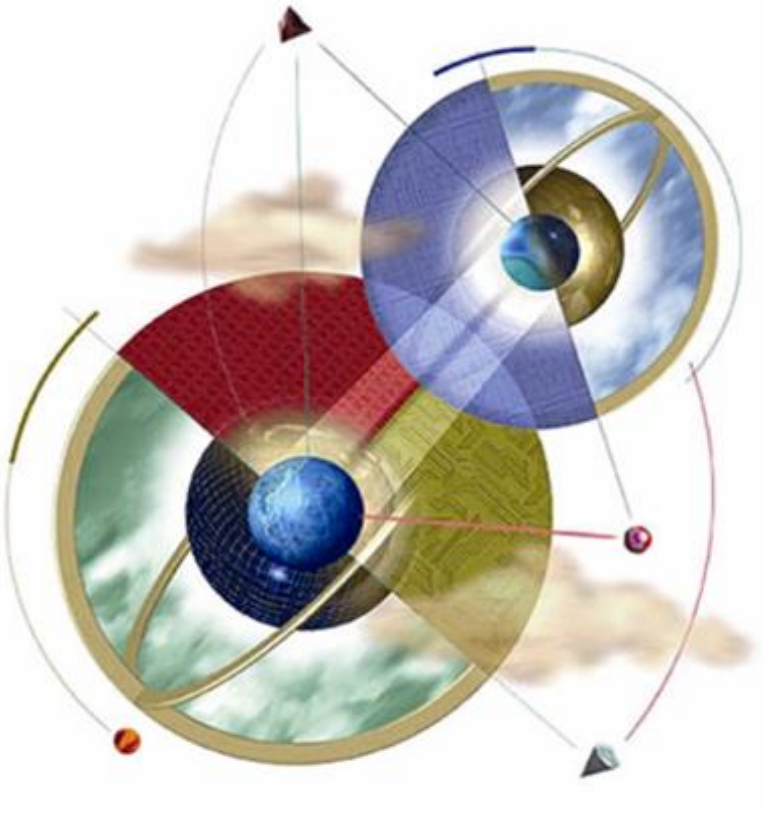
  - <<demon>> :

Logic and Sets

- Knowledge can also be represented by symbols of logic.
- Logic is the study of rules of exact reasoning – inferring conclusions from premises.
- Automated reasoning – logic programming in the context of expert systems.

Forms of Logic

- Earliest form of logic was based on the syllogism – developed by Aristotle.
- Syllogisms – have two premises that provide evidence to support a conclusion.
- Example:
  - Premise: *All cats are climbers.*
  - Premise: *Garfield is a cat.*
  - Conclusion: *Garfield is a climber.*



Chapter 3: Methods of Inference

Expert Systems: Principles and Programming, Fourth Edition

Types of Inference

- Deduction – reasoning where conclusions must follow from premises
- Induction – inference is from the specific case to the general
- Intuition – no proven theory
- Heuristics – rules of thumb based on experience
- Generate and test – trial and error
- Abduction – reasoning back from a true condition to the premises that may have caused the condition
- Default – absence of specific knowledge
- Analogy – inferring conclusions based on similarities with other situations

Deductive Logic

- Deductive logic can determine the validity of an argument.
- A logical argument is a group of statements in which the last is justified on the basis of the previous ones in the chain of reasoning
- Syllogism is one type of logical argument, which has two premises (or antecedents) and one conclusion (or consequent)
- Deductive argument – conclusions reached by following true premises must themselves be true

Syllogisms vs. Rules

- Syllogism:

  Premise: Anyone who can program is intelligent

  Premise: John can program

  Conclusion: John is intelligent

- IF-THEN rule:

  IF Anyone who can program is intelligent and
  John can program

  THEN John is intelligent.

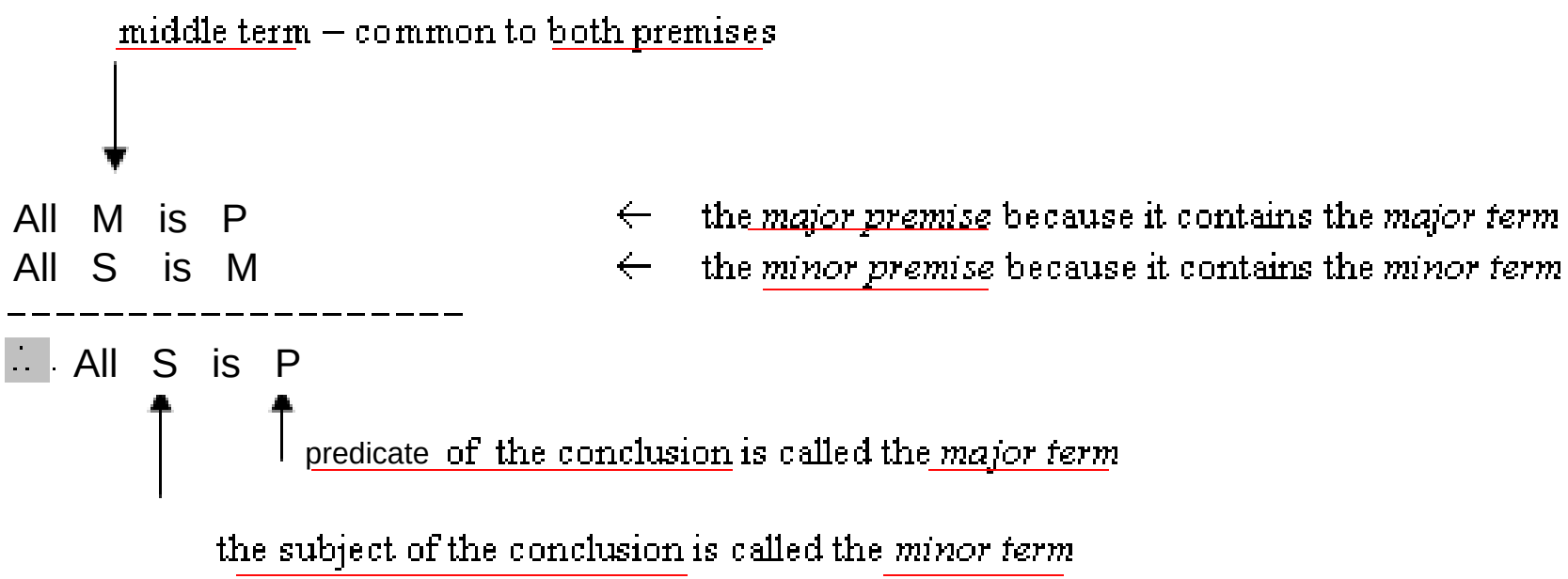
Categorical Syllogism

Premises and conclusions are defined using categorical statements of the four forms:

Table 3.2 Categorical Statements

Form	Schema	Meaning
A	All S is P	universal affirmative
E	No S is P	universal negative
I	Some S is P	particular affirmative
O	Some S is not P	particular negative

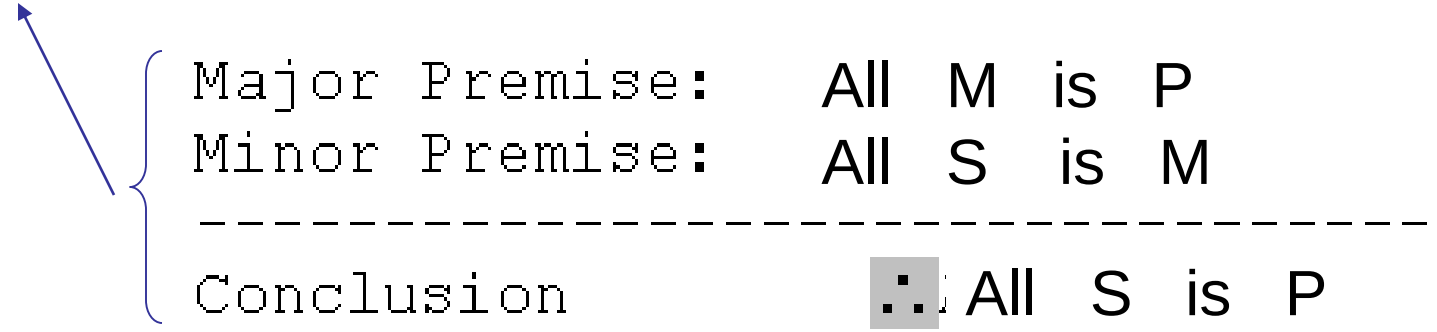
Terms


middle term – common to both premises
All M is P
All S is M
the major premise because it contains the major term
the minor premise because it contains the minor term
∴ All S is P
predicate of the conclusion is called the major term
the subject of the conclusion is called the minor term

Syllogisms in Standard Form

To write the syllogism in standard form, we would write:

AAA mood



mood: defined by three letters that gives the form of the major premise, minor premise, and conclusion, respectively.

Form	Schema	Meaning
A	All S is P	universal affirmative
E	No S is P	universal negative
I	Some S is P	particular affirmative
O	Some S is not P	particular negative

Syllogism Type

- There are four possible patterns of arranging the S, P, and M terms:

	Figure 1	Figure 2	Figure 3	Figure 4
Major Premise	M P	P M	M P	P M
Minor Premise	S M	S M	M S	M S
Conclusion	S P	S P	S P	S P

All M is P
All S is M
∴ All S is P

→ AAA-1 syllogism type

All P is M
No M is S
∴ No S is P

→ AEE-4 syllogism type

Validity in Syllogism

- AEE-1 syllogism is invalid:

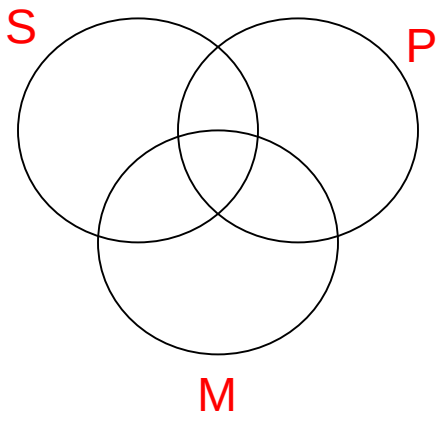
All microcomputers are computers

No mainframe is a microcomputer

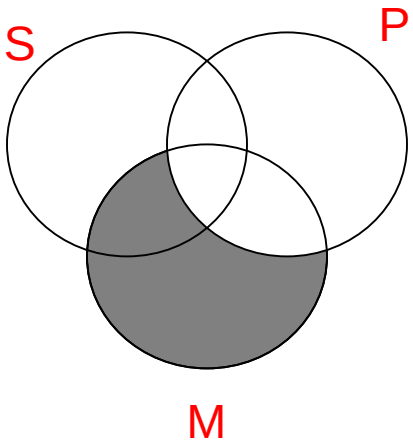
∴ No mainframe is a computer

All M is P
No S is M

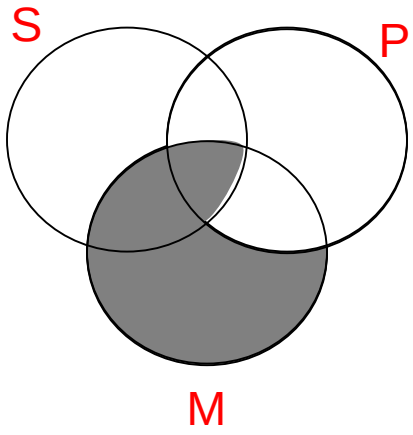
∴ No S is P



Venn Diagram



After Major Premise



After Minor Premise

* The shaded section of M indicates there are no elements in that portion.

Proving the Validity of Syllogistic Arguments Using Venn Diagrams

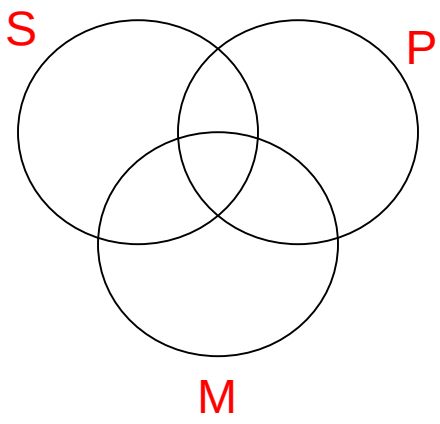
1. If a class is empty, it is shaded.
2. Universal statements, A and E are always drawn before particular ones.
3. If a class has at least one member (i.e. some), mark it with an *.
4. If an area has been shaded, not * can be put in it.

Example#1

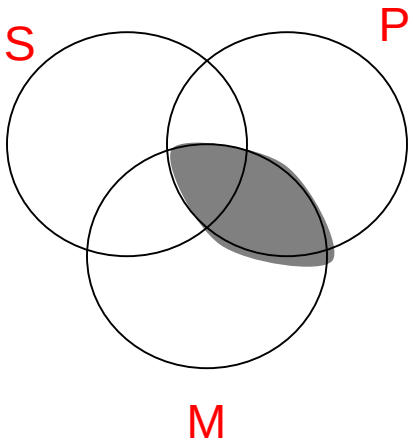
- EAE-1 syllogism is valid:

No M is P
All S is M

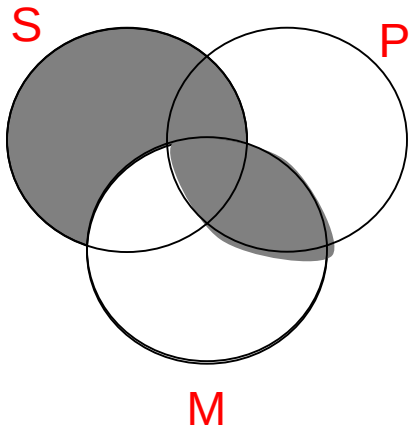
∴ No S is P



Venn Diagram



After Major Premise



After Minor Premise

Example#2

- IAI-4 syllogism is valid:

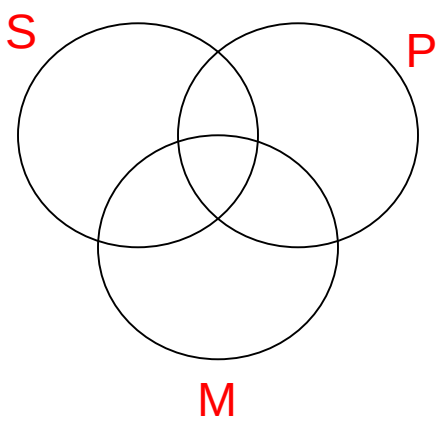
  Some computers are laptops

  All laptops are transportable

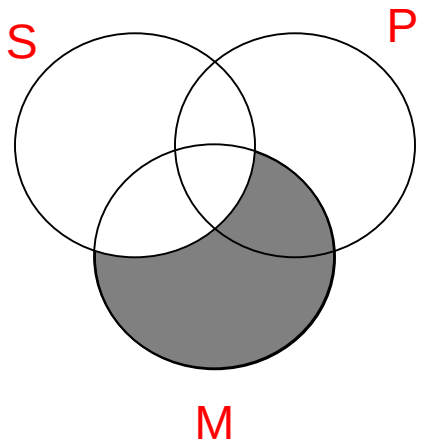
  ∴ Some transportables are computers

Some P is M
All M is S

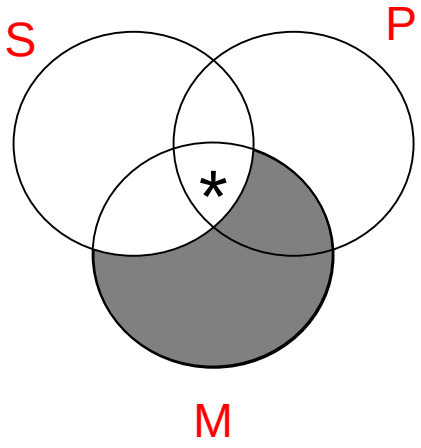
∴ Some S is P



Venn Diagram



All M is S



Some P is M

Rules of Inference

- Syllogisms address only a small portion of the possible logical statements.
- Venn diagrams are insufficient for complex arguments.
- Propositional logic offers another means of describing arguments:

$A \rightarrow B$	If there is power, the computer will work
A	There is power
$\therefore B$	$\therefore$ The computer will work

Modus Ponens

- A general schema which forms the basis of rule-based expert systems:

  $p \rightarrow q$

  p

  $\therefore q$

- Another notation:
  - $p, p \rightarrow q ; \ \therefore q$
  - $(p \rightarrow q) \wedge p \rightarrow q$
  - $P1, P2, \ldots, Pn ; \ \therefore C$
  - $P1 \wedge P2 \wedge \ldots \wedge Pn \rightarrow C$

Truth Table for Modus Ponens

p	q	$p \rightarrow q$	$(p \rightarrow q) \wedge p$	$(p \rightarrow q) \wedge p \rightarrow q$
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

Some Rules of Inference

1. Law of Detachment

$$\begin{array}{l} p \rightarrow q \\ \underline{p\quad\quad} \\ \therefore\ q \end{array}$$

2. Law of the Contrapositive

$$\begin{array}{l} \underline{p \rightarrow q} \\ \therefore\ \sim q \rightarrow \sim p \end{array}$$

3. Law of Modus Tollens

$$\begin{array}{l} p \rightarrow q \\ \underline{\sim q\quad\quad} \\ \therefore\ \sim p \end{array}$$

4. Chain Rule (Law of the Syllogism)

$$\begin{array}{l} p \rightarrow q \\ \underline{q \rightarrow r} \\ \therefore\ p \rightarrow r \end{array}$$

5. Law of Disjunctive Inference

$$\begin{array}{l} p \vee q \\ \underline{\sim p\quad\quad} \\ \therefore\ q \end{array} \qquad \begin{array}{l} p \vee q \\ \underline{\sim q\quad\quad} \\ \therefore\ p \end{array}$$

Rules of Inference

6. Law of the Double Negation	$\underline{\sim(\sim p)}$ $\therefore p$	
7. De Morgan's Law	$\underline{\sim(p \wedge q)}$ $\therefore \sim p \vee \sim q$	$\underline{\sim(p \vee q)}$ $\therefore \sim p \wedge \sim q$
8. Law of Simplification	$\underline{p \wedge q}$ $\therefore p$	$\underline{\sim(p \vee q)}$ $\therefore \sim p$
9. Law of Conjunction	p $\underline{q}$ $\therefore p \wedge q$	
10. Law of Disjunctive Addition	$\underline{p}$ $\therefore p \vee q$	
11. Law of Conjunctive Argument	$\sim(p \wedge q)$ $\underline{p}$ $\therefore \sim q$	$\sim(p \wedge q)$ $\underline{q}$ $\therefore \sim p$

Limitations of Propositional Logic

- The invalidity of an argument means that the argument cannot be proven under propositional logic
  - the conclusion is not necessarily incorrect.
  - may be poorly concocted.

p
q

∴ r

where p= All men are mortal
q = Socrates is a man
r = Socrates is mortal

- An argument may not be provable using propositional logic, but may be provable using predicate logic.
  - $(\forall x)\ (\ man(x) \rightarrow mortal(x)\ )$
    $man(Socrates)$
  - $man(Socrates) \rightarrow mortal(Socrates)$ <Universal Instantiation>
    $mortal(Socrates)$ <Modus Ponens>

First-Order Predicate Logic

- The Rule of Universal Instantiation states that an individual may be substituted for a universe.
- Syllogistic logic can be completely described by predicate logic.

Type	Schema	Predicate Representation
A	All S is P	$(\forall x)\ (S(x) \rightarrow P(x))$
E	No S is P	$(\forall x)\ (S(x) \rightarrow \sim P(x))$
I	Some S is P	$(\exists x)\ (S(x) \wedge P(x))$
O	Some S is not P	$(\exists x)\ (S(x) \wedge \sim P(x))$

Chaining

- Chain – a group of multiple inferences that connect a problem with its solution
- A chain that is searched / traversed from a problem to its solution is called a forward chain.
- A chain traversed from a hypothesis back to the facts that support the hypothesis is a backward chain.

Figure 3.21 Causal Forward Chaining

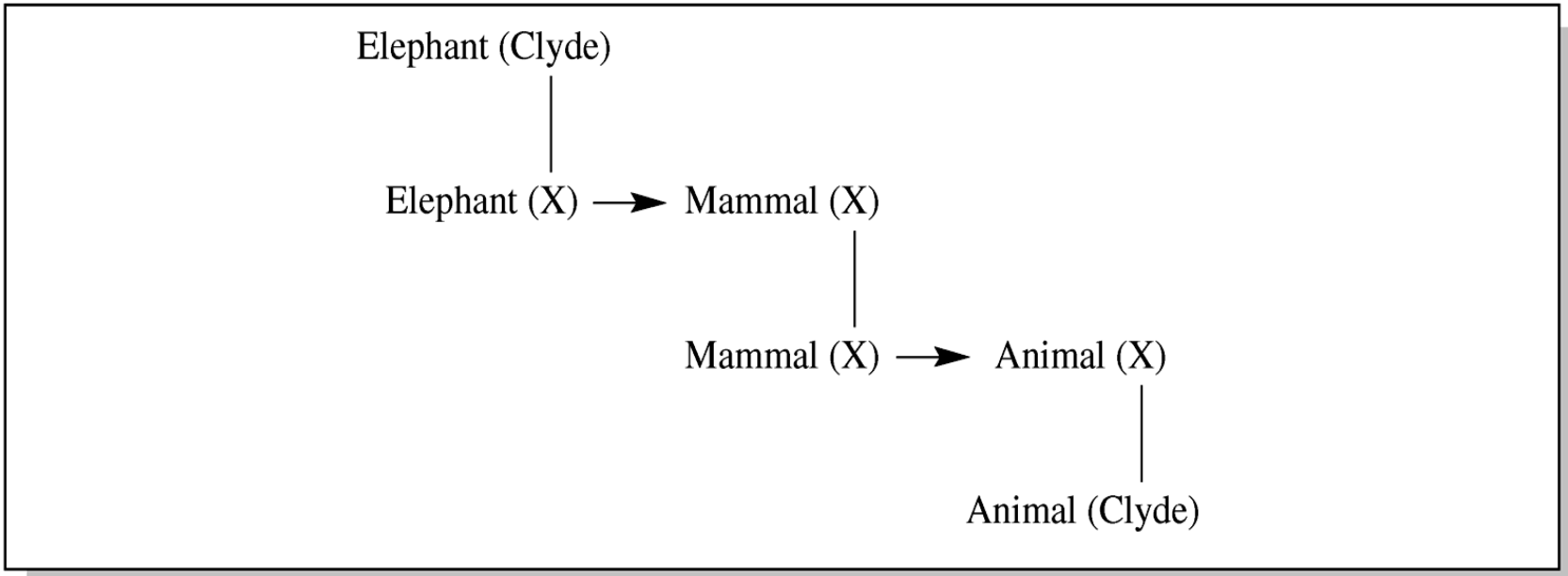


Fig. 3.23 Forward Chaining (Bottom-Up Reasoning)

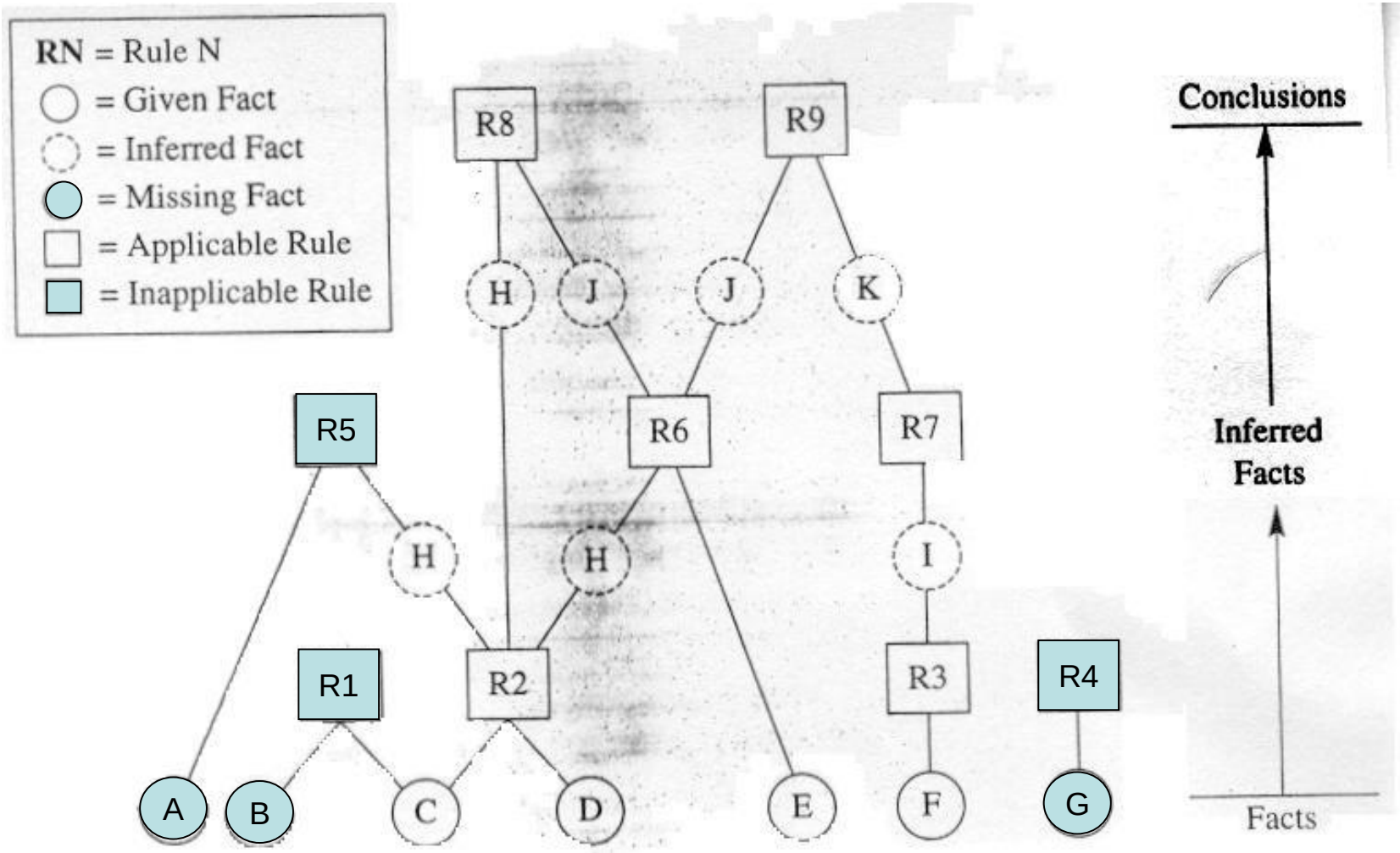
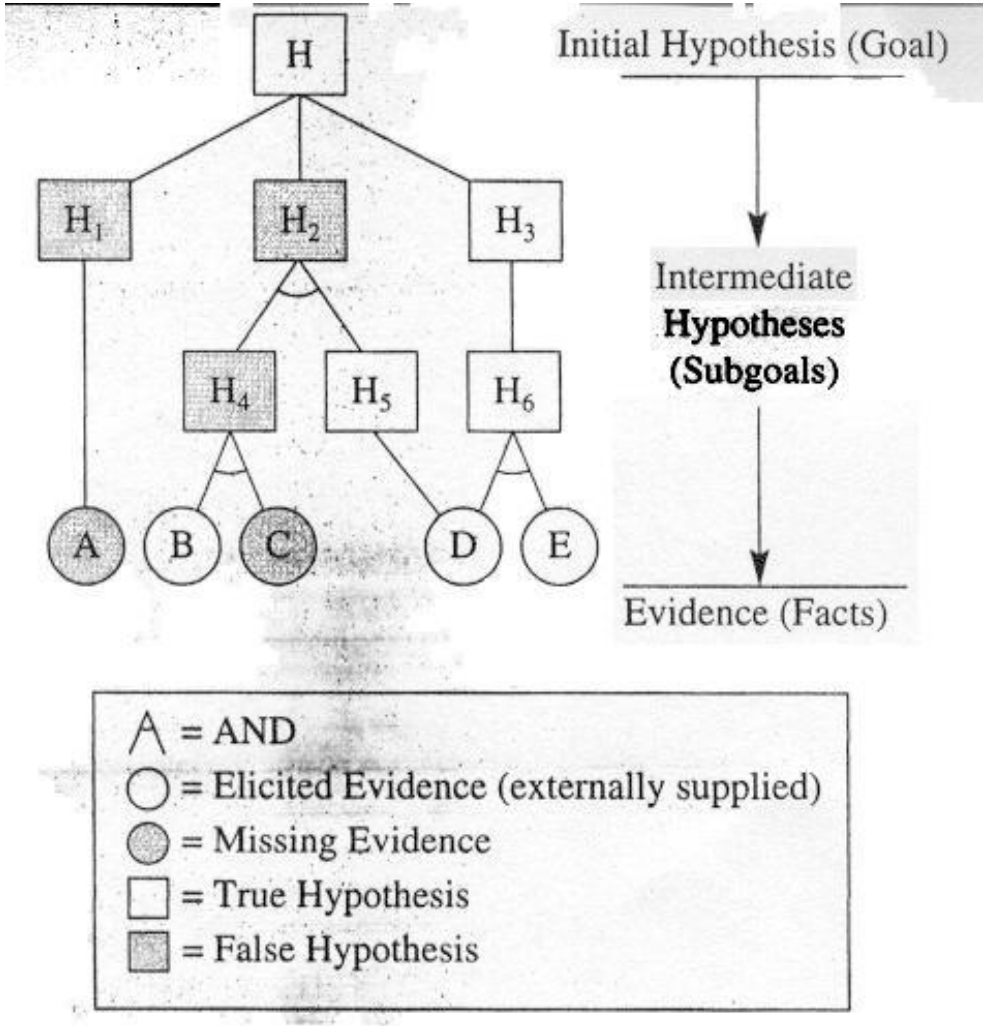
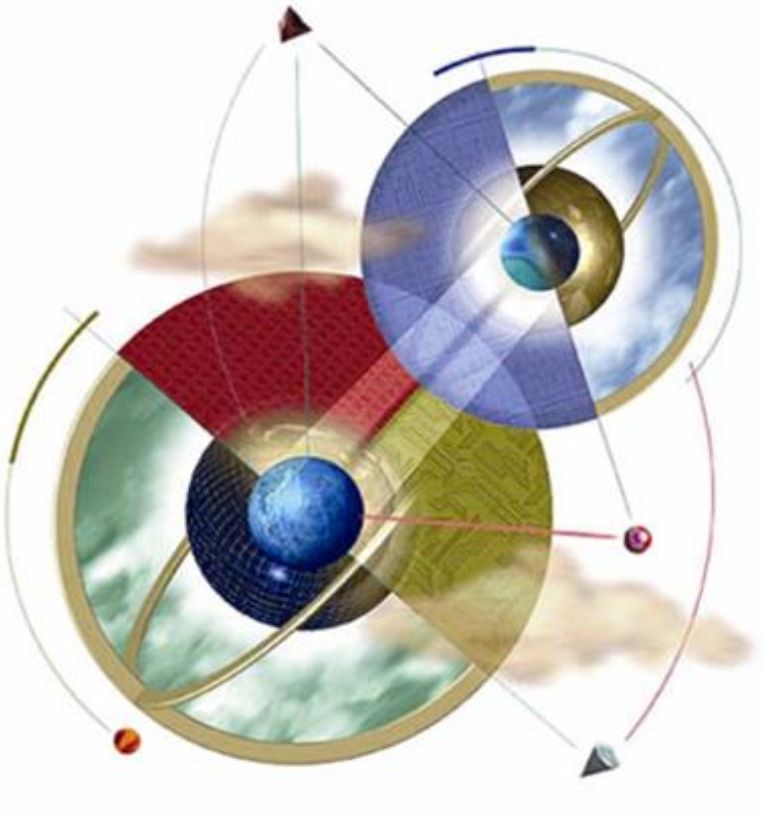


Fig. 3.23 Backward Chaining (Top-Down Reasoning)



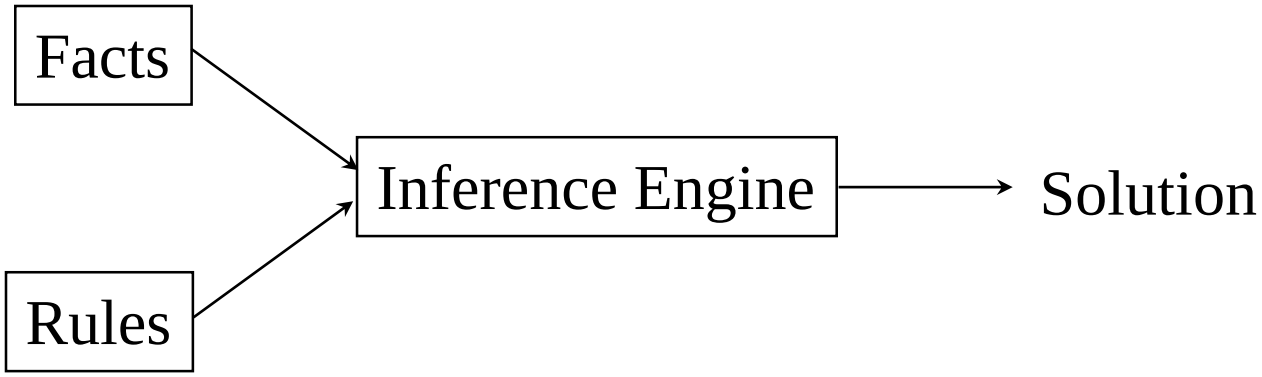


Chapter 7: Introduction to CLIPS

Expert Systems: Principles and Programming, Fourth Edition

What is CLIPS?

- ***C L***anguage ***I***ntegrated ***P***roduction ***S***ystem (CLIPS)
- CLIPS was designed using the C language at the NASA/Johnson Space Center
- Three basic components:
  - fact list
  - knowledge base (rules)
  - inference engine



Installing CLIPS IDE

- Download CLIPS version 6.4.1 (2023/04/08)
  - https://sourceforge.net/projects/clipsrules/files/CLIPS/6.40/
  - Download Latest Version & Install
- Tutorial of CLIPS
  - https://ryjo.codes/tour-of-clips.html
- CLIPS V6.2.2 & FuzzyCLIPS V6.1.0
  - two executable files in desktop PCs

Fields

- To build a knowledge base, CLIPS must read input from keyboard / files to execute commands and load programs.
- During the execution process, CLIPS groups symbols together into tokens (fields).

Numeric Fields & Symbol Fields

- The floats and integers make up the numeric fields – simply numbers.
  - Integers have only a sign and digits.
    - 20 or -17
  - Floats have a decimal and possibly “e” for scientific notation.
    - -1.5 or 3.5e10
- Symbols begin with printable ASCII characters followed by zero or more characters.
  - CLIPS is case sensitive

String Fields

- Strings must begin and end with double quotation marks (“).
  - “Joe Smith”
- Spaces within the string are significant.
  - “space “ ≠ “ space”
- The actual delimiter symbols can be included in a string by preceding the character with a backslash (\).
  - “\”tokens\”” or “\\token\\”

Entering / Exiting CLIPS

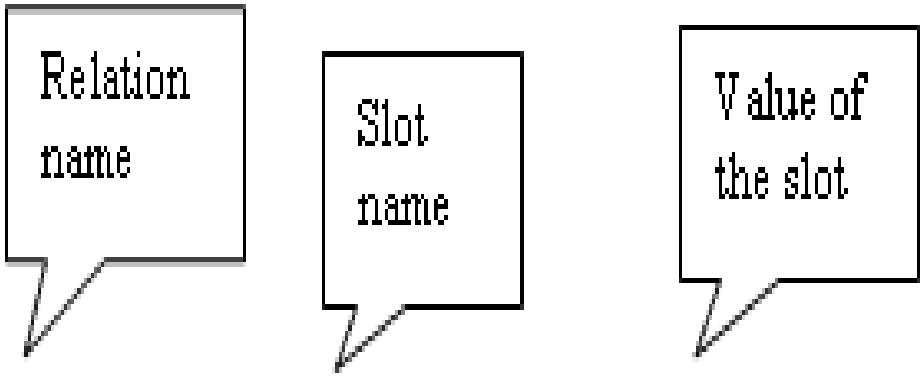
- The CLIPS prompt is: CLIPS>
- This is the type-level mode where commands can be entered.
- To exit CLIPS, one types: CLIPS> (exit) ↵
- CLIPS will accept input from the user / evaluate it / return an appropriate response:

  CLIPS> (+ 3 4) ↵ → value 7 would be returned.

Facts and CLIPS

- To solve a problem, CLIPS must have data or information with which to reason.
- Each chunk of information is called a fact.
- Facts consist of:
  - Relation name (a symbol field)
  - Zero or more slots with associated values

Example Fact in CLIPS




```
(person    (name  "John Q. Public")  ← note the quotes around the name
           (age  23)
           (eye-color  blue)
           (hair-color  black))
```

deftemplate

- Before facts can be constructed, CLIPS must be informed of the list of valid slots for a given relation name.
- A deftemplate is used to describe groups of facts sharing the same relation name and contain common information.

deftemplate General Format

```
(deftemplate <relation-name> [<optional-comment>]
   <slot-definition>*)
```

```
   (slot <slot-name>) | (multislot <slot-name>)
```

```
(deftemplate person "An example deftemplate"
    (slot name)
    (slot age)
    (slot eye-color)
    (slot hair-color))
```

Deftemplate & Facts

- (person (name Peter) (age 12) (height 140) (weight 40))
  (person (name John) (age 30) (height 175) (weight 65))

person	name	age	height	weight
	Peter	12	140	40
	John	30	175	65

- (deftemplate <relation-name> <slot-def>*)
  - (deftemplate person (slot name) (slot age) (slot height) (slot weight))
  - (deftemplate team (slot id) (multislot members))
    - (team (id 01) (members Joe Mary Peter))
    - (team (id 02) (members Frank John Sue David Kevin))
    - (team (id 03) (members))

Deftemplate vs. Ordered Facts

- Facts with a relation name defined using deftemplate are called *deftemplate facts*.
  - (person (name Joe) (age 20))
- Facts with a relation name that does not have a corresponding deftemplate are called *ordered facts* – have a single implied multifield slot for storing all the values of the relation name.
  - (person Joe John Kevin)

Adding Facts

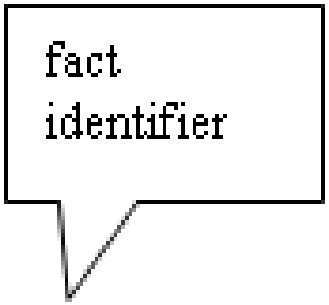
- CLIPS store all facts known to it in a fact list.
- To add a fact to the list, we use the *assert* command.

```
(deftemplate student
    (slot  name)
    (slot  age)
    (slot major))

(assert  (student (name  "John Summers")
                  (age  19)
                  (major  "Information Technology")))
```

Displaying Facts

- CLIPS> (facts) ↵




```
f-0 (student    (name  "John Summers")
                (age  19)
                (major  "Information Technology")))

For a total of 1 fact.
```

Removing Facts

- Just as facts can be added, they can also be removed.
- To remove a fact:

```
CLIPS> (retract 2) ↵
```

- Removing facts results in gaps in the fact identifier list.

Modifying Facts

- Slot values of deftemplate facts can be modified using the *modify* command:

Slot values of deftemplate facts can be modified using the *modify* command:

```
(modify <fact-index> <slot-modifier>+)
```

where <slot-modifier> is:

```
(<slot-name>  <slot-value>)
```

For example, we could make the following modification:

```
(modify 0 (age 21))
```

and then request to see the facts again:

```
(facts)↵

f-4 (student    (name  "John Summers")
                (age  21)
                (major  "Information Technology")))

For a total of 1 fact.
```

Results of Modification

- A new fact index is generated because when a fact is modified:
  - The original fact is retracted
  - The modified fact is asserted

- The *duplicate* command is similar to the *modify* command, except it does not retract the original fact.

Watch Command

- The *watch* command is useful for debugging purposes.
  - (watch <watch-item>)
    - watch-item can be facts, rules, activations, or compilations
  - (unwatch <watch-item>)
- If facts are “watched”, CLIPS will automatically print a message indicating an update has been made to the fact list whenever either of the following has been made:
  - Assertion
  - Retraction

Deffacts Construct

- The *deffacts* construct can be used to assert a group of facts.
- Groups of facts representing knowledge can be defined as follows:

  (deffacts <deffacts name> <facts>*)
- The *reset* command is used to assert the facts in a deffacts statement.
  - but all existing facts will be removed (reset)

Functions and Expressions

- CLIPS has the capability to perform calculations.
- The math functions in CLIPS are primarily used for modifying numbers that are used to make inferences by the application program.

Table 8.1 CLIPS Elementary Arithmetic Operators

Arithmetic Operators	Meaning
+	Addition
–	Subtraction
*	Multiplication
/	Division

Numeric Expressions in CLIPS

- Numeric expressions are written in CLIPS in LISP-style – using prefix form – the operator symbol goes before the operands to which it pertains.
- Example #1:

  5 + 8 (infix form) → + 5 8 (prefix form)

- Example #2:

  (infix) (y2 – y1) / (x2 – x1) > 0

  (prefix) (> (/ (- y2 y1) (- x2 x1)) 0)

Return Values

- Most functions (or operators) have a return value that can be an integer, float, symbol, string, or multivalued value.
- Some functions (*facts*, *agenda* commands) have no return values – just side effects.
- Return values for +, -, and * will be integer if all arguments are integer, but if at least one value is floating point, the value returned will be float.
- Return values for / will be float regardless of all arguments being integer.

Variable Numbers of Arguments

- Many CLIPS functions accept variable numbers of arguments.
- Example:

  CLIPS> (- 3 5 7) ↵ returns `3 - 5 =-2 - 7 = -9`
- There is no built-in arithmetic precedence in CLIPS – everything is evaluated from left to right.
- To compensate for this, precedence must be explicitly written.
  - (- (- 3 5) 7) or (- 3 (- 5 7))

Embedding Expressions

- Expressions may be freely embedded within other expressions:

```
CLIPS>  (assert (result (+ 3 6)))↵
<Fact-0>
CLIPS> (facts)↵
f-0 (result 9)
For a total of 1 fact.

CLIPS>  (assert (expression 3 + 6 * 10))↵
<Fact-0>
CLIPS> (facts)↵
f-0 (expression 3 + 6 * 10)
For a total of 1 fact.
```

The Components of a Rule

- To accomplish work, an expert system must have rules as well as facts.
- Rules can be typed into CLIPS (or loaded from a file).
- Consider the pseudocode for a possible rule:

IF the emergency is a fire

THEN the response is to activate the sprinkler system

Rule Components

- First, we need to create the deftemplate for the types of facts:

  (deftemplate emergency (slot type))

  -- type would be fire, flood, etc.

- Similarly, we must create the deftemplate for the types of responses:

  (deftemplate response (slot action))

  -- action would be “activate the sprinkler”

Rule Components

- The rule would be shown as follows:

```
(defrule fire-emergency “An example rule”
        (emergency (type fire))
        =>
        (assert (response (action activate-sprinkler-system))))
```

Analysis of the Rule

- The header of the rule consists of three parts:
  1. Keyword *defrule*
  2. Name of the rule – *fire-emergency*
  3. Optional comment string – "An example rule"
- After the rule header are 1+ conditional elements (pattern CEs)
- Each pattern consists of 1+ constraints intended to match the fields of the deftemplate fact

Rules

- (defrule <rule-name>
  <patterns>* ------LHS
  =>
  <actions>*) ------RHS
  - (defrule fire-emergency-1
    (emergency (type fire))
    =>
    (assert (response (action sprinkle))))
  - (defrule fire-emergency-2
    (emergency (type fire))
    (fired-object (material oil))
    =>
    (printout t “The fire should be excluded from air.” crlf))

Analysis of Rule

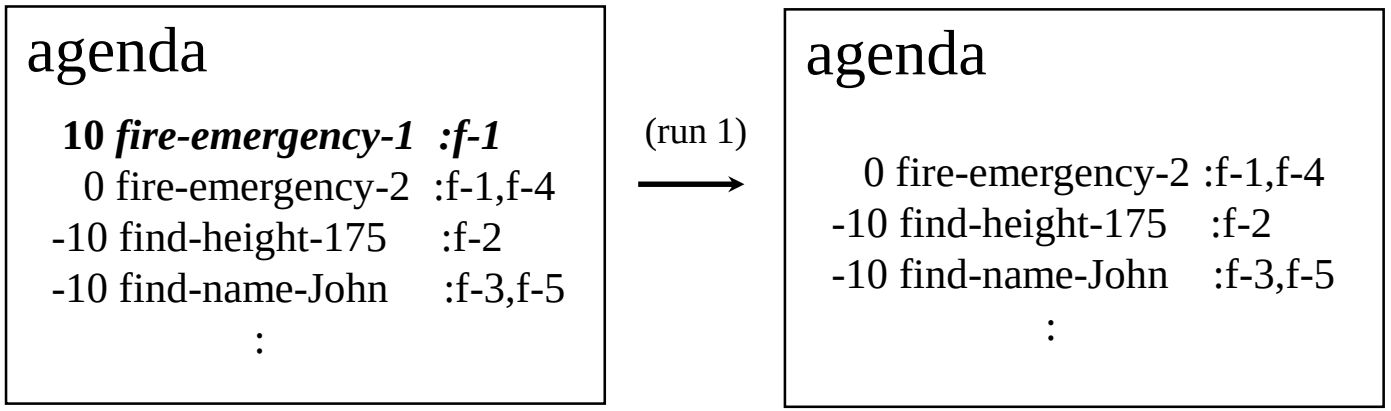
- If all the patterns of a rule match facts, the rule is activated and put on the agenda.
- The agenda is a collection of activated rules.
- The arrow => represents the beginning of the THEN part of the IF-THEN rule (RHS).
- The last part of the rule is the list of actions that will execute when the rule fires.

The Agenda and Execution

- To run the CLIPS program, use the *run* command:

  CLIPS> (run [<limit>])↵

  -- the optional argument <limit> is the maximum number of rules to be fired – if omitted, rules will fire until the agenda is empty.



Execution

- When the program runs, the rule with the highest *salience* on the agenda is fired.
- Rules become activated whenever all the patterns of the rule are matched by facts.
- The *reset* command is the key method for starting or restarting .
- Facts asserted by a *reset* satisfy the patterns of one or more rules and place activation of these rules on the agenda.

What is on the Agenda?

- To display the rules on the agenda, use the agenda command:

  CLIPS> (agenda) ↵

- *Refraction* is the property that rules will not fire more than once for a specific set of facts.
- The *refresh* command can be used to make a rule fire again by placing all activations that have already fired for a rule back on the agenda.
  - (refresh <rule-name>)

Commands (cont.)

- The *printout* command can be used to print information.
  - (printout <logical-name> <print-items>*)
    - the logical name t for standard output device
    - crlf forces a line feed
    - (printout t "The fire should be excluded from air." crlf))
- *set-break* – allows execution to be halted *before* any rule from a specified group of rules is fired.
  - (set-break <rule-name>)
    - a debugging command to set a breakpoint
  - (remove-break [<rule-name>])
    - remove one or all breakpoints

Commands (cont.)

- *Load* – allows loading of rules from an external file.
  - (load <file-name>)
    - (load “c:\\homework\\ex1.clp”)
- *Save* – opposite of load, allows saving of current constructs stored in CLIPS to disk
  - (save <file-name>)
    - (save “c:\\homework\\ex2.clp”)

Execute a CLIPS Program

1. Edit a CLIPS program file in text format (ANSI ASCII)
   - include deftemplate, deffacts, and defrule
2. Open CLIPS system and load the program file
   - (load <file-name>) or [File] → [Load]
3. Reset the CLIPS system
   - (reset) or [Execution] → [Reset]
4. Execute
   - (run) or [Execution] → [Run]
5. Clear the CLIPS system
   - (clear) or [Execution] → [Clear CLIPS]

Fire-Emergency Expert System

- A fire-emergency expert system for dealing with four types of fire-emergency:
  - Type A： the firing material is paper, wood, or cloth
  - Type B： the firing material is oil or gas
  - Type C： the firing material is battery
  - Type D： the firing material is chemicals
- Dealing with the four types of fire-emergency
  - Type A：general extinguisher or water
  - Type B：foam extinguisher or carbon dioxide extinguisher
  - Type C：dry chemicals extinguisher or carbon dioxide extinguisher
  - Type D：graphitized coke

The CLIPS Program

```
(deftemplate fire (slot type))

(deftemplate fired-object (slot material))

(defrule fire-type-A-1
        (fired-object (material paper))
        =>
        (assert (fire (type A))))

(defrule fire-type-A-2
        (fired-object (material wood))
        =>
        (assert (fire (type A))))

(defrule fire-type-A-3
        (fired-object (material cloth))
        =>
        (assert (fire (type A))))

(defrule fire-type-B-1
        (fired-object (material oil))
        =>
        (assert (fire (type B))))

(defrule fire-type-B-2
        (fired-object (material gas))
        =>
        (assert (fire (type B))))

(defrule fire-type-C-1
        (fired-object (material battery))
        =>
        (assert (fire (type C))))

(defrule fire-type-D-1
        (fired-object (material chemicals))
        =>
        (assert (fire (type D))))
```

The CLIPS Program (cont.)

```
(defrule deal-with-type-A
        (fire (type A))
        =>
       (printout t "general extinguisher or water" crlf))

(defrule deal-with-type-B
        (fire (type B))
        =>
       (printout t "foam extinguisher or carbon dioxide extinguisher" crlf))

(defrule deal-with-type-C
       (fire (type C))
        =>
      (printout t "dry chemicals extinguisher or carbon dioxide extinguisher" crlf))

(defrule deal-with-type-D
       (fire (type D))
        =>
       (printout t "graphitized coke" crlf))
```

- input fact: (assert (fired-object (material paper)))
- or default fact: (deffacts initial (fired-object (material paper)))

Commenting and Variables

- Comments – provide a good way to document programs to explain what constructs are doing.
  - begins with a semicolon (;) and ends with a carriage return
- Variables – store values, syntax requires preceding with a question mark (?) or ($?)
  - ?var
    - single-field variable <=> slot
  - $?var
    - multiple-field variable <=> multislot

Single-Field Wildcards, Multifield Wildcards, and Fact Address

- Single-field wildcards can be used in place of variables when the field to be matched against can be anything and its value is not needed later in the LHS or RHS of the rule.
  - ?
- Multifield wildcards allow matching against more than one field in a pattern.
  - $?
- A variable can be bound to a fact address of a fact matching a particular pattern on the LHS of a rule by using the pattern binding operator "<-".
  - ?abc <- (person (name David) (age 18) (weight 70))

Examples

- (defrule find-height-175
  (person (name ?name) (age ?) (height 175) (weight ?w))
  =>
  (printout t ?name “is 175 cm and “ ?w “ kg.”))
  - f-12 (person (name David) (age 18) (height 175) (weight 70))
  - f-15 (person (name John) (age 30) (height 175) (weight 65))
- (defrule find-height-175-and-is-father
  (person (name ?name) (age ?) (height 175) (weight ?w))
  (father-child (father ?name) (child $?))
  =>
  (printout t ?name “is 175 cm and “ ?w “ kg.”))
  - f-12 (person (name David) (age 18) (height 175) (weight 70))
  - f-15 (person (name John) (age 30) (height 175) (weight 65))
  - f-20 (father-child (father John) (child Mary Sue Joe))

Multifield Variables

- a multifield variable is referred to on the RHS
  - not necessary to include the $ as part of the variable name on the RHS
  - the $ is only used on the LHS to indicate the zero or more fields
  - (defrule print-children

    (person (name $?name) (children $?children))

    =>

    (printout t ?name " has children " ?children crlf))
  - the multifield values are surrounded by parentheses when printed
    - (John H. Smith) has children (Joe Paul Mary)

The *OR* Conditional Element

Consider the two rules:

```
(defrule shut-off-electricity-1
    (emergency (type flood))
    =>
    (printout t "Shut off the electricity" crlf))

(defrule shut-off-electricity-2
    (extinguisher-system (type water-sprinkler
                         (status on))
    =>
    (printout t "Shut off the electricity" crlf))
```

OR Conditional Element

These two rules can be combined into one rule – *or* CE requires only one CE be satisfied:

```
(defrule shut-off-electricity)
    (or (emergency (type flood))
        (extinguisher-system (type water-sprinkler)
                             (status on)))
    =>
    (printout t "Shut off the electricity" crlf))
```

The *And* Conditional Element

The *and* CE is opposite in concept to the *or* CE – requiring all the CEs be satisfied:

```
(defrule shut-off-electricity
    (and ?power <- (electrical-power (status on))
         (emergency (type flood)))
    =>
    (modify ?power (status off))
    (printout t "Shut off the electricity" crlf))
```

Not Conditional Element

When it is advantageous to activate a rule based on the absence of a particular fact in the list, CLIPS allows the specification of the absence of the fact in the LHS of a rule using the *not* conditional element:

```
IF the monitoring status is to be reported and
   there is an emergency being handled
THEN report the type of the emergency

IF the monitoring status is to be reported and
   there is no emergency being handed
THEN report that no emergency is being handled
```

Not Conditional

We can implement this as follows:

```
(defrule report-emergency
    (report-status)
    (emergency (type ?type))
    =>
    (printout t "Handling " ?type " emergency"
            crlf))

(defrule no-emergency
    (report-status)
    (not (emergency))
    =>
    (printout t "No emergency being handled" crlf))
```

A Complex Example

- (defrule can-not-drive-in-Taiwan
  (query ?person)
  (not (or (driving-license (id ?) (country Taiwan) (name ?person))
  (and (driving-license (id ?) (country ?else) (name ?person))
  (DL-accepted-in-Taiwan ?else)))))
  =>
  (printout t ?person " can not drive a car in Taiwan." crlf))

f-3 (query John)
f-4 (query Kevin)
f-5 (query Joe)
f-6 (driving-license (id 85346) (country Taiwan) (name Kevin))
f-7 (driving-license (id 91653) (country Japan) (name Joe))
f-8 (driving-license (id 53861) (country America) (name John))
f-9 (DL-accepted-in-Taiwan America)

Retract Facts

- (retract <fact-address>+)
- (defrule add-sum
  ?data <- (data (item ?value))
  ?old-total <- (total ?total)
  =>
  (retract ?old-total ?data)
  (assert (total (+ ?total ?value))))
  - f-2 (total 0)
  - f-4 (data (item 20))
  - f-5 (data (item 15))
  - f-8 (data (item 12))

Infinite Loop

```
(defrule process-moved-information
    (moved (name ?name) (address ?address))
    ?f2 <- (person (name ?name) (address ?))
    =>
    (modify ?f2 (address ?address)))
```

- The program executes an infinite loop
  - it asserts a new person fact after modification
  - reactivate the process-moved-information rule

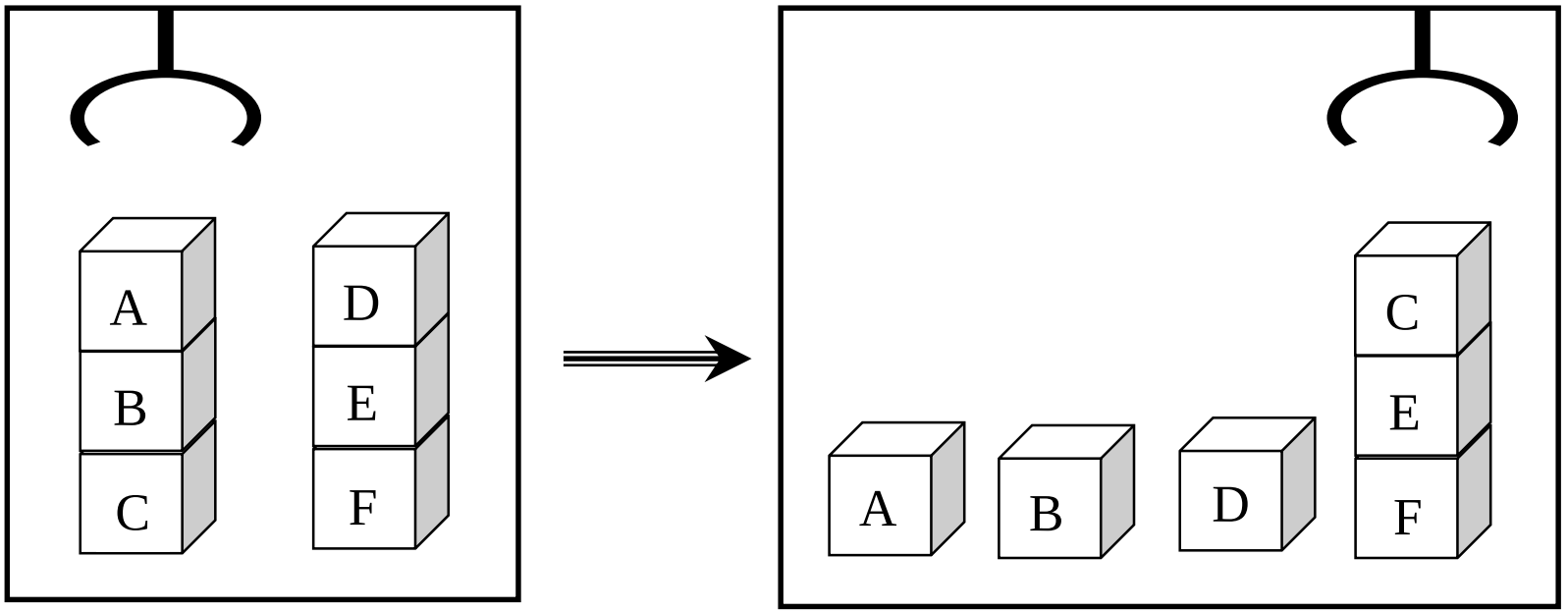
Salience

- (declare (salience 100))
  - the priority to fire rules (-10000 ... 10000)
  - default is 0
  - (defrule rule-01
    (declare (salience 50))
    (person (name ?name) (age 20))
    =>
    (printout t ?name " is young." crlf))
    (defrule rule-02
    (declare (salience 20))
    (person (name ?name) (age 20))
    =>
    (printout t ?name " is 20 years old." crlf))

f-5 (person (name Peter) (age 20))
f-6 (person (name David) (age 20))

Blocks World

- goal: to move one block on top of another block
  - to move C on top of E



(Initial State)

(Goal State)

The CLIPS Program for Blocks World

```
(deffacts initial-state
   (block A)
   (block B)
   (block C)
   (block D)
   (block E)
   (block F)
   (on-top-of (upper nothing) (lower A))
   (on-top-of (upper A) (lower B))
   (on-top-of (upper B) (lower C))
   (on-top-of (upper C) (lower floor))
   (on-top-of (upper nothing) (lower D))
   (on-top-of (upper D) (lower E))
   (on-top-of (upper E) (lower F))
   (on-top-of (upper F) (lower floor))
   (goal (move C) (on-top-of E))
)
```

```
(deftemplate on-top-of (slot upper) (slot lower))

(deftemplate goal (slot move) (slot on-top-of))

(defrule move-directly
   ?goal <- (goal (move ?b1) (on-top-of ?b2))
   (block ?b1)
   (block ?b2)
   (on-top-of (upper nothing) (lower ?b1))
   ?stack1 <- (on-top-of (upper nothing) (lower ?b2))
   ?stack2 <- (on-top-of (upper ?b1) (lower ?b3))
 =>
   (retract ?goal ?stack1 ?stack2)
   (assert (on-top-of (upper ?b1) (lower ?b2)))
   (assert (on-top-of (upper nothing) (lower ?b3)))
   (printout t ?b1 " move on top of " ?b2 "." crlf)
)
```

The CLIPS Program for Blocks World (cont.)

```
(defrule move-to-floor
   ?goal <- (goal (move ?b1) (on-top-of floor))
   (block ?b1)
   (on-top-of (upper nothing) (lower ?b1))
   ?stack <- (on-top-of (upper ?b1) (lower ?b2))
 =>
   (retract ?goal ?stack)
   (assert (on-top-of (upper ?b1) (lower floor)))
   (assert (on-top-of (upper nothing) (lower ?b2)))
   (printout t ?b1 " move on top of floor." crlf)
)
```

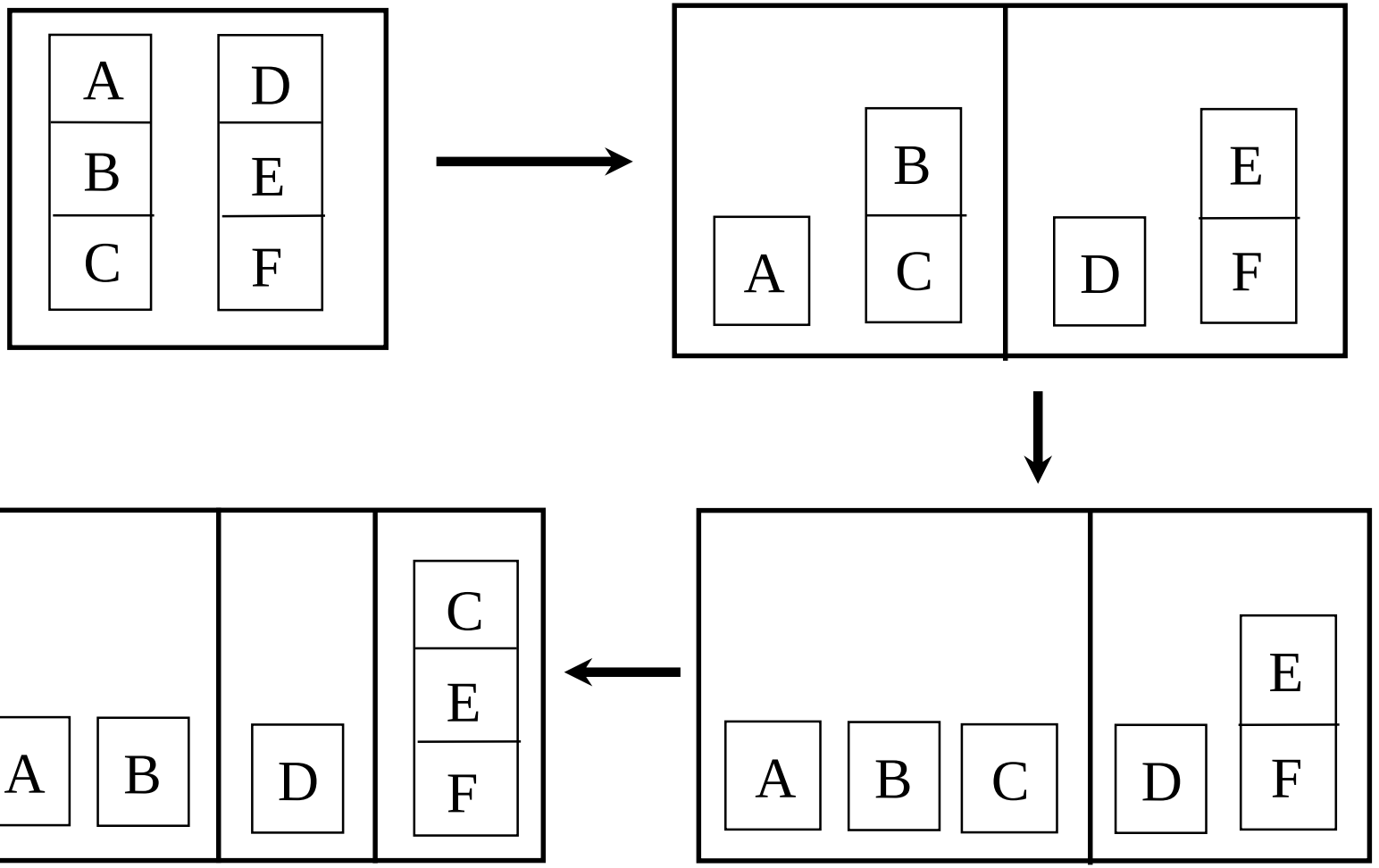
```
(defrule clear-move-block
   (goal (move ?b1) (on-top-of ?))
   (block ?b1)
   (block ?b2)
   (on-top-of (upper ?b2) (lower ?b1))
 =>
   (assert (goal (move ?b2) (on-top-of floor)))
)
```

```
(defrule clear-place-block
   (goal (move ?) (on-top-of ?b1))
   (block ?b1)
   (block ?b2)
   (on-top-of (upper ?b2) (lower ?b1))
 =>
   (assert (goal (move ?b2) (on-top-of floor)))
)
```

Results:

A move on top of floor.
B move on top of floor.
D move on top of floor.
C move on top of E.

Trace



Trace (cont.)

Cycle	Stack-1	Stack-2	Goal	Rule
initial	ABC	DEF	C→E	
1			B→Floor D→Floor	clear-move-block clear-place-block
2			A→Floor	clear-move-block
3	~~ABC~~ A, BC		~~A→Floor~~	move-to-floor
4	~~BC~~ B, C		~~B→Floor~~	move-to-floor
5		~~DEF~~ D, EF	~~D→Floor~~	move-to-floor
6	~~C~~	~~EF~~ CEF	~~C→E~~	move-directly

Blocks World Revisited

- Using multifield variables and wildcards to re-implement the blocks world problem

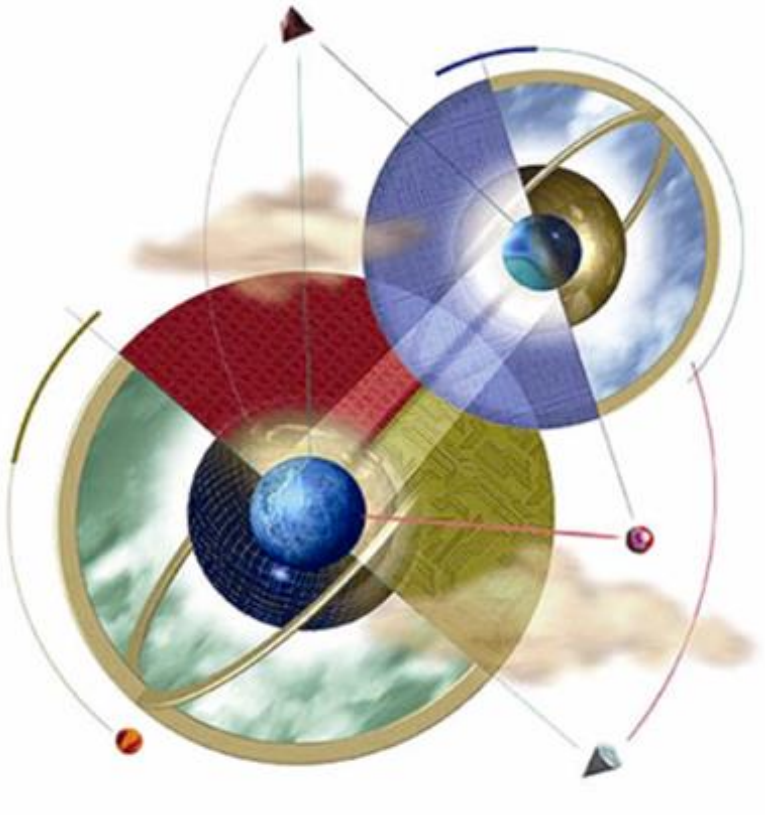
```
(deftemplate goal (slot move) (slot on-top-of))
(deffacts initial-state
    (stack A B C)
    (stack D E F)
    (goal (move C) (on-top-of E))
)
(defrule move-directly
   ?goal <- (goal (move ?block1) (on-top-of ?block2))
   ?stack-1 <- (stack ?block1 $?rest1)
   ?stack-2 <- (stack ?block2 $?rest2)
=>
   (retract ?goal ?stack-1 ?stack-2)
   (assert (stack $?rest1))
   (assert (stack ?block1 ?block2 $?rest2))
   (printout t ?block1 " move on top of " ?block2 "." crlf))
```

Blocks World Revisited (cont.)

```
(defrule move-to-floor
   ?goal <- (goal (move ?block1) (on-top-of floor))
   ?stack-1 <- (stack ?block1 $?rest)
=>
   (retract ?goal ?stack-1)
   (assert (stack ?block1))
   (assert (stack $?rest))
   (printout t ?block1 " move on top of floor." crlf))
```

```
(defrule clear-move-block
   (goal (move ?block1))
   (stack ?top $? ?block1 $?)
=>
   (assert (goal (move ?top) (on-top-of floor))))
```

```
(defrule clear-place-block
   (goal (on-top-of ?block1))
   (stack ?top $? ?block1 $?)
=>
   (assert (goal (move ?top) (on-top-of floor))))
```



Chapter 8: Advanced Pattern Matching

Expert Systems: Principles and Programming, Fourth Edition

Field Constraints

- In addition to pattern matching capabilities and variable bindings, CLIPS has more powerful pattern matching operators – field constraints.
- Consider writing a rule for all people who do *not* have brown hair:
  - We could write a rule for every type of hair color that is not brown.
    - add another pattern CEs to test the variable ?color
    - or, place a field constraint as the value of the slot directly
    - or, attach a field constraint to the variable ?color

Connective Constraints

- Connective constraints are used to connect variables and other constraints.
  - *Not field constraint* – the ~ acts on the one constraint or variable that immediately follows it.
    - (defrule person-without-brown-hair
      (person (name ?name) (hair ?color))
      (test (neq ?color brown))
      =>
      (printout t ?name " does not have brown hair." crlf))
    - (defrule person-without-brown-hair
      (person (name ?name) (hair ~brown))
      =>
      (printout t ?name " does not have brown hair." crlf))

Connective Constraints (cont.)

- *Or field constraint* – the symbol | is used to allow one or more possible values to match a field or a pattern.
  - (defrule person-with-black-or-red-hair
    (person (name ?name) (hair black | red))
    =>
    (printout t ?name " has black or red hair." crlf))
- *And field constraint* – the symbol & is useful with binding instances of variables and on conjunction with the *not* constraint
  - (defrule person-without-black-or-red-hair
    (person (name ?name) (hair ?color&~black&~red))
    =>
    (printout t ?name " has " ?color " hair." crlf))

Combining Field Constraints

- Field constraints can be used together with variables and other literals to provide powerful pattern matching capabilities.
- Example #1: ?eyes1&blue|green
  - This constraint binds the person's eye color to the variable, ?eyes1 if the eye color of the fact being matched is either blue or green.
- Example #2: ?hair1&~black&~red
  - This constraint binds the variable ?hair1 if the hair color of the fact being matched is not black.

Complex Field Constraints

- For example, a rule to determine whether the two people exist:
  - The first person has either blue or green eyes and does not have black hair
  - The second person does not have the same color eyes as the first person and has either red hair or the same color hair as the first person
- (defrule complex-match

  (person (name ?name1) (eyes ?eyes1&blue|green)(hair ?hair1&~black))

  (person (name ?name2&~?name1) (eyes ?eyes2&~?eyes1)
  (hair ?hair2&red | ?hair1))

  =>

  (printout t ?name1 " has " ?eyes1 " eyes and " ?hair1 " hair." crlf)

  (printout t ?name2 " has " ?eyes2 " eyes and " ?hair2 " hair." crlf))

Predicate Functions

- A predicate function is defined to be any function that returns:
  - TRUE
  - FALSE
- Any value other than FALSE is considered TRUE.
- We say the predicate function returns a Boolean value.

The *Test Conditional* Element

- Processing of information often requires a loop.
- Sometimes a loop needs to terminate automatically as the result of an arbitrary expression.
- The *test condition* provides a powerful way to evaluate expressions on the LHS of a rule.
- Rather than pattern matching against a fact in a fact list, the *test CE* evaluates an expression – outermost function must be a predicate function.

Test Condition

- A rule will be triggered only if all its test CEs are satisfied along with other patterns.

  (test <predicate-function>)

  Example:

  (test (> ?value 1))

Examples for *Test Condition*

- (defrule find-height-larger-than-170
  (person (name ?name) (age ?) (height ?height) (weight ?))
  (test (> ?height 170))
  =>
  (printout t ?name "'s height is larger than 170 cm." crlf))
- (defrule find-person
  (person (name ?name) (age ?age) (height ?height) (weight ?weight))
  (test (and (>= ?height 170)
  (or (> ?weight 60)
  (< ?age 30))))
  =>
  (printout t ?name " is the person we seek." crlf))

f-5 (person (name Peter) (age 35) (height 175) (weight 60))
f-6 (person (name David) (age 25) (height 170) (weight 55))

Predicate Field Constraint

- The predicate field constraint **:** allows for performing predicate tests *directly within patterns*.
  - The predicate field constraint is *more efficient* than using the test CE.

```
(defrule find-height-larger-than-170
(person (name ?name) (age ?) (height ?height&:(> ?height 170)) (weight ?))
=>
(printout t ?name "'s height is larger than 170 cm." crlf))
```

```
f-5 (person (name Peter) (age 35) (height 175) (weight 60))
f-6 (person (name David) (age 25) (height 170) (weight 55))
f-7 (person (name John) (age 12) (height 145) (weight 40))
f-8 (person (name Kevin) (age 31) (height 200) (weight 98))
```

Return Value Constraint

- The return value constraint = allows the return value of a function to be used for comparison inside LHS patterns
  - The function must have a single-field return value.
- (defrule find-one-is-30cm-taller-than-another
  (person (name ?name1) (age ?) (height ?height) (weight ?))
  (person (name ?name2) (age ?) (height =(+ ?height 30)) (weight ?))
  =>
  (printout t ?name2 " is 30 cm taller than " ?name1 crlf))

f-5 (person (name Peter) (age 35) (height 175) (weight 60))
f-6 (person (name David) (age 25) (height 170) (weight 55))
f-7 (person (name John) (age 12) (height 145) (weight 40))
f-8 (person (name Kevin) (age 31) (height 200) (weight 98))

Summing Values Using Rules

- Suppose you wanted to sum the areas of a group of rectangles.
  - The heights and widths of the rectangles can be specified using the deftemplate:

```
(deftemplate rectangle (slot height) (slot width))
```

- The sum of the rectangle areas could be specified using an ordered fact such as:

```
(sum 20)
```

Summing Values

- A deffacts containing sample information is:

```
(deffacts initial-information
   (rectangle (height 10) (width 6))
   (rectangle (height 7) (width 5)
   (rectangle (height 6) (width 8))
   (rectangle (height 2) (width 5))
   (sum 0))
```

- to permit rectangles of same size
  - (rectangle (id 12) (height 10) (width 6))

Summing Values

```
(deftemplate rectangle (slot height) (slot width))

(deffacts initial-rectangles
  (rectangle (height 10) (width 6))
  (rectangle (height 8) (width 5))
  (rectangle (height 3) (width 7))
  (sum 0))

(defrule compute-area
  (rectangle (height ?height) (width ?width))
  =>
  (assert (area (* ?height ?width) ) ) )

(defrule sum-1
  ?sum <- (sum ?total)
  ?new-area <- (area ?area)
  =>
  (retract ?sum ?new-area)
  (assert (sum (+ ?total ?area) ) )
  (printout t "The new sum is " (+ ?total ?area) crlf))
```

Summing Values (cont.)

- (defrule sum-2
  (rectangle (height ?height) (width ?width))
  ?sum <- (sum ?total)
  =>
  (retract ?sum)
  (assert (sum (+ ?total (* ?height ?width))))
  (printout t "The new sum is " (+ ?total (* ?height ?width)) crlf))

What happen??

- (defrule sum-3
  ?rect <- (rectangle (height ?height) (width ?width))
  ?sum <- (sum ?total)
  =>
  (retract ?rect ?sum)
  (assert (sum (+ ?total (* ?height ?width))))
  (printout t "The new sum is " (+ ?total (* ?height ?width)) crlf))

Compare with sum-1

The Bind Function

- Sometimes it is advantageous to store a value in a temporary variable to avoid recalculation.
- The *bind* function can be used to bind the value of a variable to the value of an expression using the following syntax:

  `(bind <variable> <value>)`

- (defrule sum-3
  ?rect <- (rectangle (height ?height) (width ?width))
  ?sum <- (sum ?total)
  =>
  (retract ?rect ?sum)
  (bind ?new (+ ?total (* ?height ?width)))
  (assert (sum ?new))
  (printout t "The new sum is " ?new crlf))

I/O Functions

- When a CLIPS program requires input from the user of a program, a *read* function can be used to provide input from the keyboard:

```
(defrule get-name
    =>
    (printout t "What is your name? ")
    ( bind  ?response (read))
    (assert (user's-name ?response)))
```

Read Function from Keyboard

- The *read* function can only input a single field at a time.
  - CLIPS> (read)

    John K. Smith ↵
- Characters entered after the first field are discarded.
  - K. Smith are discarded
- To input, say a first and last name, they must be delimited with quotes, "John K. Smith".
- Data must be followed by a carriage return (↵) to be read.

I/O from/to Files

- Input can also come from external files.
- Output can be directed to external files.
- Before a file can be accessed, it must be opened using the *open* function:

  Example:

  ```
  (open "c:\\datafile.dat" mydata "r")
  ```

- The *open* function acts as a predicate function
  - Returns true if the file was successfully opened
  - Returns false otherwise

I/O from/to Files (cont.)

```
(open "c:\\datafile.dat" mydata "r")
```

- datafile.dat – is the name of the file (path can also be provided)
- mydata – is the logical name that CLIPS associates with the file
- "r" – represents the mode – how the file will be used – here read access

Table 8.2 File Access Modes

Mode	Action
"r"	Read access only
"w"	Write access only (overwrite all)
"r+"	Read and write access
"a"	Append access only (append to EOF)

Close Function

- Once access to the file is no longer needed, it should be closed.
- Failure to close a file may result in loss of information.
- General format of the *close* function:

  (close [<logical-name>])

  – ex. (close mydata)

Reading / Writing to a File

- Which logical name used, depends on where information will be written – logical name *t* refers to the terminal (standard output device).

Writing to a File	Reading from a File
(open "myresult.dat" stuff "w") ↵	(open "mydata.dat" stuff "r") ↵
(printout stuff "apple" crlf) ↵	(read stuff) ↵
(printout stuff 8 crlf) ↵	(read stuff) ↵
(close stuff) ↵	(read stuff) ↵
	(close stuff) ↵

Formatting

- Output sent to a terminal or file may need to be formatted – enhanced in appearance.
- To do this, we use the *format* function which provides a variety of formatting styles.
- General format:

  (format <logical-name> <control-string> <parameters>*)

  – output the result to the logical name

  – return the result

Formatting (cont.)

- Logical name :
  - *t* indicates standard output device
  - logical name associated with a file
  - *nil* indicates that no output is printed, but the formatted string is still returned
- Control string:
  - Must be delimited with quotes (“)
  - Consists of *format flags* (%) to indicate how parameters should be printed
  - one-to-one correspondence between *flags* and number of *parameters* – constant values or expressions

Formatting

- Example:

  (format nil "Name: %-15s Age: %3d" "Bob Green" 35) ↵

Produces the results and returns:

"Name: Bob Green Age: 35"

– %-15s %3d

- - indicates left-justified
- s indicates string
- 15 , 3 indicates the width
- d indicates integer

Specifications of Format Flag

%-m.Nx

- The "-" means to left justify (right is the default)
- m – total field width
- N – number of digits of precision (default = 6)
- x – display format specification

Table 8.3 Display Format Specifications

Character	Meaning
d	Integer
f	Floating-point
e	Exponential (in power-of-ten format)
g	General (numeric); display in whatever format is shorter
o	Octal; unsigned number (N specifier not applicable)
x	Hexadecimal; unsigned number (N specifier not applicable)
s	String; quoted strings will be stripped of quotes
n	Carriage return/line feed
%	The "%" character itself

Readline Function

- To read an entire line of input, the *readline* function can be used:

  (readline [<logical-name>])

- Example:

```
(defrule get-name
   =>
   (printout t "What is your name? ")
   (bind ?response (readline))
   (assert (user's-name ?response)))
```

  - What is your name? John K. Smith ↵
    - ?response = "John K. Smith"
    - (user's-name "John K. Smith")

explode$ Function

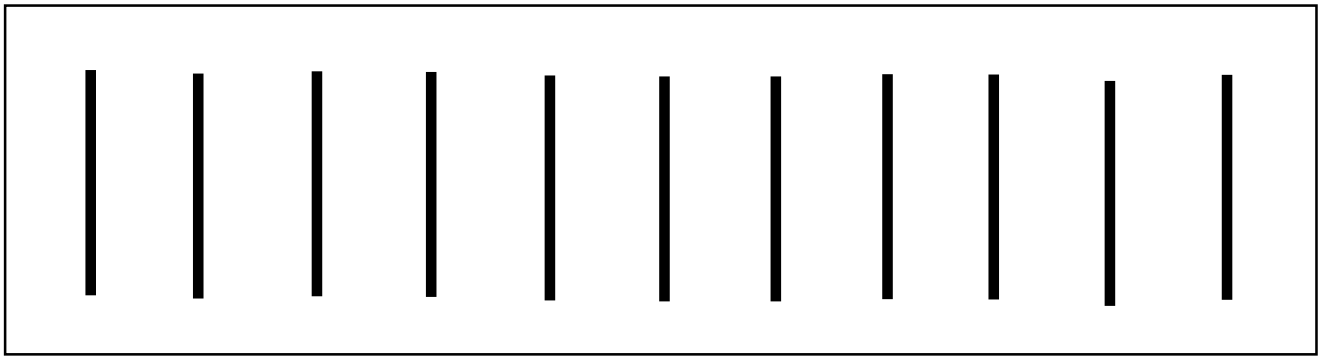
- (readline)
  - read a line (as a string)
- (explode$ <string>)
  - convert a string into a multi-field value

```
(defrule get-name
  =>
  (printout t "What is your name? ")
  (bind ?response (explode$ (readline)))
  (assert (user's-name ?response)))
```

  - What is your name? John K. Smith ↵
    - ?response = John K. Smith
    - (user's-name John K. Smith)

The Game of Sticks

- two player
- a pile of sticks to be taken
- must take 1, 2, or 3
- take the last stick => lose



The CLIPS Program

```
(deftemplate choose (slot get) (slot remainder))

(deffacts initial
   (choose (get 1) (remainder 1))
   (choose (get 1) (remainder 2))
   (choose (get 2) (remainder 3))
   (choose (get 3) (remainder 0)))

(defrule choose-total-number
   (declare (salience 1000))
  =>
   (printout t "How many sticks in the pile? ")
   (assert (total  (read))))

(defrule player-select
   (declare (salience 900))
  =>
   (printout t "Who moves first (Computer:c   Human: h)? ")
   (assert (player-get  (read))))
```

```
(defrule computer-get
   ?player <- (player-get c)
   ?total <- (total ?num)
   (test (>= ?num 1))
   (choose (get ?get) (remainder =(mod ?num 4)))
=>
   (retract ?player ?total)
   (printout t  ?num " sticks in the pile." crlf)
   (printout t "Computer take " ?get " sticks." crlf)
   (assert (total (- ?num ?get)))
   (assert (player-get h)))
```

The CLIPS Program (cont.)

```
(defrule human-get
   ?player <- (player-get h)
   ?total <- (total ?num)
   (test (>= ?num 1))
  =>
   (retract ?player ?total)
   (printout t “There are “ ?num “ sticks in the pile.” crlf)
   (printout t “How many sticks do you wish to take? (1~3) “)
   (assert (total (- ?num (read))))
   (assert (player-get c)))
```

```
(defrule computer-win
   (player-get c)
   (total ?num)
   (test (< ?num 1))
  =>
   (printout t “You lose!” crlf))
```

```
(defrule Human-win
   (player-get h)
   (total ?num)
   (test (< ?num 1))
  =>
   (printout t “You win!” crlf))
```

The Game of Sticks Revisited

```
(deftemplate choose (slot get) (slot remainder))

(deffacts initial
   (phase choose-total)
   (choose (get 1) (remainder 1))
   (choose (get 1) (remainder 2))
   (choose (get 2) (remainder 3))
   (choose (get 3) (remainder 0)))
```

```
(defrule choose-total
   (phase choose-total)
  =>
   (printout t "How many sticks in the pile? ")
   (assert (total  (read))))
```

```
(defrule choose-total-right
   ?phase <- (phase choose-total)
   (total ?total)
   (test (integerp ?total))
   (test (> ?total 0))
=>
   (retract ?phase)
   (assert (phase choose-player)))
```

```
(defrule choose-total-wrong
   ?phase <- (phase choose-total)
   ?total <- (total ?num&:(or (not (integerp ?num))
                              (<= ?num 0)))
=>
   (retract ?phase ?total)
   (assert (phase choose-total))
   (printout t "Please input a positive integer!!" crlf))
```

The Game of Sticks Revisited (cont.)

```
(defrule player-select
   (phase choose-player)
  =>
   (printout t "Who moves first  (Computer:c   Human: h)? ")
   (assert (player-get  (read))))
```

```
(defrule player-select-right
   ?phase <- (phase choose-player)
   (player-get  c | h)
=>
   (retract ?phase)
   (assert (phase play-game)))
```

```
 (defrule player-select-wrong
   ?phase <- (phase choose-player)
   ?p-get <- (player-get ~c & ~h)
=>
   (retract ?phase ?p-get)
   (assert (phase choose-player))
   (printout t "Please input c or h!!" crlf))
```

The Game of Sticks Revisited (cont.)

```
(defrule human-get
  (phase play-game)
  (player-get h)
  (total ?num)
  (test (>= ?num 1))
 =>
  (printout t "There are " ?num " sticks in the pile." crlf)
  (printout t "How many sticks do you wish to take? (1~3) ")
  (assert (human-take (read))))
```

```
(defrule human-get-right
  (phase play-game)
  ?player <- (player-get h)
  ?h-take <- (human-take ?take)
  ?total <- (total ?num)
  (test (and (integerp ?take)
             (>= ?take 1)
             (<= ?take 3)
             (<= ?take ?num)))
 =>
  (retract ?player ?h-take ?total)
  (assert (total (- ?num ?take)))
  (assert (player-get c)))
```

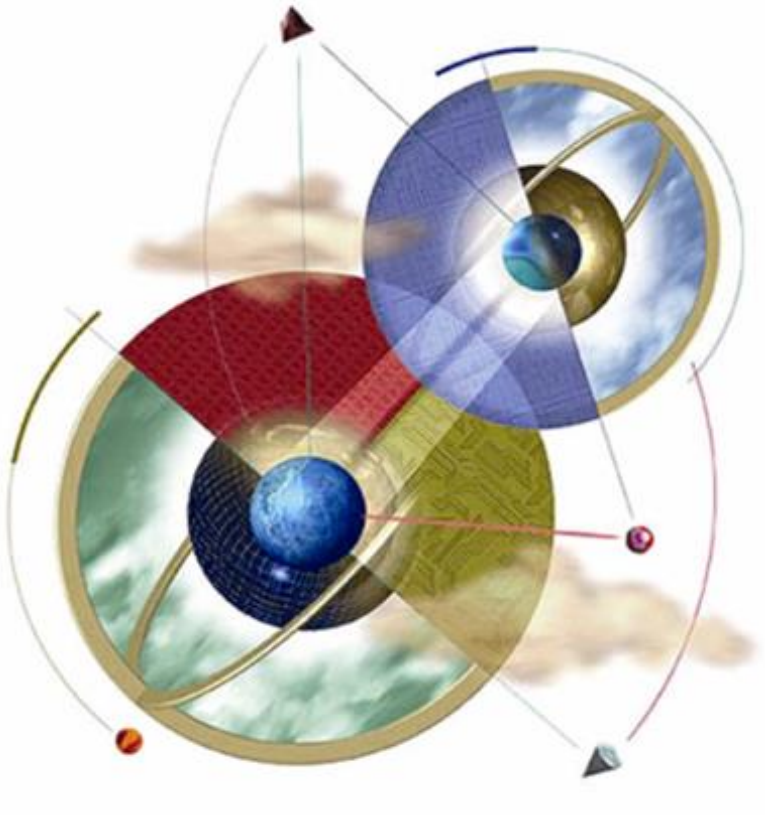
```
(defrule human-get-wrong
  (phase play-game)
  ?player <- (player-get h)
  ?h-take <- (human-take ?take)
  (total ?num)
  (test (or (not (integerp ?take)) (< ?take 1)  (> ?take 3)  (> ?take ?num)))
 =>
  (retract ?player ?h-take)
  (assert (player-get h))
  (printout t "Please input a number (1~3)!!" crlf))
```

The Game of Sticks Revisited (cont.)

```
(defrule computer-get
   (phase play-game)
   ?player <- (player-get c)
   ?total <- (total ?num)
   (test (>= ?num 1))
   (choose (get ?get) (remainder =(mod ?num 4)))
=>
   (retract ?player ?total)
   (printout t  ?num " sticks in the pile." crlf)
   (printout t "Computer take " ?get " sticks." crlf)
   (assert (total (- ?num ?get)))
   (assert (player-get h)))
```

```
(defrule computer-win
   (phase play-game)
   (player-get c)
   (total ?num)
   (test (< ?num 1))
  =>
   (printout t "You lose!" crlf))
```

```
(defrule Human-win
   (phase play-game)
   (player-get h)
   (total ?num)
   (test (< ?num 1))
  =>
   (printout t "You win!" crlf))
```



Chapter 9: Modular Design, Execution Control, and Rule Efficiency

Expert Systems: Principles and Programming, Fourth Edition

Deftemplate Attributes

- CLIPS provides slot attributes which can be specified when deftemplate slots are defined.
- Slot attributes provide strong typing and constraint checking.
- One can define the allowed types that can be stored in a slot, range of numeric values.
- Multislots can specify min / max numbers of fields they can contain.
- Default attributes can be provided for slots not specified in an assert command.

Type Attribute

- Defines the data types can be placed in a slot
- Example:

```
(deftemplate person
  (multislot name (type SYMBOL))
  (slot age (type INTEGER)))
```

- Once defined, CLIPS will enforce these restrictions on the slot attributes

  name – must store symbols

  age – must store integers

Static and Dynamic Constraint Checking

- CLIPS provides two levels of constraint checking (type checking)
  - Static constraint checking
    - Performed when CLIPS parses expression (compile time)
    - Can be enabled/disabled by calling the *set-static-constraint-checking* function and passing it TRUE/FALSE
      - enabled by default
      - (set-static-constraint-checking FALSE)
  - Dynamic constraint checking
    - Performed on facts when they are asserted (run time)
    - Can be enabled / disabled with *set-dynamic-constraint-checking*
      - disabled by default
      - (set-dynamic-constraint-checking TRUE)

Allowed Value Attributes

- CLIPS allows one to specify a list of allowed values for a specific type
- (deftemplate person

  (multislot name (type SYMBOL))

  (slot age (type INTEGER))

  (slot gender (type SYMBOL) (allowed-symbols male female)))
  - allowed-symbols does not restrict the type of the gender slot to being a symbol
    - if the slot's value is a symbol, then it must be either male or female
    - any string, integer, or float would be a legal value if the (type SYMBOL) were removed
    - (allowed-values male female) can be used to completely restrict the slot, and (type SYMBOL) can be removed in this case

Allowed Value Attributes (cont.)

- CLIPS provides several different allowed value attributes
  - allowed-symbols
  - allowed-strings
  - allowed-lexemes
  - allowed-integers
  - allowed-floats
  - allowed-numbers
  - allowed-values

Range Attributes

- This attribute allows the specification of minimum and maximum numeric values.
  - (range <lower-limit> <upper-limit>)
- Example:

```
(deftemplate person
   (multislot name (type SYMBOL))
   (slot age (type INTEGER) (range  0  120)))
```

Cardinality Attributes

- This attribute allows the specification of minimum and maximum number of values that can be stored in a multislot.
  - (cardinality <lower-limit> <upper-limit>)
- Example:

```
(deftemplate volleyball-team
   (slot name (type STRING))
   (multislot players (type STRING) (cardinality 6 6))
   (multislot alternates (type STRING) (cardinality 0 2)))
```

Default Attribute

- Previously, each deftemplate fact asserted had an explicit value stated for every slot.
- It is often convenient to automatically have a specified value stored in a slot if no value is explicitly stated in an *assert* command.
  - (default <default-specification>)
  - Example:
    - (deftemplate example (slot aa (default 3)) (slot bb (default 5))
      (multislot cc (default red green blue)))
    - (assert (example (bb 10)))
      - (example (aa 3) (bb 10) (cc red green blue))

Salience

- CLIPS provides two explicit techniques for controlling the execution of rules:
  - Salience
  - Modules
- Salience allows the priority of rules to be explicitly specified.
- The agenda acts like a stack (LIFO) – most recent activation placed on the agenda being first to fire.

Salience (cont.)

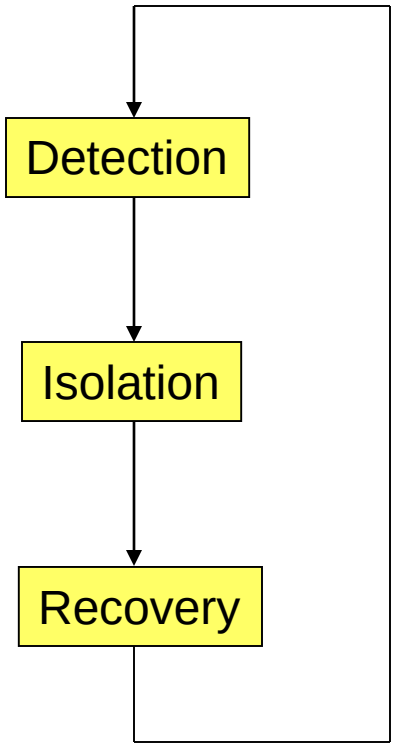
- Salience allows more important rules to stay at the top of the agenda, regardless of when they were added.
- Lower salience rules are pushed lower on the agenda; higher salience rules are higher.
- Salience is set using numeric values in the range -10,000 → +10,000
- Zero is the default priority.
- Salience can be used to force rules to fire in a sequential fashion.
- Rules of equal salience, activated by different patterns are prioritized based on the stack order of facts.

Phases and Control Facts

- The purest concept of expert system: the rules act opportunistically whenever they are applicable
- Most expert systems have some procedural aspect to them
  - game of sticks: control facts (player-get c) to control human's move or computer's move
  - a major problem in development and maintenance: control knowledge is intermixed with domain knowledge
  - domain knowledge and control knowledge should be separated

An Example: Electronic Device Problem

- Different phases for solving electronic device problem
  - fault detection
    - recognize that the electronic device is not working properly
  - isolation
    - determine the components of the device that have caused the fault
  - recovery
    - determine the steps necessary to correct the fault



Implementing the Flow of Control in the Electronic Device System

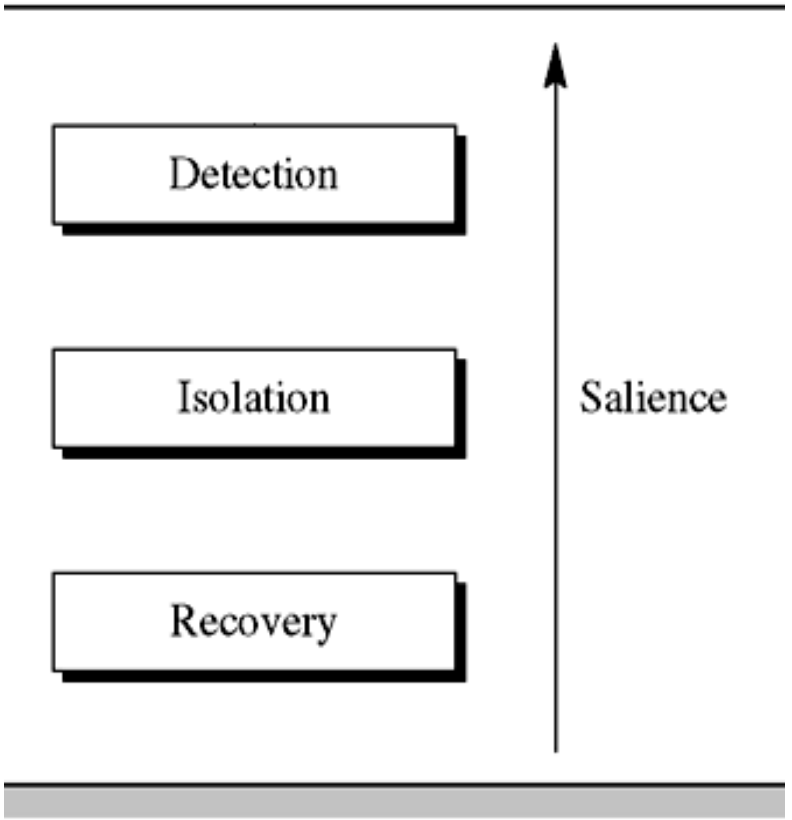
- Four ways to implement the flow of control:
  1. Embed the control knowledge directly into the rules.
  2. Use salience to organize the rules
  3. Separate the control knowledge from the domain knowledge
  4. Use modules to organize the rules (will be discussed latter)

1. Embed the control knowledge directly into the rules

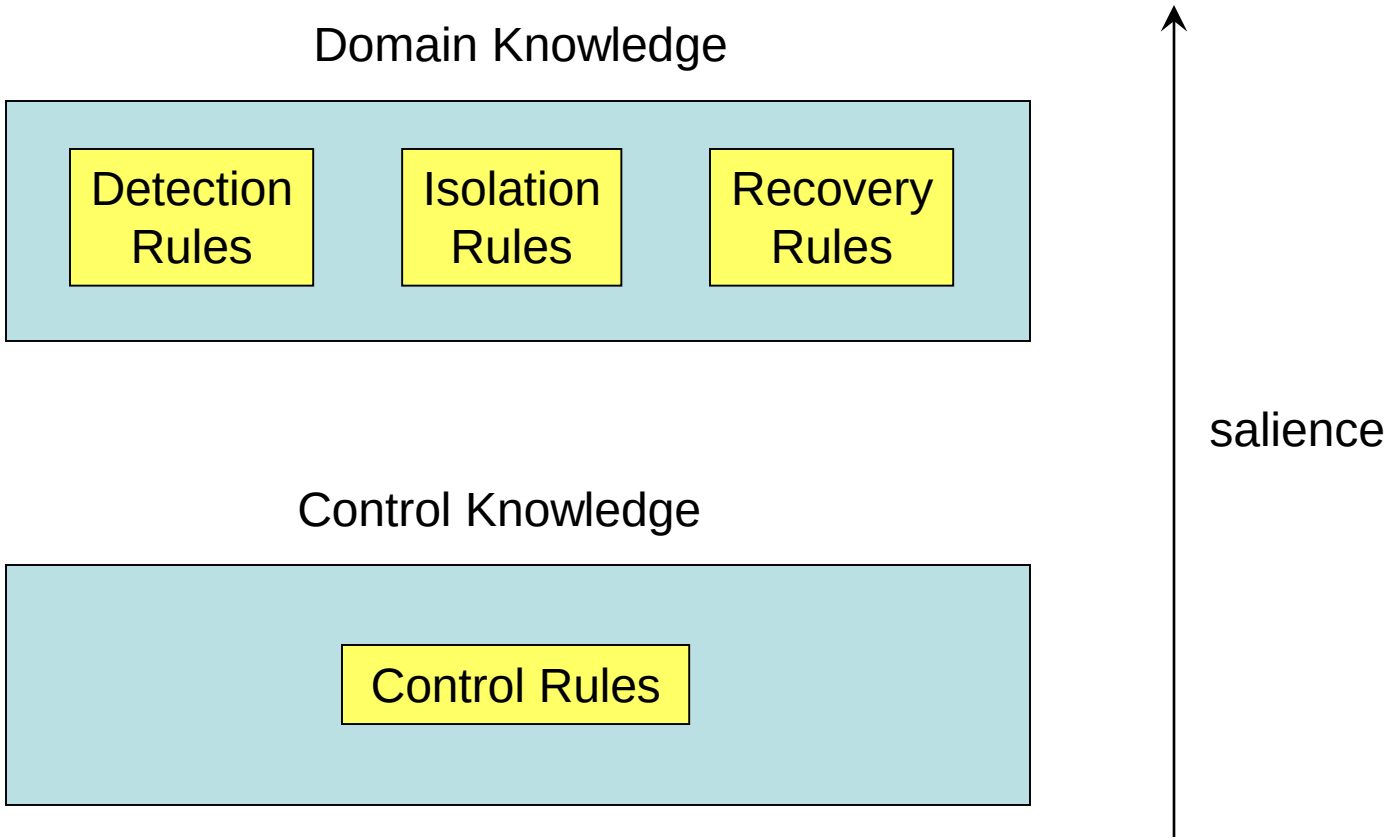
- Each rule would be given a pattern (phase xxx) indicating in which phase it would be applicable.
- Detection rules would include rules indicating when the isolation phase should be entered.
  - retract (phase detection) and assert (phase isolation)
- Drawbacks:
  - intermixing control knowledge and domain knowledge makes it difficult to develop and maintain
  - it is not always easy to determine when a phase is completed

2. Use salience to organize the rules

- Drawbacks:
  - does not guarantee the correct order of execution
    - detection rules always fire before isolation rules
    - the firing of some isolation rules might cause the activation of some detection rules



3. Separate the control knowledge from the domain knowledge



3. Separate the control knowledge from the domain knowledge (cont.)

- Each rule is given a control pattern (phase xxx) that indicates its applicable phase
- All control rules are then written to transfer control between the different phases

```
(defrule detection-to-isolation
    (declare (salience -10)
    ?phase <- (phase detection)
    =>
     (retract ?phase)
     (assert (phase isolation)))
```

```
(defrule isolation-to-recovery
    (declare (salience -10)
    ?phase <- (phase isolation)
    =>
     (retract ?phase)
     (assert (phase recovery)))
```

- the salience of domain knowledge rules is higher than control rules
  - after all activated rules in one phase are fired, the control rule (i.e. with lower salience) can then be fired (i.e. transfer to next phase)

Misuse of Salience

- Because salience is such a powerful tool, allowing explicit control over execution, it can easily be misused.
- Overuse of salience results in a poorly coded program
  - the main advantage of a rule-based program is that the programmer does not have to worry about controlling execution
- Salience should be used to determine the order when rules fire, not for selecting a single rule from a group of rules when patterns can control criteria for selection.
- Rule of thumb
  - No more than seven salience values should ever be required for coding an expert system – bested limited to 3 – 4.
  - For large expert systems, programmers should use modules to control the flow of execution – limited to 2 – 3 salience values.

The defmodule Construct

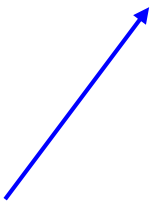
- Up to now, all *defrules*, *deftemplates*, and *deffacts* have been contained in a single work space (i.e. MAIN module)
- CLIPS uses the *defmodule* construct to partition a knowledge base by defining the various modules.
- Syntax:

```
(defmodule <module-name> [<comment>])
```

The MAIN Module

By default, CLIPS defines a MAIN module as seen below:

```
CLIPS> (clear)↵
CLIPS> (deftemplate sensor (slot name))↵
CLIPS> (ppdeftemplate sensor)↵
 (deftemplate MAIN::sensor (slot name))
CLIPS>
```



The symbol :: is called the module separator

Examples of Defining Modules

- Examples:

```
CLIPS> (defmodule DETECTION) ↵
CLIPS> (defmodule ISOLATION) ↵↵
CLIPS> (defmodule RECOVERY)  ↵↵
```

- the *current* module
  - CLIPS commands operating on a construct work only on the constructs contained in the current module.
  - When CLIPS starts or is cleared, the current module is MAIN
  - When a new modules is defined, it becomes current.
  - The function *set-current-module* is used to change the current module
    - (set-current-module ISOLATION)

Specified Module Name

- The definition of templates and rules would be place in the current module
- To override this, the module where the construct will be placed can be specified in the construct's name:

```
CLIPS> (defrule ISOLATION::example2 =>) ↵
CLIPS> (ppdefrule example2) ↵
(defrule ISOLATION::example2 =>)
CLIP>
```

Functions about Modules

- To find out which module is current:

  CLIPS> (get-current-module) ↵
- To change the current module:

  CLIPS> (set-current-module DETECTION) ↵
- Specifying modules in commands:

  CLIPS> (list-defrules RECOVERY) ↵

  CLIPS> (list-deffacts ISOLATION) ↵

  CLIPS> (list-deftemplates *) ↵
  - * indicates all of the modules

Facts in Different Modules

- Just as constructs can be partitioned by placing them in separate modules, facts can also be partitioned.
- Asserted facts are automatically associated with the module in which their corresponding deftemplates are defined.
- The *facts* command can accept a module name as an argument:

```
(facts [<module-name>])
```

Importing/Exporting Facts

- Unlike defrule and deffacts constructs, deftemplate constructs can be shared with other modules.
- A fact is “owned” by the module in which its deftemplate is contained.
- The owning module can export the deftemplate associated with the fact making that fact and all other facts using that deftemplate visible to other modules.

Importing/Exporting Facts

- It is not sufficient just to export the deftemplate to make a fact visible to another module.
- To use a deftemplate defined in another module, a module must also import the deftemplate definition.
- A construct must be defined before it can be specified in an import list, but it does not have to be defined before it can be specified in an export list; so, it is impossible for two modules to import from each other.

Exporting Constructs

- (defmodule DETECTION (export ?ALL))
  - the DETECTION module will export all exportable constructs (e.g. deftemplates and other procedural or OOP constructs)
- (defmodule DETECTION (export ?NONE))
  - the DETECTION module doesn't export any construct (default state)
- (defmodule DETECTION (export deftemplate ?ALL))
  - the DETECTION module will export all exportable deftemplates
- (defmodule DETECTION (export deftemplate <DT-name>+))
  - the DETECTION module will export a specific list of deftemplates

Importing Constructs

- (defmodule RECOVERY (import DETECTION ?ALL))
  - the RECOVERY module will import all constructs exported from the DETECTION module
- (defmodule RECOVERY (import DETECTION deftemplate ?ALL))
  - the RECOVERY module will import all deftemplates exported from the DETECTION module
- (defmodule RECOVERY (import DETECTION deftemplate <DT-name>+))
  - the RECOVERY module will import a specific list of deftemplates exported from the DETECTION module
- (defmodule RECOVERY (import DETECTION ?ALL) (import ISOLATION deftemplate ?ALL))
  - the RECOVERY module will import constructs from two modules

Modules and Execution Control

- The defmodule construct can be used to control the execution of rules.
- Each module defined in CLIPS has its own agenda.
- Execution can be controlled by deciding which module's agenda is selected for executing rules.

The Agenda Command

- To display activations for the current module:

  CLIPS> `(agenda)` ↵

- To display activations for the DETECTION module:

  CLIPS> (agenda DETECTION) ↵

The Focus Command

- Assume there are rules on several agendas – when the *run* command is issued, only rules in the current focus module are fired.
- The *reset* and *clear* commands automatically set the current focus to the MAIN module.
- The current focus does not change when the current module is changed.
  - (set-current-module DETECTION) ↵ doesn't change the current focus
- As rules execute, the current focus becomes empty, is popped from the focus stack, the next module becomes the current focus until empty.

The Focus Command

- To change the current focus:

```
CLIPS> (focus <module-name>+)↵
```

- Example:

```
CLIPS> (focus DETECTION) ↵
TRUE
CLIPS> (run)↵
```

- Now rules on the DETECTION module will be fired.

Manipulating the Focus Stack

CLIPS provides several commands for manipulating the current focus and stack:

1. (clear-focus-stack) – removes all modules from focus stack
2. (get-focus) – returns module name of current focus or FALSE if empty
3. (pop-focus) – removes current focus from stack or FALSE if empty

Manipulating the Focus Stack

4. (get-focus-stack) – returns a multifield value containing the modules on the focus stack
5. (watch focus) – can be used to see changes in the focus stack
6. (return) – terminate execution of a rule's RHS, remove current focus from focus stack, return execution to next module
7. (halt) – terminate the execution of all modules

Implementing the Flow of Control

4. Use modules to organize the rules

```
(defmodule DETECTION)
(defmodule ISOLATION)
(defmodule RECOVERY)

(defrule MAIN::change-phase
   =>
   (focus DETECTION ISOLATION RECOVERY))
```

OR

```
(defrule MAIN::change-phase
   =>
   (focus RECOVERY)
   (focus ISOLATION)
   (focus DETECTION))
```

CLIPS Commands - Math

1. (max <numeric>+)
   - returns the value of its largest argument
   - (max 10 5 15 12) ➔ 15
2. (min <numeric>+)
   - returns the value of its smallest argument
   - (min 10 5 15 12) ➔ 5
3. (** <numeric-1> <numeric-2>)
   - returns the value of <numeric-1> raised to the power of <numeric-2>
   - (** 2 3) ➔ 8
4. (round <numeric>)
   - returns the value of argument rounded to the closest integer
   - (round 6.6) ➔ 7
5. (integer <numeric>)
   - (integer 6.6) ➔ 6

CLIPS Commands – String

1. (assert-string <string>)
   - Converts a string to a fact and asserts it
2. (str-cat <argument>*)
   - Returns all of its arguments concatenated as a string
3. (str-length <string>)
   - Returns the length of a string in characters
4. (str-compare <string1> <string2>)
   - Returns zero if string1 is equal to string 2
   - Returns a positive integer if string1 is lexicographically greater than string 2
   - Returns a negative integer if string1 is lexicographically less than string 2

CLIPS Commands - Multifield

1. (create$ <expression>+)
   - create a multifield value
   - (create$ a b c d) ➔ (a b c d)
2. (explode$ <string>)
   - returns a multifield value from a string
   - (explode$ “a b c d”) ➔ (a b c d)
3. (implode$ <multifield>)
   - returns a string containing the fields from a multifield value
   - (implode$ (create$ a b c d)) ➔ “a b c d”

CLIPS Commands – Multifield (cont.)

4. (first$ <multifield>)
   - returns the first field of <multifield>
   - (first$ (create$ a b c d)) ➔ a
5. (rest$ <multifield>)
   - returns a multifield value containing all but the first field
   - (rest$ (create$ a b c d)) ➔ (b c d)
6. (nth$ <integer> <multifield>)
   - returns the ***n***th field containing in <multifield>
   - (nth$ 3 (create$ a b c d)) ➔ c

CLIPS Commands – Multifield (cont.)

7. (member$ <single-field> <multifield>)
   - examines if the <single-field> is in the <multifield> or not
   - returns the position if in the <multifield> or FALSE if not
   - (member$ b (create$ a b c d)) ➔ 2
8. (length$ <multifield>)
   - returns the number of fields in a multifield value
   - (length$ (create$ a b c d)) ➔ 4
9. (subseq$ <multifield> <begin> <end>)
   - extracts the fields in the specified range and returns them in a multifield value
   - (subseq$ (create$ a b c d e) 2 4) ➔ (b c d)
10. (subsetp <multifield-1> <multifield-2>)
    - returns TRUE in <multifield-1> is a subset of <multifield-2>, otherwise FALSE
    - (subsetp (create$ b c d) (create$ a b c d e)) ➔ TRUE

CLIPS Commands – Multifield (cont.)

11. (insert$ <multifield> <integer> <single-or-multifield>)
    - inserts the <single-or-multifield> vaule into the <multifield>, before the ***n***th value
    - (insert$ (create$ a b c d) 3 h) ➔ (a b h c d)
12. (delete$ <multifield> <begin> <end>)
    - deletes the fields in the specified range and returns the result
    - (delete$ (create$ a b c d e) 2 4) ➔ (a e)
13. (delete-member$ <multifield> <single-or-multifield>+)
    - deletes specified values contained within a multifield value
    - (delete-member$ (create$ a b c d e f g) (create$ b c) f e) ➔ (a d g)

CLIPS Commands – Multifield (cont.)

14. (replace$ <multifield> <begin> <end> <single-or-multifield>)
    - replaces the fields in the specified range with the <sigle-or-multifield> value and returns the result
    - (replace$ (create$ a b c d e) 2 4 (create$ f g)) ➔ (a f g e)
15. (replace-member$ <multifield> <substitute> <search>+)
    - replaces specified values contained within a multifield value
    - (replace-member$ (create$ a b c d e f g h) k g (create$ b c) e) ➔ (a k d k f k h)

CLIPS Commands – Predicate

1. (eq <expression> <expression>+)
   - returns TRUE if its first argument is equal in type and value to all its subsequent arguments, otherwise FALSE
   - (eq (create$ a b c) (explode$ “a b c”)) ➔ TRUE
   - (eq 5 5.0) ➔ FALSE
2. (neq <expression> <expression>+)
   - returns TRUE if its first argument is not equal in type and value to all its subsequent arguments, otherwise FALSE
   - (neq (implode$ (create$ a b c)) “a b c”) ➔ FALSE
3. logical commands
   - (and <expression>+) , (or <expression>+) , (not <expression>)
   - (= <numeric-1> <numeric-2>) , (<> <numeric-1> <numeric-2>) , (> <numeric-1> <numeric-2>) , (< <numeric-1> <numeric-2>) , (>= <numeric-1> <numeric-2>) , (<= <numeric-1> <numeric-2>)

CLIPS Commands – Predicate (cont.)

4. (multifieldp <expression>)
   - returns TRUE if <expression> is a multifield value, otherwise FALSE
5. data type testing
   - (numberp <expression>) : a float or an integer
   - (integerp <expression>)
   - (floatp <expression>)
   - (lexemep <expression>) : a string or a symbol
   - (stringp <expression>)
   - (symbolp <expression>)

Chapter A. FuzzyCLIPS

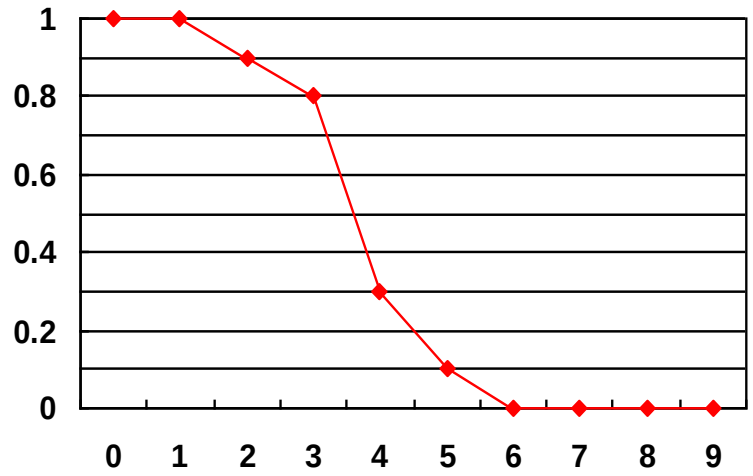
- FuzzyCLIPS makes modifications to CLIPS containing the capability of handling fuzzy concept and reasoning
  - can express rules using fuzzy terms
  - fuzzy sets and fuzzy relations deal with vagueness
  - certainty factors for rules and facts manipulate uncertainty
  - existing CLIPS programs still execute correctly

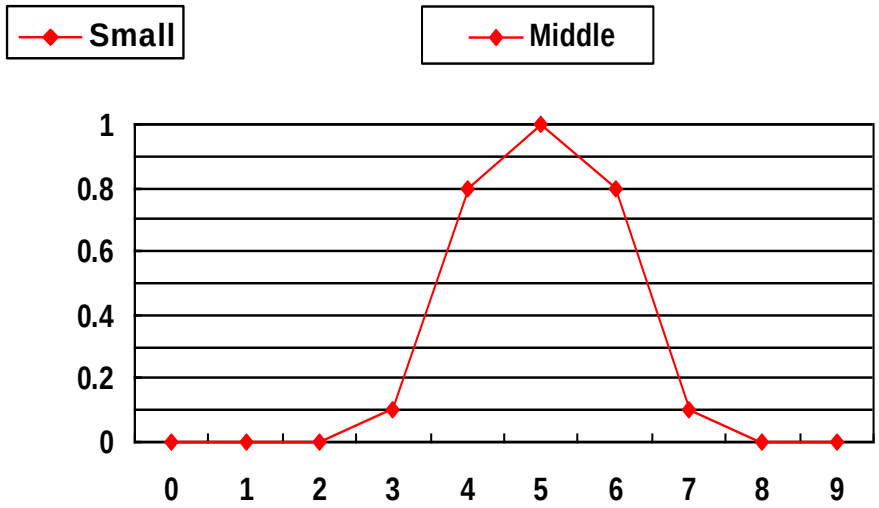
Define fuzzy variables using deftemplate

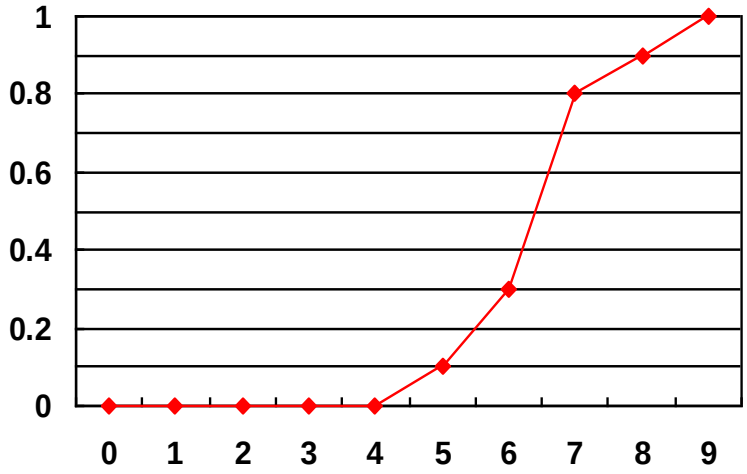
- (deftemplate <fuzzy-variable-name>
  <from> <to> [<unit>]
  ((<value-1> <fuzzy-set-1>) (<value-2> <fuzzy-set-2>) …)
)
- <fuzzy-set> ::= <singletons> | <standard> | <linguistic-expr>
- (deftemplate digit
  0 9
  (
  (Small (1 1) (2 0.9) (3 0.8) (4 0.3) (5 0.1) (6 0))
  (Middle (2 0) (3 0.1) (4 0.8) (5 1) (6 0.8) (7 0.1) (8 0))
  (Large (4 0) (5 0.1) (6 0.3) (7 0.8) (8 0.9) (9 1))
)
)

Singletons must be from small to large
(Small (6 0) (5 0.1) (1 1)) (X)

Fuzzy sets of Small, Middle, and Large







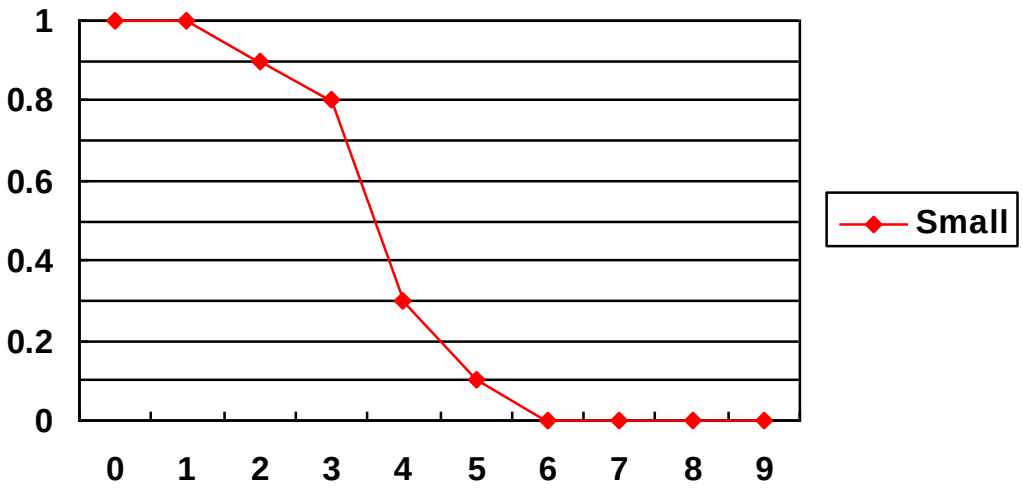
Small

Large

(Small (1 1) (2 0.9) (3 0.8) (4 0.3) (5 0.1) (6 0))
(Middle (2 0) (3 0.1) (4 0.8) (5 1) (6 0.8) (7 0.1) (8 0))
(Large (4 0) (5 0.1) (6 0.3) (7 0.8) (8 0.9) (9 1))

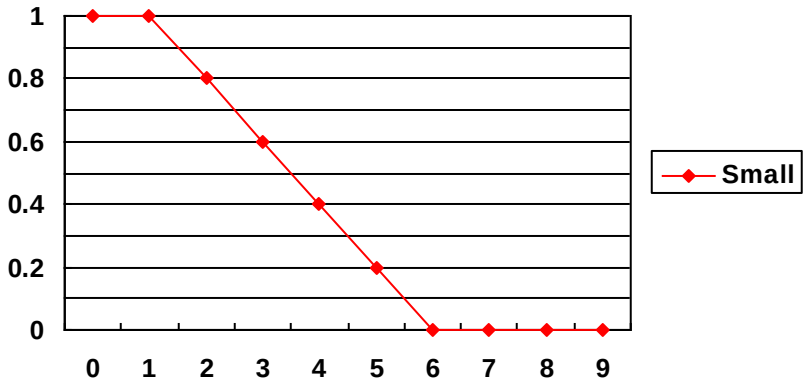
Singleton representation

- <singletons> ::= $(x_1\ \mu_1)\ (x_2\ \mu_2)\ \ldots\ (x_n\ \mu_n)$
- $(x,\mu_A(x))$ is called a singleton
  - the grade (degree) of membership μ_A of x in fuzzy set A
  - (Small (1 1) (2 0.9) (3 0.8) (4 0.3) (5 0.1) (6 0))
    - limits (range) is 0 9, $\forall$ x<1, $\mu_{Small}(x)=1$, $\forall$ x>6, $\mu_{Small}(x)=0$

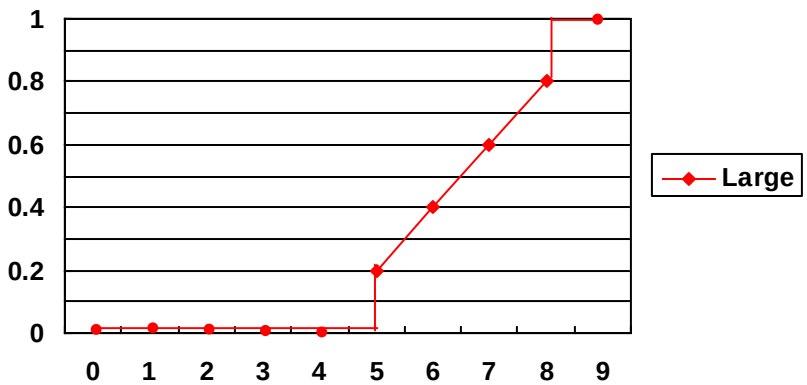


More examples for the singleton representation

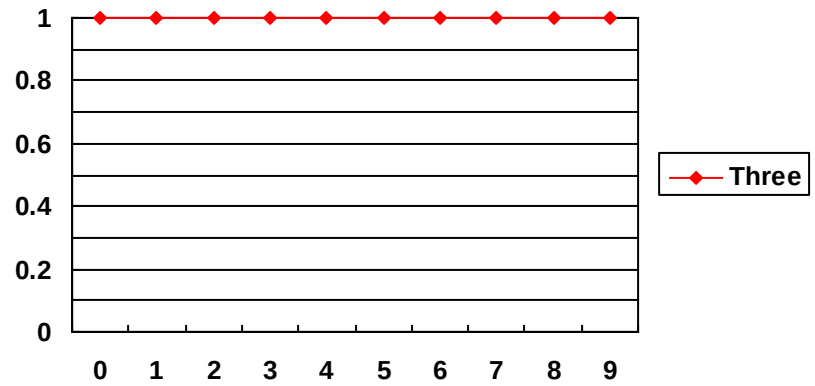
- (Small (1 1) (6 0))



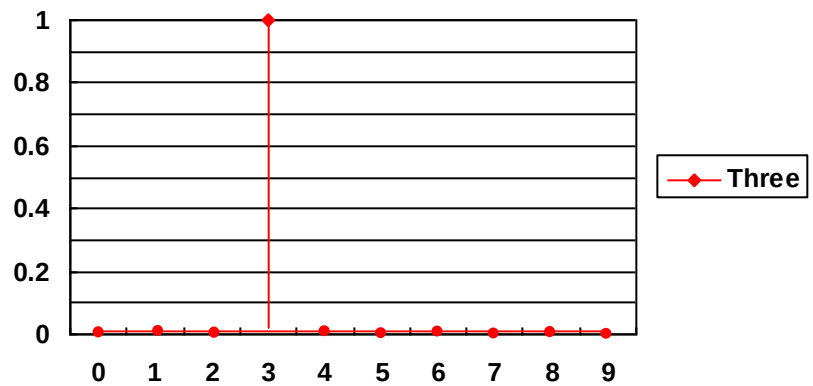
- (Large (5 0) (5 0.2) (8 0.8) (8 1))



- (Three (3 1))

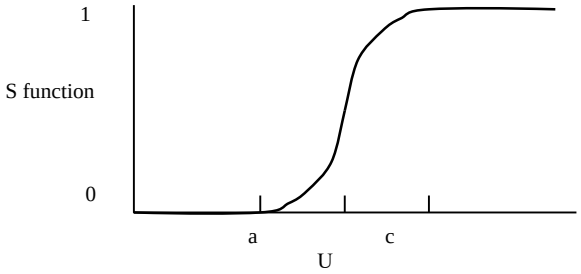


- (Three (3 0) (3 1) (3 0))



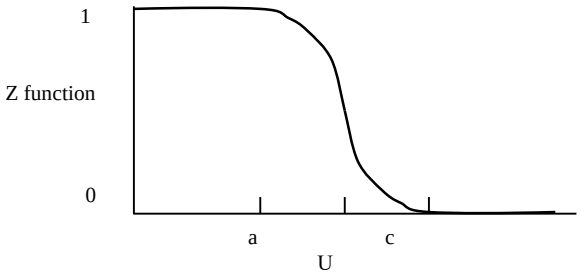
Standard Function representation

- <standard>::= (S a c) | (s a c) | (Z a c) | (z a c) | (PI d b) | (pi d b)

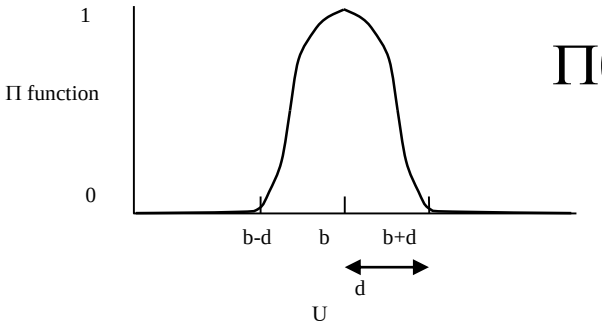


$$S(u,a,c)=0,\quad u\le a,\ u\in U \qquad S(u,a,c)=2\left(\frac{u-a}{c-a}\right)^2,\quad a<u\le\frac{a+c}{2}$$

$$S(u,a,c)=1,\quad c<u \qquad S(u,a,c)=1-2\left(\frac{c-u}{c-a}\right)^2,\quad \frac{a+c}{2}<u\le c$$



$$Z(u,a,c)=1-S(u,a,c)$$

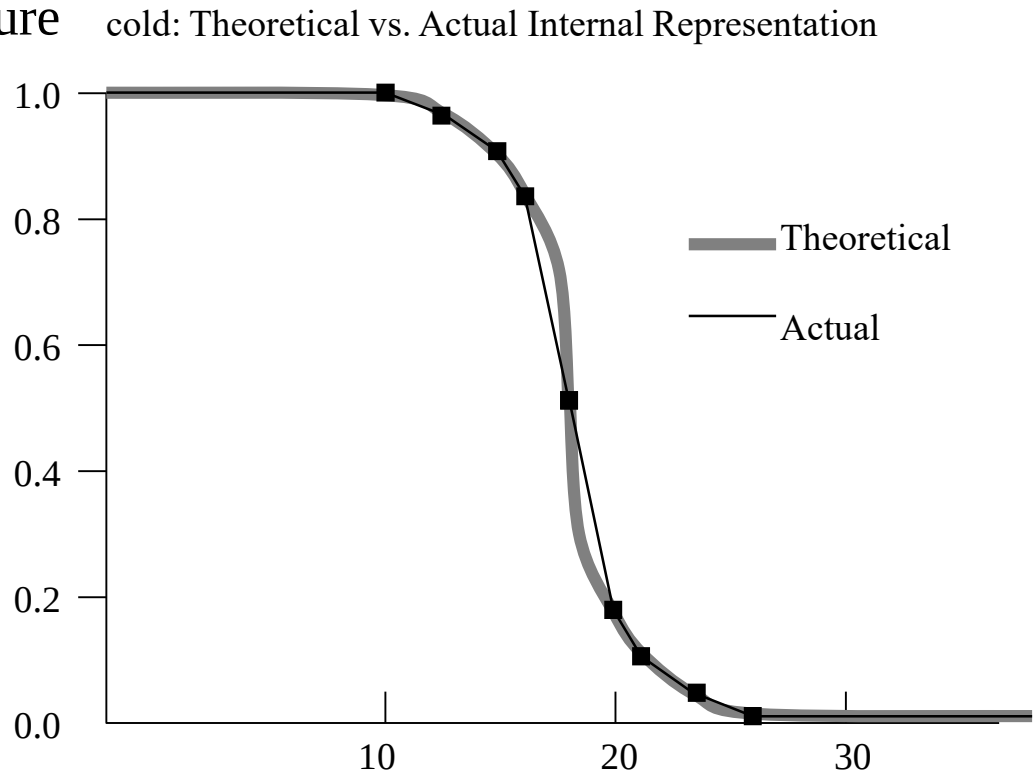


$$\Pi(u,d,b)=S(u,b-d,b),\quad u\le b$$

$$\Pi(u,d,b)=Z(u,b,b+d),\quad b<u$$

Standard Function example

- (deftemplate water-temperature
  0 100 Celsius
  (
  (cold (z 10 26))
  (warm (pi 7 32))
  (hot (s 36 50))
)
)
- FuzzyCLIPS converts all standard notation to singleton representation.
- Nine points, equally spaced along the x axis, are selected to represent the functions



The number of points (9) can be changed by modifiying the value of ArraySIZE

Linguistic Expression representation

- (deftemplate water-temperature
  0 100 Celsius
  (
    (cold (z 10 26))
    (hot (s 37 60))
    (warm not [hot or cold])
)
)
- the term *warm* is described as being *not hot or cold*. It uses the terms *hot* and *cold* previously defined in this deftemplate
- Only terms described in this deftemplate (before the term definition being defined) can be used (along with any available modifiers and the *and*, *or*, *not* operations).
- $\mu_{warm}(x) = 1 - \max(\mu_{hot}(x) , \mu_{cold}(x))$

Modifiers

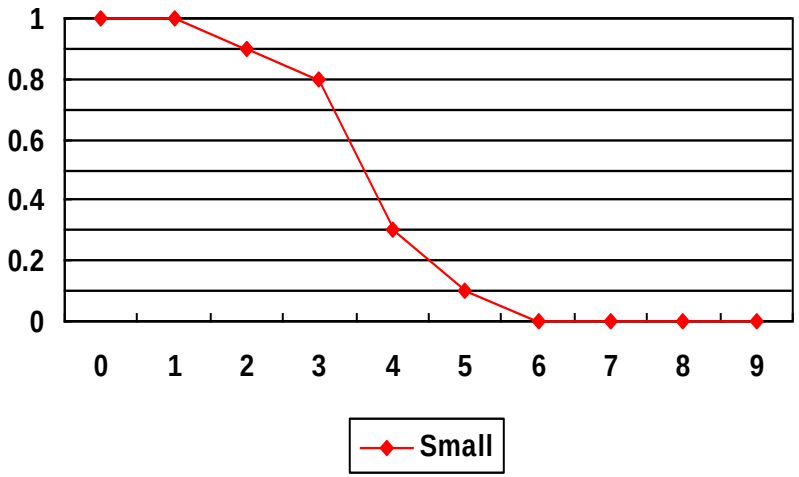
- Predefined Modifiers

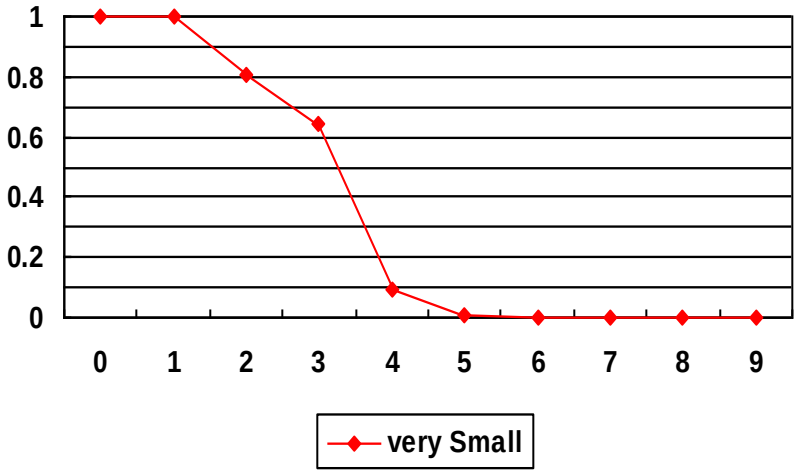
Modifier Name	Modifier Description
not	1-y
very	y**2
somewhat	y**0.333
more-or-less	y**0.5
extremely	y**3

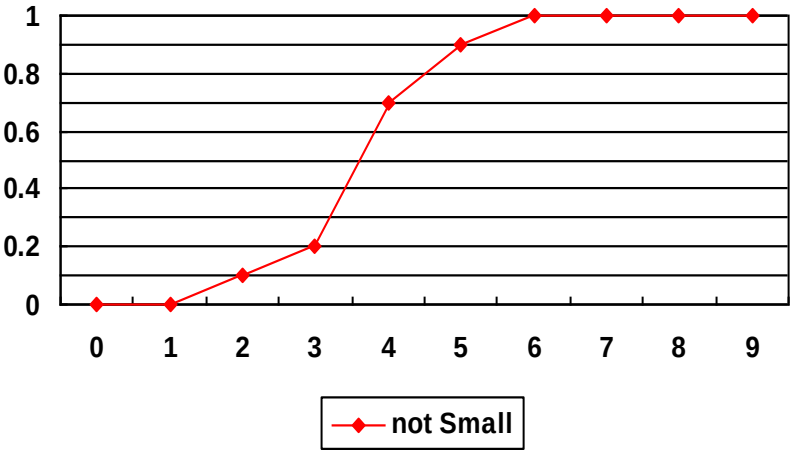
- User Defined Modifiers
  - (add-fuzzy-modifier modname modfunction)

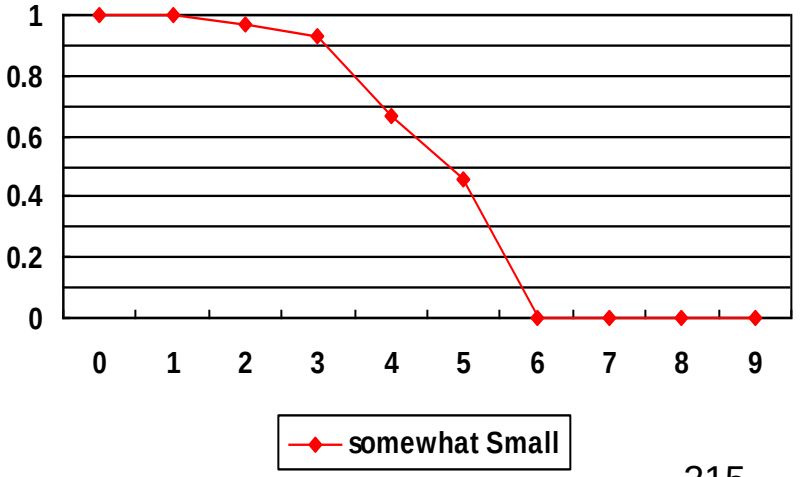
```
(add-fuzzy-modifier my-somewhat sqrt)
(deffunction most-extremely-fcn (?x)
   (** ?x 5)
)
(add-fuzzy-modifier most-extremely most-extremely-fcn)
```

Small, not Small, very Small, somewhat Small









Define fuzzy slots in deftemplate

- <fuzzy-slot> ::= (slot <slot-name> (type FUZZY-VALUE <fuzzy-variable>))
- (deftemplate water-temperature
  0 100 Celsius
  (
    (cold (z 10 26))
    (warm (PI 5 32))
    (hot (s 37 60))
)
)
- (deftemplate swimming-pool
  (slot name (type SYMBOL))
  (slot size)
  (slot temperature (type FUZZY-VALUE water-temperature))
)

```
(defrule warm-pool
   (swimming-pool (name ?n) (size ?s)
                  (temperature warm))
    =>
   (printout t ?n " is a warm pool." crlf))
```

Fuzzy patterns in LHS of defrule

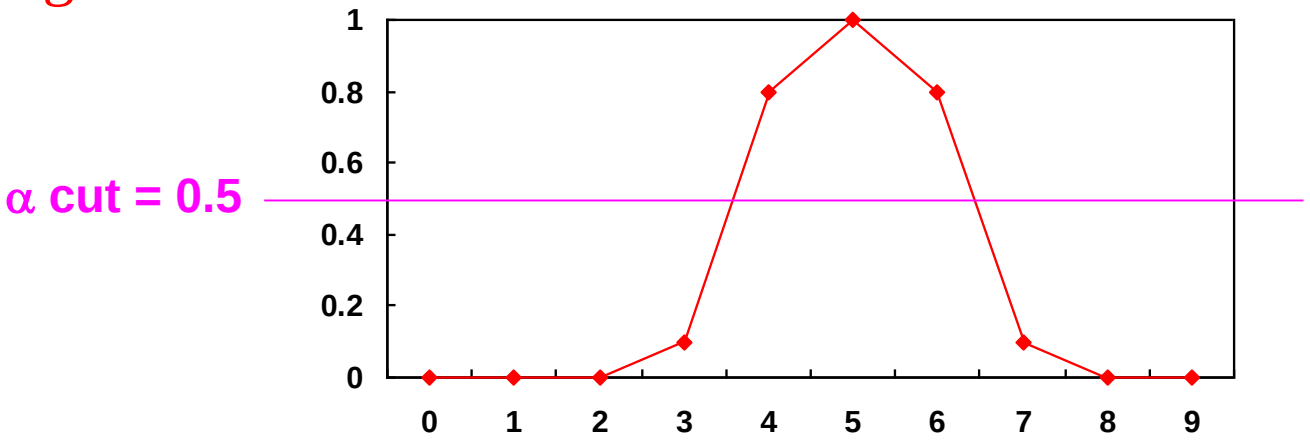
- Fuzzy variables & fuzzy value
  - (water-temperature warm)
  - (water-temperature ?t)
  - (water-temperature ?t & cold)
- Fuzzy slots
  - (swimming-pool (name ?n) (size ?s) (temperature warm))
  - (swimming-pool (name ?n) (size ?s) (temperature ?t))
  - (swimming-pool (name ?n) (size ?s) (temperature ?t & cold))

Facts with fuzzy variables

- (deffacts initial
  (water-temperature warm)
  (water-temperature (30 0) (50 1))
  (water-temperature (40 0) (40 1) (40 0))
  (water-temperature (s 30 50))
  (swimming-pool (name AA) (size 1000) (temperature (z 30 50)))
  (water-temperature 40) (X) not fuzzy value
  (swimming-pool (name BB) (size 500) (temperature 40)) (X)
)
- (assert (swimming-pool (name CC) (size 800) (temperature very cold)))
  (assert (swimming-pool (name DD) (size 700) (temperature (30 0) (50 1))))
  (assert (swimming-pool (name EE) (size 600) (temperature (s 30 50))))
  (assert (water-temperature (z 30 50)))
  (assert (water-temperature (pi 10 40)))

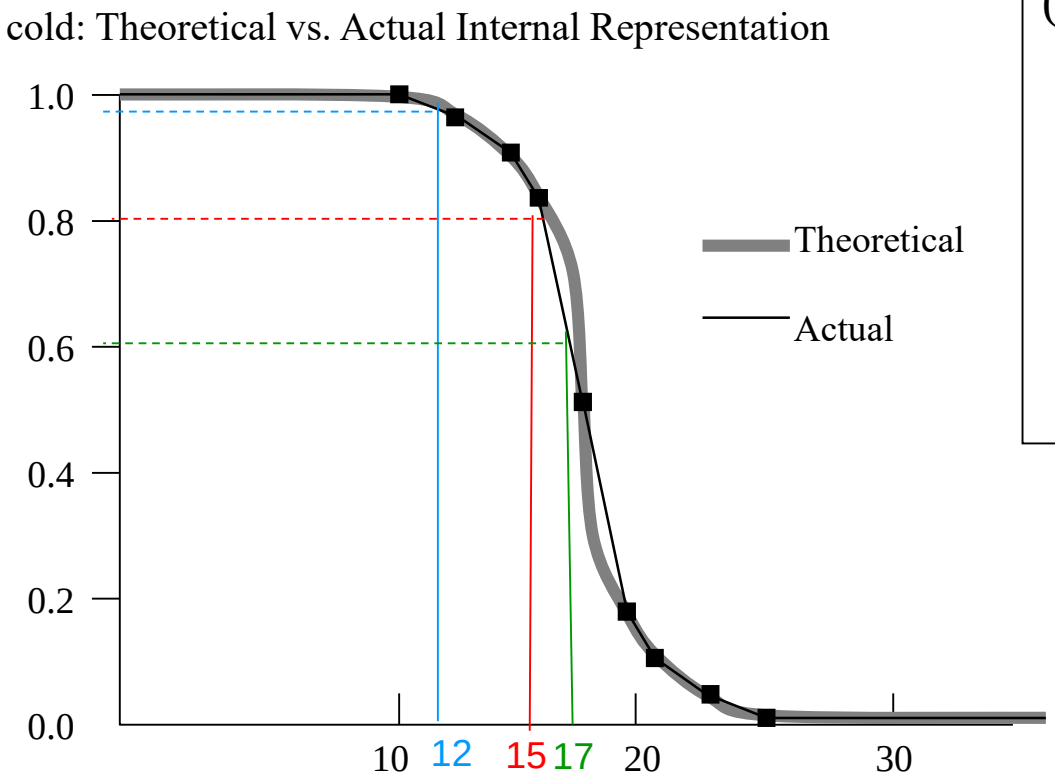
α cut (alpha cut)

- The set of elements that belong to a fuzzy set A at least to the degree α



- (set-alpha-value α) ; $0 \leq \alpha \leq 1$
  - (set-alpha-value 0.4): When fuzzy slots are matched to fuzzy patterns on LHS, the match is successful if the maximum of the intersection between the two fuzzy sets is greater than 0.4
  - the default alpha-value is 0.0
  - (clear) does not reset the alpha-value to 0.0

Example: cold




```
(deftemplate water-temperature
        0 100 Celsius
        (
          (cold (z 10 26))
          (warm (PI 5 32))
          (hot (s 37 60))
        )
    )
```

	cold	very cold
10	1	1
12	0.97	0.94
15	0.80	0.64
17	0.61	0.37

Activation for fuzzy rules

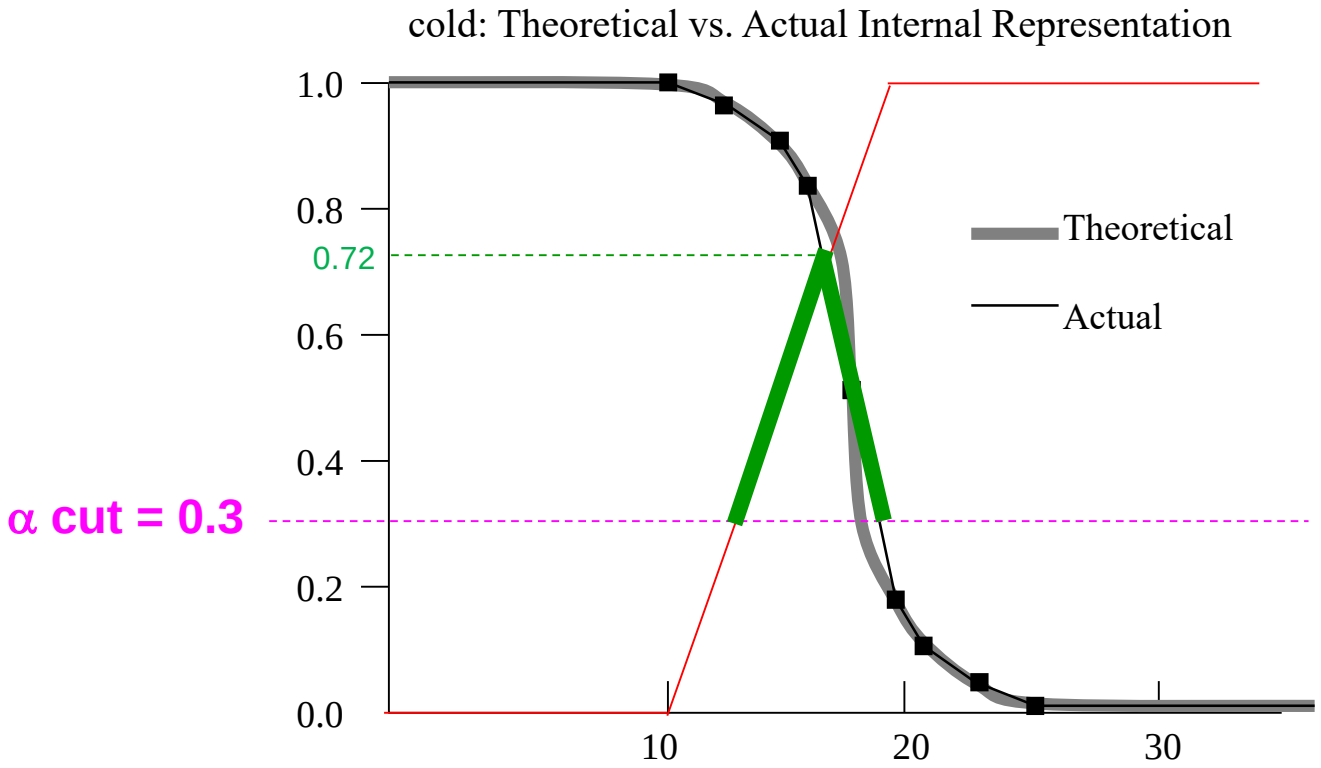
- (assert (swimming-pool (name aaa) (size 100) (temperature (10 0) (10 1) (10 0))))
  (assert (swimming-pool (name bbb) (size 150) (temperature (12 0) (12 1) (12 0))))
  (assert (swimming-pool (name ccc) (size 100) (temperature (15 0) (15 1) (15 0))))
  (assert (swimming-pool (name ddd) (size 200) (temperature (17 0) (17 1) (17 0))))
- (defrule cold-pool
  (swimming-pool (name ?n) (size ?s) (temperature cold))
  =>
  (printout t ?n " is a cold pool." crlf))
- (defrule very-cold-pool
  (swimming-pool (name ?n) (size ?s) (temperature very cold))
  =>
  (printout t ?n " is a very cold pool." crlf))

	cold	very cold
10	1	1
12	0.97	0.94
15	0.80	0.64
17	0.61	0.37

- (set-alpha-value 0.4): cold: aaa bbb ccc ddd very cold: aaa bbb ccc
- (set-alpha-value 0.7): cold: aaa bbb ccc very cold: aaa bbb
- (set-alpha-value 0.95): cold: aaa bbb very cold: aaa
- (set-alpha-value 1): cold: aaa very cold: aaa

Example: Intersection

$\mu_{intersection}(x) = \min(\mu_1(x) , \mu_2(x))$



(cold (z 10 26))
(swimming-pool (name fff) (size 200) (temperature (10 0) (20 1)))

Intersection between two fuzzy sets

- (assert (swimming-pool (name fff) (size 200) (temperature (10 0) (20 1))))
- (defrule cold-pool
  (swimming-pool (name ?n) (size ?s) (temperature cold))
  =>
  (printout t ?n " is a cold pool." crlf))
- (defrule very-cold-pool
  (swimming-pool (name ?n) (size ?s) (temperature very cold))
  =>
  (printout t ?n " is a very cold pool." crlf))
- Some point must be greater than the alpha value
- (set-alpha-value 0.5): cold: fff very cold: fff
- (set-alpha-value 0.6): cold: fff very cold:
- (set-alpha-value 0.8): cold: very cold:

$0.72^2 = 0.5184$

Combinations of patterns in LHS

- (assert (swimming-pool (name aaa) (size 100) (temperature (10 0) (10 1) (10 0))))
  (assert (swimming-pool (name bbb) (size 150) (temperature (12 0) (12 1) (12 0))))
  (assert (swimming-pool (name ccc) (size 100) (temperature (15 0) (15 1) (15 0))))
  (assert (swimming-pool (name ddd) (size 200) (temperature (17 0) (17 1) (17 0))))
- (defrule one-cold-and-one-very-cold-pool
  (swimming-pool (name ?n1) (size ?s1) (temperature cold))
  (swimming-pool (name ?n2&~?n1) (size ?s2) (temperature very cold))
  =>
  (printout t ?n1 " is a cold pool and " ?n2 " is a very cold pool." crlf))
- All patterns must be greater than the alpha value
- (set-alpha-value 0.7): bbb is a cold pool and aaa is a very cold pool.
  ccc is a cold pool and aaa is a very cold pool.
  aaa is a cold pool and bbb is a very cold pool.
  ccc is a cold pool and bbb is a very cold pool.
- (set-alpha-value 0.95): bbb is a cold pool and aaa is a very cold pool.
- (set-alpha-value 1):

	cold	very cold
10	1	1
12	0.97	0.94
15	0.80	0.64
17	0.61	0.37

Certainty Factor (CF)

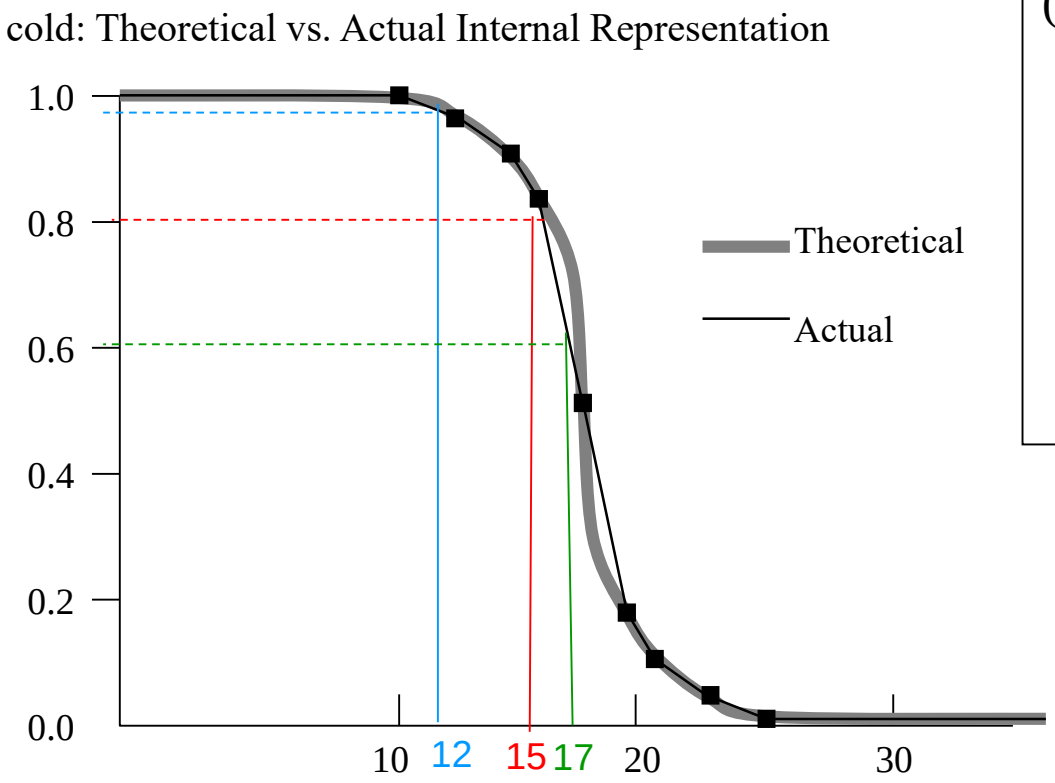
- degree of membership: vagueness
- certainty factors (confidence level): uncertainty
- Certainty factor is a numeric value from 0.0 to 1.0
  - default is 1.0
- Certainty factors of facts
  - (assert (water-temperature (z 30 50)) CF 0.6)
  - (assert (swimming-pool (name aaa) (size 90) (temperature cold)) CF 0.8)
  - (deffacts initial
    (water-temperature (25 0) (35 0.6) (45 1)) CF 0.3
    (digit very small) CF 0.7
)

CF of inferred facts by rules

- (assert (swimming-pool (name aaa) (size 100) (temperature (10 0) (10 1) (10 0))))
  (assert (swimming-pool (name bbb) (size 150) (temperature (12 0) (12 1) (12 0))))
  (assert (swimming-pool (name ccc) (size 100) (temperature (15 0) (15 1) (15 0))))
  (assert (swimming-pool (name ddd) (size 200) (temperature (17 0) (17 1) (17 0))))
- (defrule one-cold-and-one-very-cold-pool
  (swimming-pool (name ?n1) (size ?s1) (temperature cold))
  (swimming-pool (name ?n2&~?n1) (size ?s2) (temperature very cold))
  =>
  (assert (match ?n1?n2)))
- The CF of inferred facts are the minimum of all matched patterns
- (set-alpha-value 0.7): (match bbb aaa) CF 0.97
  (match ccc aaa) CF 0.80
  (match aaa bbb) CF 0.94
  (match ccc bbb) CF 0.80
- (set-alpha-value 0.95): (match bbb aaa) CF 0.97
- (set-alpha-value 1):

	cold	very cold
10	1	1
12	0.97	0.94
15	0.80	0.64
17	0.61	0.37

Example: cold




```
(deftemplate water-temperature
        0 100 Celsius
        (
          (cold (z 10 26))
          (warm (PI 5 32))
          (hot (s 37 60))
        )
    )
```

	cold	very cold
10	1	1
12	0.97	0.94
15	0.80	0.64
17	0.61	0.37

Certainty Factors of Rules

- (defrule <rule-name>
  (declare (CF <value>))
  <patterns>* ------LHS
  =>
  <actions>*) ------RHS
- (set-threshold **cf**) ; $0 \leq$ **cf** ≤ 1
  - (set-threshold 0.4): When facts are matched to fuzzy patterns on LHS, the match is successful if (CF of the fact)*(CF of the rule) is greater than 0.4
  - the default threshold is 0.0
  - (clear) does not reset the threshold to 0.0

CF for activations of rules

- (swimming-pool (name aaa) (size 100) (temperature (10 0) (10 1) (10 0))) CF 0.5
  (swimming-pool (name bbb) (size 150) (temperature (12 0) (12 1) (12 0))) CF 0.6
  (swimming-pool (name ccc) (size 100) (temperature (15 0) (15 1) (15 0))) CF 0.7
  (swimming-pool (name ddd) (size 200) (temperature (17 0) (17 1) (17 0))) CF 0.8
- (defrule one-cold-and-one-very-cold-pool
  (declare (CF 0.8))
  (swimming-pool (name ?n1) (size ?s1) (temperature cold))
  (swimming-pool (name ?n2&~?n1) (size ?s2) (temperature very cold))
  =>
  (printout t ?n1 " is a cold pool and " ?n2 " is a very cold pool." crlf))
- (CF of the fact)*(CF of the rule) must be greater than threshold value
- (set-threshold 0.5): ccc is a cold pool and ddd is a very cold pool.
  ddd is a cold pool and ccc is a very cold pool.
- (set-threshold 0.6):

aaa 0.5*0.8
bbb 0.6*0.8
ccc 0.7*0.8
ddd 0.8*0.8

CF of inferred facts by rules

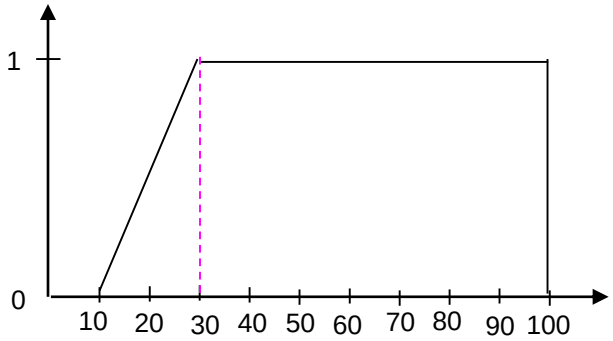
- (swimming-pool (name aaa) (size 100) (temperature (10 0) (10 1) (10 0))) CF 0.5
  (swimming-pool (name bbb) (size 150) (temperature (12 0) (12 1) (12 0))) CF 0.6
  (swimming-pool (name ccc) (size 100) (temperature (15 0) (15 1) (15 0))) CF 0.7
  (swimming-pool (name ddd) (size 200) (temperature (17 0) (17 1) (17 0))) CF 0.8
- (defrule one-cold-and-one-very-cold-pool
  (declare (CF 0.8))
  (swimming-pool (name ?n1) (size ?s1) (temperature cold))
  (swimming-pool (name ?n2&~?n1) (size ?s2) (temperature very cold))
  =>
  (assert (match ?n1?n2)))

	cold	very cold
10	1	1
12	0.97	0.94
15	0.80	0.64
17	0.61	0.37

- The CF of inferred facts are the minimum of (CF of the fact) *(CF of the rule)* (degree of matching)
- (set-threshold 0.5): (match ccc ddd) CF 0.21 <min(0.7*0.8*0.8, 0.8*0.8*0.37>
  (match ddd ccc) CF 0.36 <min(0.8*0.8*0.61, 0.7*0.8*0.64>

Defuzzify

- A crisp value can be extracted from a fuzzy set
  - moment-defuzzify: the centre of gravity (mean of all non-zero values)
  - maximum-defuzzify: mean of maxima



- (water-temperature (10 0) (30 1))
- (defrule defuzzification

  (water-temperature ?fv)

  =>

  (printout t "Temperature 1 is " (moment-defuzzify ?fv) " " (get-u-units ?fv) crlf)

  (printout t "Temperature 2 is " (maximum-defuzzify ?fv) crlf))
- Temperature 1 is 60.0 Celsius

  Temperature 2 is 65.0
- (get-u-units <fuzzy-value>) returns the units of the universe of discourse

```
(deftemplate water-temperature
      0 100 Celsius
      (
        (cold (z 10 26))
        (warm (PI 5 32))
        (hot (s 37 60))
      )
)
```

Obtain the certainty factor value

- (get-cf <fact-index>)
  - Return the certainty factor of a fact
- (match aaa ddd) CF 0.21
  (match ddd ccc) CF 0.36
- (defrule example-print-cf
  ?f1 <- (match ?n1 ?n2)
  =>
  (printout t "CF of (match " ?n1 " " ?n2 ") is " (get-cf ?f1)))
- CF of (match aaa ddd) is 0.21
  CF of (match ddd ccc) is 0.36

Fuzzy value function

- (fuzzyvaluep ?var)
  - The testing predicate function returns TRUE if ?var is of type FUZZY-VALUE, otherwise FALSE
- (get-fs <fuzzy-value>)
  - Returns the entire fuzzy set

Fuzzy union and fuzzy intersection

- (fuzzy-union <fuzzy-value> <fuzzy-value>)
  - Returns a new fuzzy value that is the union of two other fuzzy values
- (fuzzy-intersection <fuzzy-value> <fuzzy-value>)
  - Returns a new fuzzy value that is the intersection of two other fuzzy values

Example of fuzzy value function

- (swimming-pool (name ppp) (size 100) (temperature (10 0) (30 1)))
  (swimming-pool (name qqq) (size 150) (temperature (20 1) (40 0)))
- (defrule compute-fuzzy-value
  (swimming-pool (name ?n1) (size ?) (temperature ?t1))
  (swimming-pool (name ?n2&~?n1) (size ?) (temperature ?t2))
  =>
  (printout t "Fuzzy-Union is " (get-fs (fuzzy-union ?t1 ?t2)) crlf)
  (printout t "Fuzzy-intersection is " (get-fs (fuzzy-intersection ?t1 ?t2)) crlf))
- Fuzzy-Union is (20 1.0) (25 0.75) (30 1.0)
  Fuzzy-intersection is (10 0) (25 0.75) (40 0)