CPE/CSC 481: Knowledge-Based Systems

Franz J. Kurfess

Computer Science Department
California Polytechnic State University
San Luis Obispo, CA, U.S.A.

Rule-Based Reasoning

Motivation

- CLIPS is a decent example of an expert system shell
  - rule-based, forward-chaining system
- it illustrates many of the concepts and methods used in other ES shells
- it allows the representation of knowledge, and its use for solving suitable problems

Objectives

- be familiar with the important concepts and methods used in rule-based ES shells
  - facts, rules, pattern matching, agenda, working memory, forward chaining
- understand the fundamental workings of an ES shell
  - knowledge representation
  - reasoning
- apply rule-based techniques to simple examples
- evaluate the suitability of rule-based systems for specific tasks dealing with knowledge

Shallow and Deep Reasoning

- shallow reasoning
  - also called experiential reasoning
  - aims at describing aspects of the world heuristically
  - short inference chains
  - possibly complex rules
- deep reasoning
  - also called causal reasoning
  - aims at building a model of the world that behaves like the "real thing"
  - long inference chains
  - often simple rules that describe cause and effect relationships

Forward Chaining

- given a set of basic facts, we try to derive a conclusion from these facts
- example: What can we conjecture about Clyde?

```
IF elephant(x) THEN mammal(x)

IF mammal(x) THEN animal(x)

elephant (Clyde)
```

modus ponens:

```
IF p THEN q
p
```
$$\frac{\text{IF p THEN q} \quad \text{p}}{\text{q}}$$

unification:

find compatible values for variables

Forward Chaining Example

```
IF elephant(x) THEN mammal(x)
IF mammal(x) THEN animal(x)
elephant(Clyde)
```

unification:

find compatible values for variables

modus ponens:

```
IF p THEN q
p
q
```

```
IF elephant(   x   ) THEN mammal(   x   )
```

```
elephant (Clyde)
```

Forward Chaining Example

```
IF elephant(x) THEN mammal(x)

IF mammal(x) THEN animal(x)

elephant(Clyde)
```

unification:

find compatible values for variables

modus ponens:

```
IF p THEN q
p
_____________
q
```

```
IF elephant(Clyde) THEN mammal(Clyde)
```

```
elephant (Clyde)
```

Forward Chaining Example

```
IF elephant(x) THEN mammal(x)

IF mammal(x) THEN animal(x)

elephant(Clyde)
```

unification:
find compatible values for variables

modus ponens:

```
IF p THEN q
p
―――――――――
q
```

```
IF mammal(   x   ) THEN animal(   x   )

IF elephant(Clyde) THEN mammal(Clyde)

elephant (Clyde)
```

Forward Chaining Example

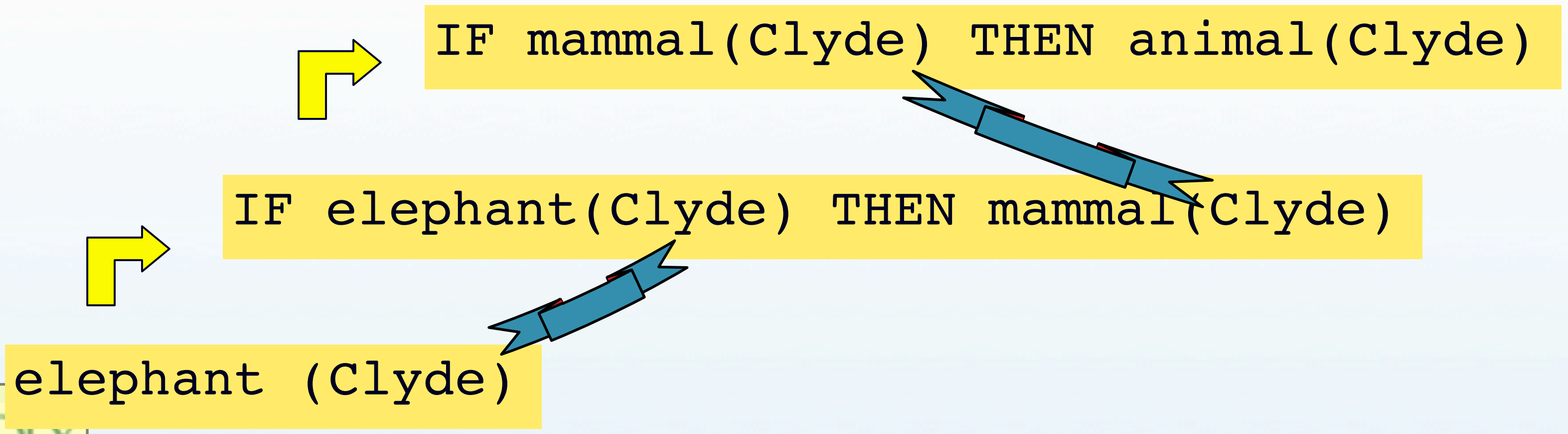
```
IF elephant(x) THEN mammal(x)

IF mammal(x) THEN animal(x)

elephant(Clyde)
```

unification:

find compatible values for variables

modus ponens:

```
IF p THEN q
p
q
```


IF mammal(Clyde) THEN animal(Clyde)
IF elephant(Clyde) THEN mammal(Clyde)
elephant (Clyde)

Forward Chaining Example

```
IF elephant(x) THEN mammal(x)
IF mammal(x) THEN animal(x)
elephant(Clyde)
```

unification:
find compatible values for variables

modus ponens:

```
IF p THEN q
p
q
```

```
animal(  x  )
IF mammal(Clyde) THEN animal(Clyde)
IF elephant(Clyde) THEN mammal(Clyde)
elephant (Clyde)
```

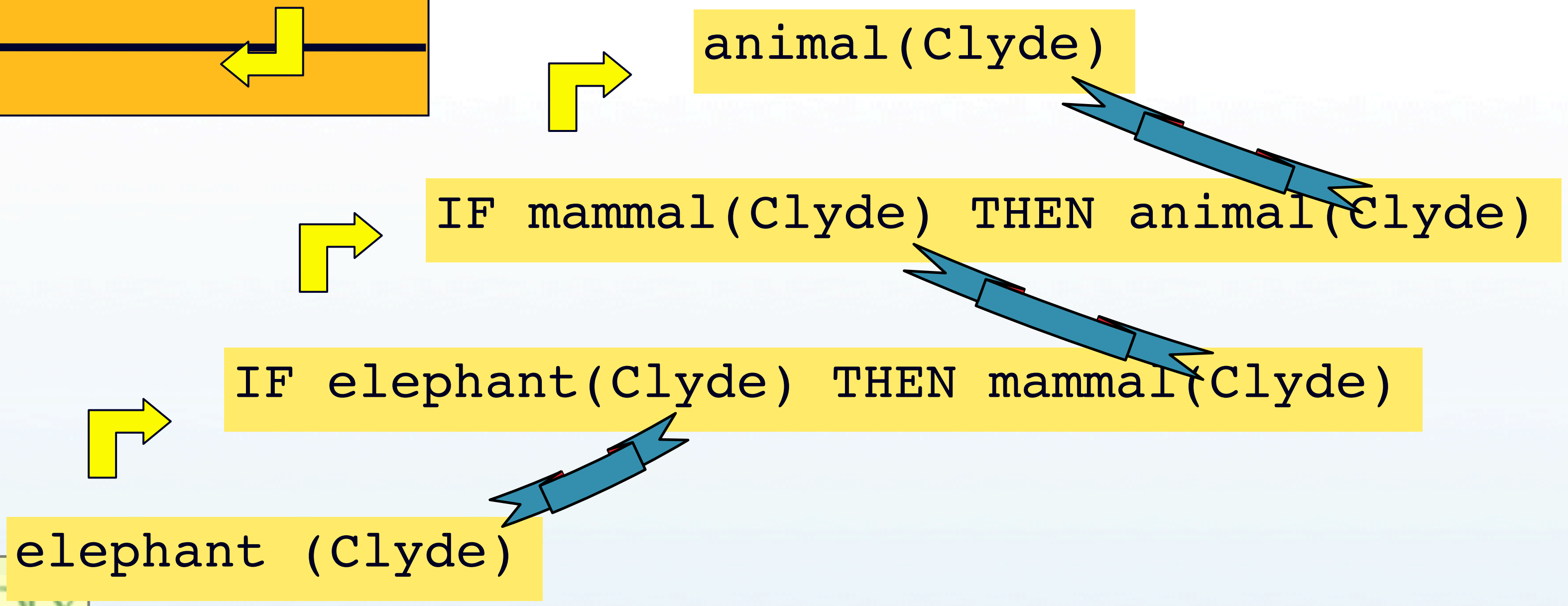
Forward Chaining Example

```
IF elephant(x) THEN mammal(x)
IF mammal(x) THEN animal(x)
elephant(Clyde)
```

unification:
find compatible values for variables

modus ponens:

$$\frac{\begin{array}{l}\texttt{IF p THEN q}\\ \texttt{p}\end{array}}{\texttt{q}}$$


animal(Clyde)
IF mammal(Clyde) THEN animal(Clyde)
IF elephant(Clyde) THEN mammal(Clyde)
elephant (Clyde)

Backward Chaining

- try to find supportive evidence (i.e. facts) for a hypothesis
- example: Is there evidence that Clyde is an animal?

```
IF elephant(x) THEN mammal(x)
IF mammal(x) THEN animal(x)
elephant (Clyde)
```

modus ponens:

```
IF p THEN q
p
-----------
q
```

unification:

find compatible values for variables

Backward Chaining Example

```
IF elephant(x) THEN mammal(x)
IF mammal(x) THEN animal(x)
elephant(Clyde)
```

unification:

find compatible values for variables

modus ponens:

```
IF p THEN q
p
-----------
q
```

```
animal(Clyde)
```

?

```
IF mammal(  x  ) THEN animal(  x  )
```

Backward Chaining Example

```
IF elephant(x) THEN mammal(x)
IF mammal(x) THEN animal(x)
elephant(Clyde)
```

unification: find compatible values for variables

modus ponens:

```
IF p THEN q
p
q
```

```
animal(Clyde)
```

?

```
IF mammal(Clyde) THEN animal(Clyde)
```

Backward Chaining Example

```
IF elephant(x) THEN mammal(x)
IF mammal(x) THEN animal(x)
elephant(Clyde)
```

unification: find compatible values for variables

modus ponens:

```
IF p THEN q
p
-----------
q
```

```
animal(Clyde)
```

?

```
IF mammal(Clyde) THEN animal(Clyde)
```

?

```
IF elephant(   x   ) THEN mammal(   x   )
```

Backward Chaining Example

```
IF elephant(x) THEN mammal(x)
IF mammal(x) THEN animal(x)
elephant(Clyde)
```

unification:

find compatible values for variables

modus ponens:

```
IF p THEN q
p
-----------
q
```

```
animal(Clyde)
```

?

```
IF mammal(Clyde) THEN animal(Clyde)
```

?

```
IF elephant(Clyde) THEN mammal(Clyde)
```

Backward Chaining Example

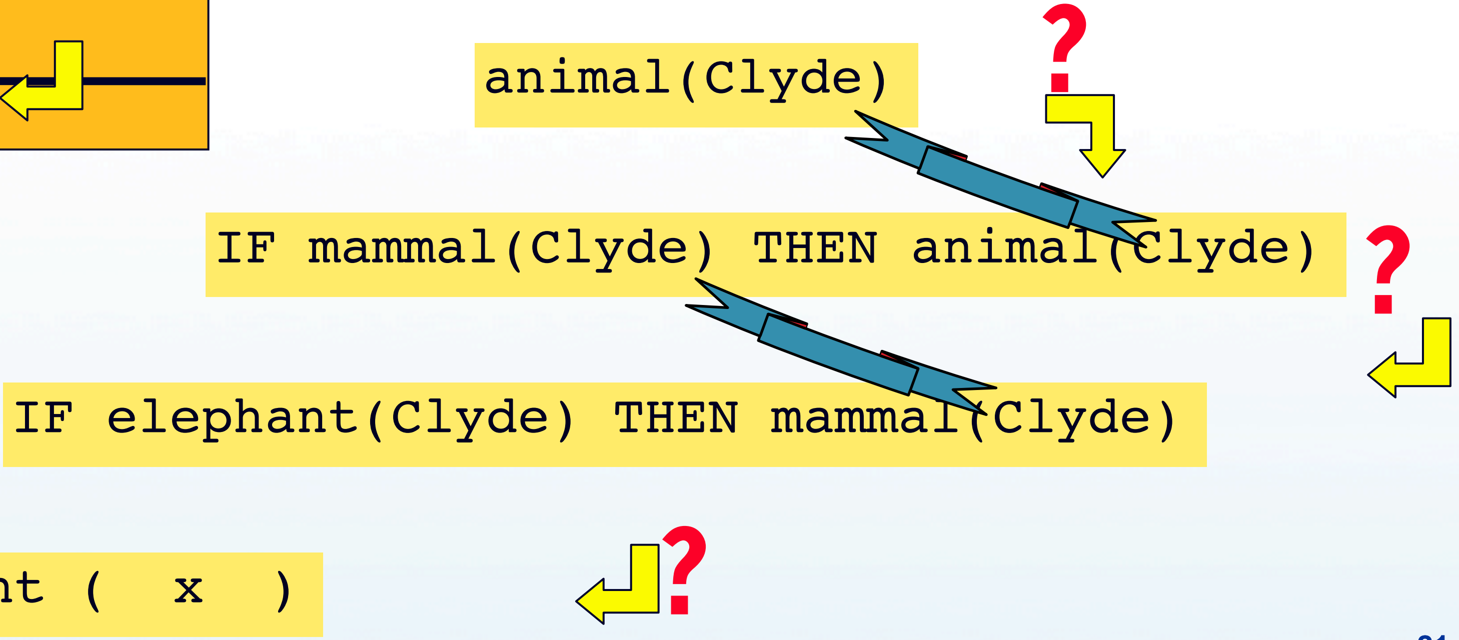
```
IF elephant(x) THEN mammal(x)
IF mammal(x) THEN animal(x)
elephant(Clyde)
```

unification:

find compatible values for variables

modus ponens:

```
IF p THEN q
p
-----------
q
```



```
animal(Clyde)
IF mammal(Clyde) THEN animal(Clyde)
IF elephant(Clyde) THEN mammal(Clyde)
elephant ( x )
```

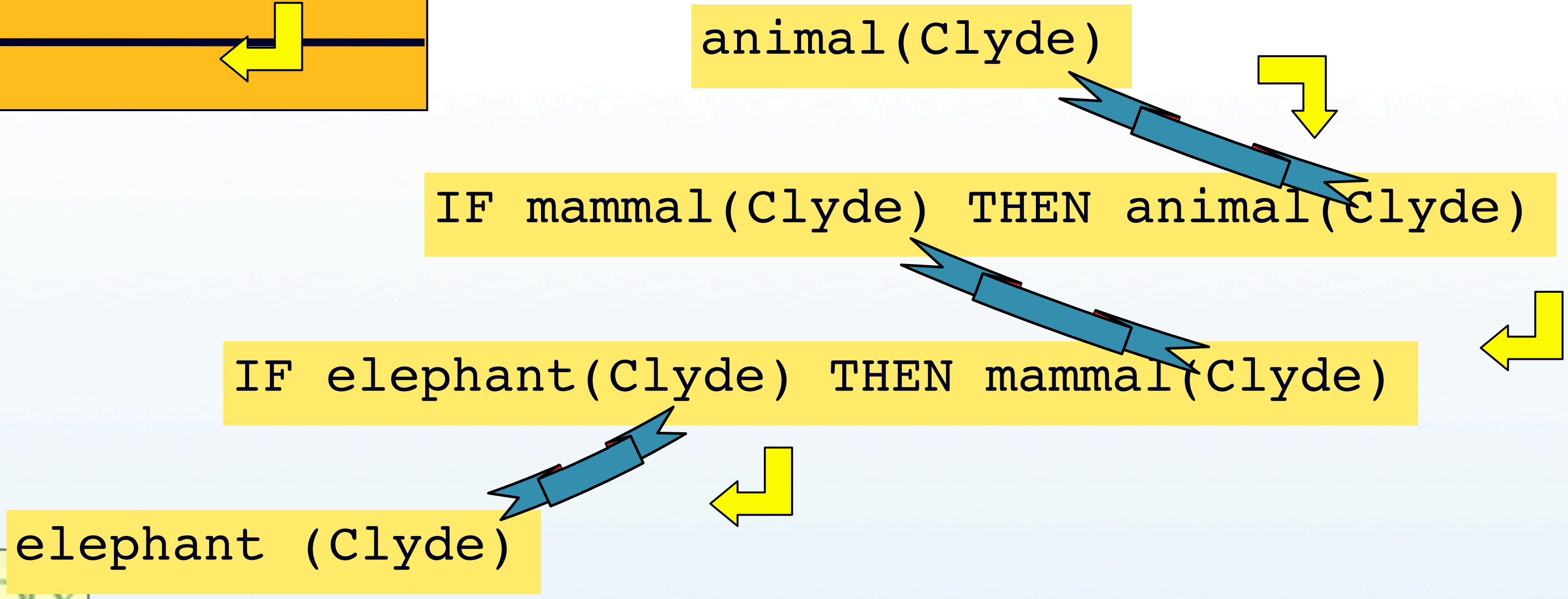
Backward Chaining Example

```
IF elephant(x) THEN mammal(x)
IF mammal(x) THEN animal(x)
elephant(Clyde)
```

unification: find compatible values for variables

modus ponens:

```
IF p THEN q
p
q
```



```
animal(Clyde)
IF mammal(Clyde) THEN animal(Clyde)
IF elephant(Clyde) THEN mammal(Clyde)
elephant (Clyde)
```

Forward vs. Backward Chaining

Forward Chaining	*Backward Chaining*
planning, control	diagnosis
data-driven	goal-driven (hypothesis)
bottom-up reasoning	top-down reasoning
find possible conclusions supported by given facts	find facts that support a given hypothesis
similar to breadth-first search	similar to depth-first search
antecedents (LHS) control evaluation	consequents (RHS) control evaluation

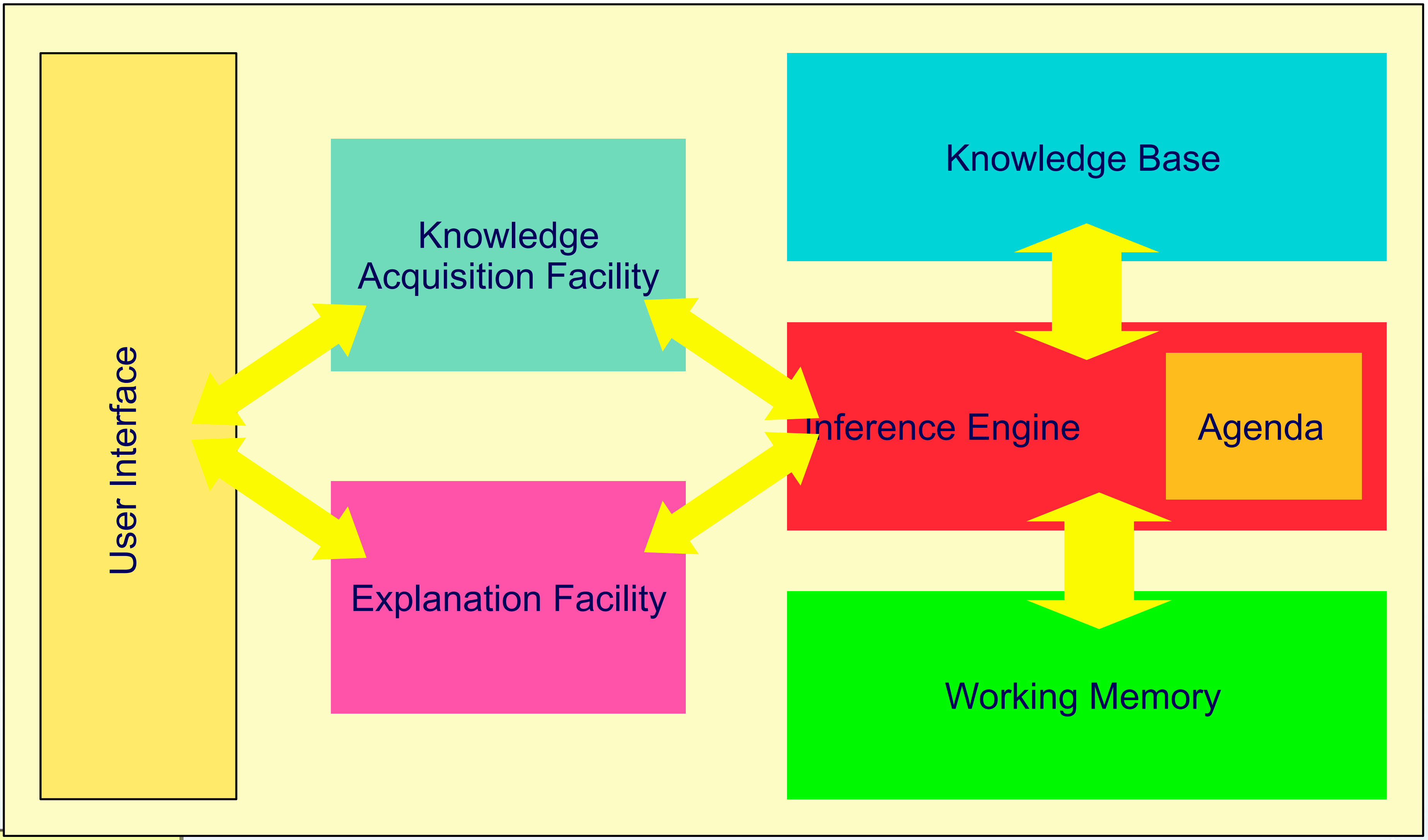
Reasoning in Rule-Based Systems

24

ES Elements

- knowledge base
- inference engine
- working memory
- agenda
- explanation facility
- knowledge acquisition facility
- user interface

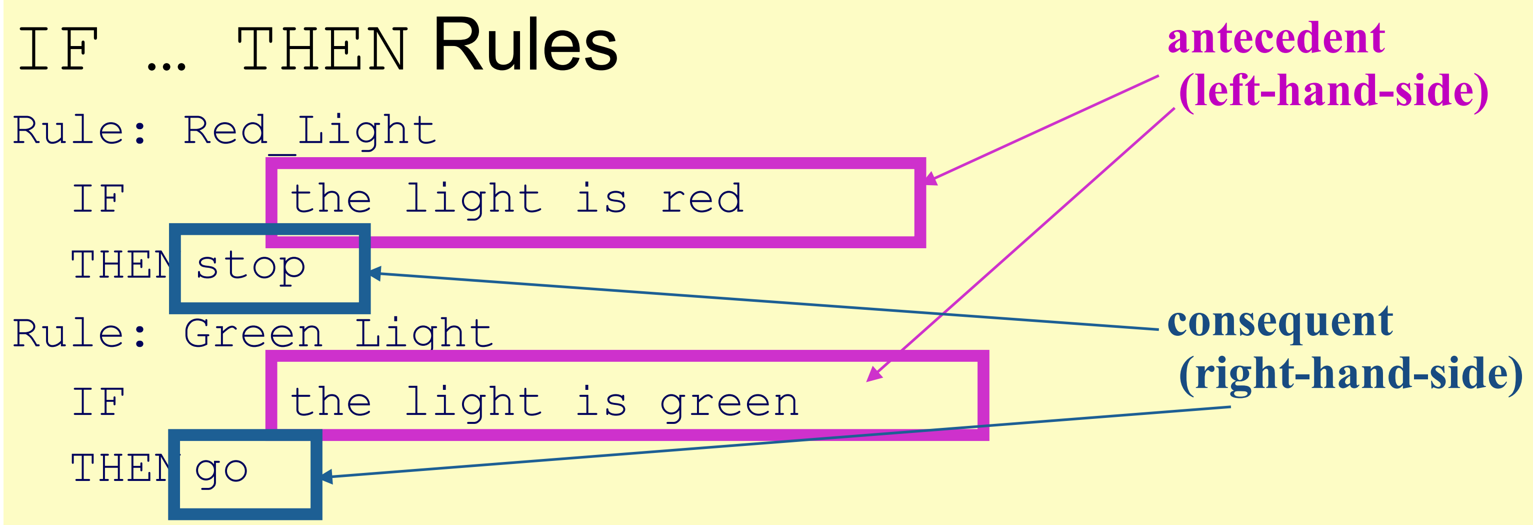
ES Structure



Rule-Based ES

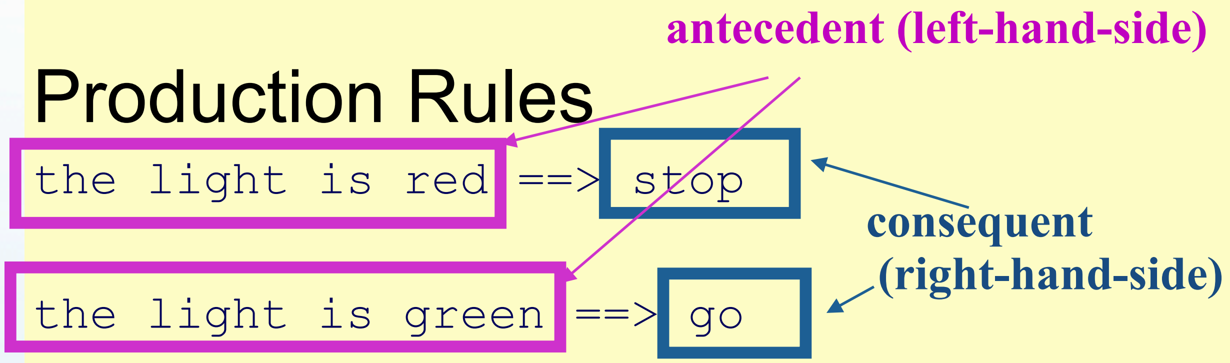
- knowledge is encoded as IF … THEN rules
  - these rules can also be written as production rules
- the inference engine determines which rule antecedents are satisfied
  - the left-hand side must "match" a fact in the working memory
- satisfied rules are placed on the agenda
- rules on the agenda can be activated ("fired")
  - an activated rule may generate new facts through its right-hand side
  - the activation of one rule may subsequently cause the activation of other rules

Example Rules



`IF … THEN` Rules

```
Rule: Red_Light
   IF        the light is red
   THEN stop
Rule: Green_Light
   IF        the light is green
   THEN go
```



Production Rules

```
the light is red ==> stop

the light is green ==> go
```

MYCIN Sample Rule

Human-Readable Format

IF the stain of the organism is gram negative

AND the morphology of the organism is rod

AND the aerobiocity of the organism is gram anaerobic

THEN the there is strongly suggestive evidence (0.8)

that the class of the organism is enterobacteriaceae

MYCIN Format

```
IF (AND (SAME CNTEXT GRAM GRAMNEG)
        (SAME CNTEXT MORPH ROD)
        (SAME CNTEXT AIR AEROBIC)
THEN (CONCLUDE CNTEXT CLASS ENTEROBACTERIACEAE
     TALLY .8)
```

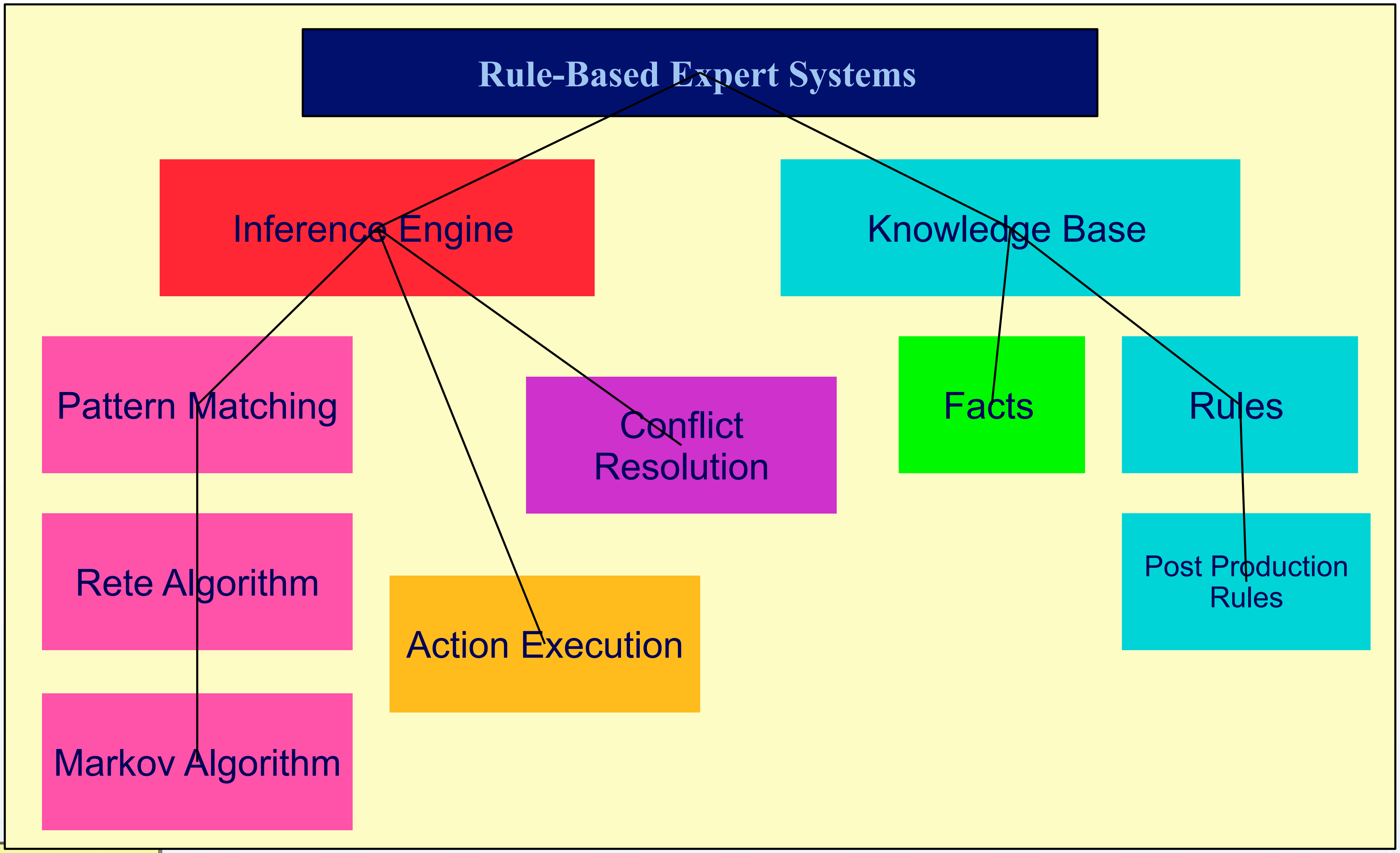
Inference Engine Cycle

- describes the execution of rules by the inference engine
  - conflict resolution
    - select the rule with the highest priority from the agenda
  - execution
    - perform the actions on the consequent of the selected rule
    - remove the rule from the agenda
  - match
    - update the agenda
      - add rules whose antecedents are satisfied to the agenda
      - remove rules with non-satisfied agendas
- the cycle ends when no more rules are on the agenda, or when an explicit stop command is encountered

Forward and Backward Chaining

- different methods of rule activation
  - forward chaining (data-driven)
    - reasoning from facts to the conclusion
    - as soon as facts are available, they are used to match antecedents of rules
    - a rule can be activated if all parts of the antecedent are satisfied
    - often used for real-time expert systems in monitoring and control
    - examples: CLIPS, OPS5
  - backward chaining (query-driven)
    - starting from a hypothesis (query), supporting rules and facts are sought until all parts of the antecedent of the hypothesis are satisfied
    - often used in diagnostic and consultation systems
    - examples: EMYCIN

Foundations of Expert Systems



Post Production Systems

- production rules were used by the logician Emil L. Post in the early 40s in symbolic logic
- Post's theoretical result
  - any system in mathematics or logic can be written as a production system
- basic principle of production rules
  - a set of rules governs the conversion of a set of strings into another set of strings
    - these rules are also known as rewrite rules
    - simple syntactic string manipulation
    - no understanding or interpretation is required
    - also used to define grammars of languages
      - e.g. BNF grammars of programming languages

Emil Post

- 20th century mathematician
- worked in logic, formal languages
  - truth tables
  - completeness proof of the proposition
    presented in Principia Mathematica
  - recursion theory
    - mathematical model of computation sim
      machine
- not related to Emily Post ;-)

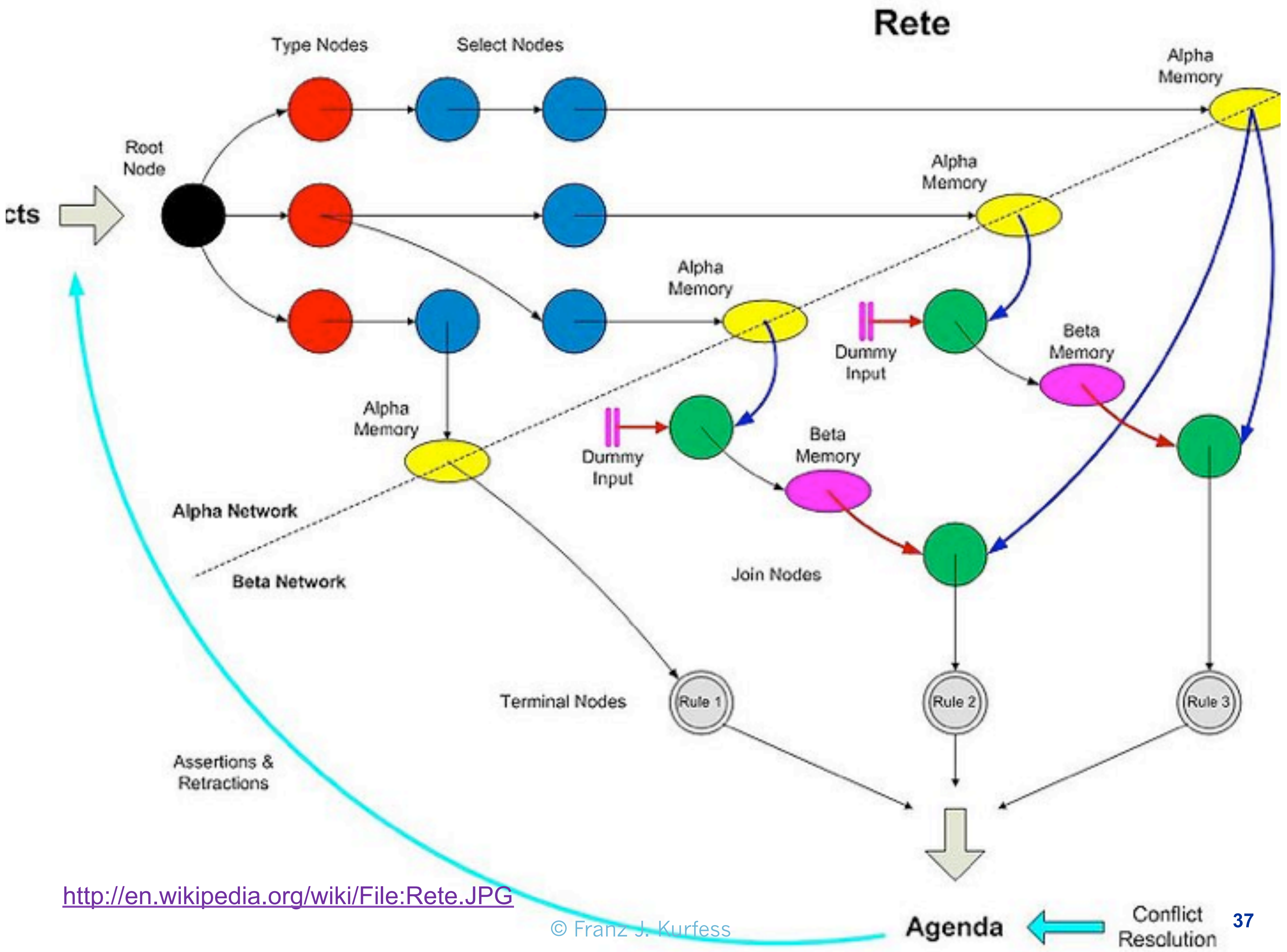
http://en.wikipedia.org/wiki/Emil_Post

Markov Algorithms

- in the 1950s, A. A. Markov introduced priorities as a control structure for production systems
  - rules with higher priorities are applied first
  - allows more efficient execution of production systems
  - but still not efficient enough for expert systems with large sets of rules
  - he is the son of Andrey Markov, who developed Markov chains

Rete Algorithm

- developed by Charles L. Forgy in the late 70s for CMU's OPS (Official Production System) shell
  - stores information about the antecedents in a network
  - in every cycle, it only checks for changes in the networks
  - this greatly improves efficiency



http://en.wikipedia.org/wiki/File:Rete.JPG

CLIPS Introduction

- CLIPS stands for
  - C Language Implementation Production System
- forward-chaining
  - starting from the facts, a solution is developed
- pattern-matching
  - Rete matching algorithm: find ``fitting'' rules and facts
- knowledge-based system shell
  - empty tool, to be filled with knowledge
- multi-paradigm programming language
  - rule-based, object-oriented (Cool) and procedural

The CLIPS Programming Tool

- history of CLIPS
  - influenced by OPS5 and ART
  - implemented in C for efficiency and portability
  - developed by NASA, distributed & supported by COSMIC
  - runs on PC, Mac, UNIX, VAX VMS
- CLIPS provides mechanisms for expert systems
  - a top-level interpreter
  - production rule interpreter
  - object oriented programming language
  - LISP-like procedural language

[Jackson 1999]

Components of CLIPS

- rule-based language
  - can create a fact list
  - can create a rule set
  - an inference engine matches facts against rules
- object-oriented language (COOL)
  - can define classes
  - can create different sets of instances
  - special forms allow you to interface rules and objects

CAL POLY

[Jackson 1999]

Invoke / Exit CLIPS

- entering CLIPS

  double-click on icon, or type program name (CLIPS)

  system prompt appears:

  ```
  CLIPS>
  ```

- exiting CLIPS

  at the system prompt

  ```
  CLIPS>
  ```

  type `(exit)`

  - Note: enclosing parentheses are important; they indicate a command to be executed, not just a symbol

Facts

- elementary information items ("chunks")
- relation name
  - symbolic field used to access the information
  - often serves as identifier for the fact
- slots (zero or more)
  - symbolic fields with associated values
- `deftemplate` construct
  - used to define the structure of a fact
    - names and number of slots
- `deffacts`
  - used to define initial groups of facts

Examples of Facts

- ordered fact

```
(person-name Franz J. Kurfess)
```

- deftemplate fact

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

Defining Facts

- Facts can be asserted

```
CLIPS> (assert (today is sunday))
<Fact-0>
```

- Facts can be listed

```
CLIPS> (facts)
f-0 (today is sunday)
```

- Facts can be retracted

```
CLIPS>  (retract 0)
CLIPS> (facts)
```

Instances

- an instance of a fact is created by

```
(assert (person (name "Franz J. Kurfess")
                (age 46)
                (eye-color brown)
                (hair-color brown))
)
```

Initial Facts

```
(deffacts kurfesses "some members of the Kurfess family"
   (person (name "Franz J. Kurfess") (age 46)
                (eye-color brown)     (hair-color brown))
   (person (name "Hubert   Kurfess") (age 44)
                (eye-color blue)      (hair-color blond))
   (person (name "Bernhard Kurfess") (age 41)
                (eye-color blue)      (hair-color blond))
   (person (name "Heinrich Kurfess") (age 38)
                (eye-color brown)     (hair-color blond))
   (person (name "Irmgard  Kurfess") (age 37)
                (eye-color green)     (hair-color blond))
)
```

Usage of Facts

- adding facts
  - `(assert <fact>+)`
- deleting facts
  - `(retract  <fact-index>+)`
- modifying facts
  - `(modify <fact-index> (<slot-name> <slot-value>)+ )`
    - retracts the original fact and asserts a new, modified fact
- duplicating facts
  - `(duplicate <fact-index> (<slot-name> <slot-value>)+ )`
    - adds a new, possibly modified fact
- inspection of facts
  - `(facts)`
    - prints the list of facts
  - `(watch facts)`
    - automatically displays changes to the fact list

Rules

- general format

```
(defrule <rule name> ["comment"]
    <patterns>* ; left-hand side (LHS)
                ; or antecedent of the rule
=>
    <actions>*) ; right-hand side (RHS)
                ; or consequent of the rule
```

Rule Components

- rule header
  - defrule keyword, name of the rule, optional comment string
- rule antecedent (LHS)
  - patterns to be matched against facts
- rule arrow
  - separates antecedent and consequent
- rule consequent (RHS)
  - actions to be performed when the rule fires

Examples of Rules

- simple rule

```
(defrule birthday-FJK
        (person (name "Franz J. Kurfess")
               (age 46)
               (eye-color brown)
               (hair-color brown))
   (date-today April-13-02)
=>
  (printout t "Happy birthday, Franz!")
  (modify 1 (age 47))
)
```

Properties of Simple Rules

- very limited:
  - LHS must match facts exactly
  - facts must be accessed through their index number
  - changes must be stated explicitly
- can be enhanced through the use of variables

Variables, Operators, Functions

- variables
  - symbolic name beginning with a question mark "?"
  - variable bindings
    - variables in a rule pattern (LHS) are bound to the corresponding values in the fact, and then can be used on the RHS
    - all occurrences of a variable in a rule have the same value
    - the left-most occurrence in the LHS determines the value
    - bindings are valid only within one rule
  - access to facts
    - variables can be used to make access to facts more convenient:

```
?age <- (age harry 17)
```

Wildcards

- question mark `?`
  - matches any single field within a fact
- multi-field wildcard `$?`
  - matches zero or more fields in a fact

Field Constraints

- not constraint ~
  - the field can take any value except the one specified
- or constraint |
  - specifies alternative values, one of which must match
- and constraint &
  - the value of the field must match all specified values
  - mostly used to place constraints on the binding of a variable

Mathematical Operators

- basic operators (`+,-,*,/`) and many functions (trigonometric, logarithmic, exponential) are supported
- prefix notation
- no built-in precedence, only left-to-right and parentheses
- test feature
  - evaluates an expression in the LHS instead of matching a pattern against a fact
- pattern connectives
  - multiple patterns in the LHS are implicitly `AND`-connected
  - patterns can also be explicitly connected via `AND, OR, NOT`
- user-defined functions
  - external functions written in C or other languages can be integrated
  - Jess is tightly integrated with Java

Examples of Rules

- more complex rule

```
(defrule find-blue-eyes
       (person (name ?name)
              (eye-color blue))
   =>
       (printout t ?name " has blue eyes.”
        crlf))
```

Example Rule with Field Constraints

```
(defrule silly-eye-hair-match
  (person (name ?name1)
    (eye-color ?eyes1&blue|green)
    (hair-color ?hair1&~black))
  (person (name ?name2&~?name1)
    (eye-color ?eyes2&~?eyes1)
    (hair-color ?hair2&red|?hair1))

  =>
  (printout t ?name1 " has "?eyes1 " eyes and "  ?
    hair1 " hair."    crlf)
  (printout t ?name2 " has "?eyes2 " eyes  and " ?
    hair2 " hair."    crlf))
```

Using Templates

```
(deftemplate student “a student record”
   (slot name (type STRING))
   (slot age (type NUMBER) (default 18)))

  CLIPS> (assert (student (name fred)))


(defrule print-a-student
   (student (name ?name) (age ?age))
  =>
   (printout t ?name “ is “ ?age))
```

An Example CLIPS Rule

```
(defrule sunday "Things to do on Sunday"
   (salience 0)    ; salience in the interval [-10000,
     10000]
   (today is Sunday)
   (weather is sunny)
  =>
   (assert (chore wash car))
   (assert (chore chop wood)))
```

Getting the Rules Started

- The reset command creates a special fact

```
CLIPS> (load “today.clp”)
CLIPS> (facts)
CLIPS> (reset)
CLIPS> (facts)
f-0 (initial-fact) ...
```

```
(defrule start
   (initial-fact)
  =>
   (printout t “hello”))
```

Variables & Pattern Matching

- Variables make rules more applicable

```
(defrule pick-a-chore
   (today is ?day)
   (chore is ?job)
  =>
   (assert (do ?job on ?day)))
```

- if conditions are matched, then bindings are used

[Jackson 1999]

Retracting Facts from a Rule

```
(defrule do-a-chore
  (today is ?day)      ; ?day must have a
    consistent binding
  ?chore <- (do ?job on ?day)
  =>
  (printout t ?job " done")
  (retract ?chore))
```

- a variable must be assigned to the item for retraction

Pattern Matching Details

- one-to-one matching

```
(do ?job on ?day)
(do washing on monday)
```

- use of wild cards

```
(do ? ? monday)
(do ? on ?)
(do ? ? ?day)
(do $?)
(do $? monday)
(do ?chore $?when)
```

[Jackson 1999]

Manipulation of Constructs

- show list of constructs

  `(list-defrules), (list-deftemplates), (list-deffacts)`

  - prints a list of the respective constructs

- show text of constructs

  `(ppdefrule <defrule-name>), (ppdeftemplate <deftemplate-name>), (ppdeffacts <deffacts-name>)`

  - displays the text of the construct (``pretty print")

- deleting constructs

  `(undefrule <defrule-name>), (undeftemplate <deftemplate-name>), (undeffacts <deffacts-name>)`

  - deletes the construct (if it is not in use)

- clearing the CLIPS environment

  `(clear)`

  - removes all constructs and adds the initial facts to the CLIPS environment

Input / Output

- print information

  `(printout <logical-device> <print-items>*)`
  - logical device frequently is the standard output device t (terminal)
- terminal input

  `(read [<logical-device>]), (readline [<logical-device>])`
  - read an atom or string from a logical device
  - the logical device can be a file which must be open
- open / close file

  `(open <file-name> <file-ID> [<mode>]), (close [<file-ID>])`
  - open /close file with <file-id> as internal name
- load / save constructs from / to file

  `(load <file-name>), (save <file-name>)`
  - backslash \ is a special character and must be ``quoted'' (preceded by a backslash \)
    - e.g. (load "B:\\clips\\example.clp")

Program Execution

- agenda
  - if all patterns of a rule match with facts, it is put on the agenda
    - `(agenda)` displays all activated rules
- salience
  - indicates priority of rules
- refraction
  - rules fire only once for a specific set of facts
    - prevents infinite loops
  - `(refresh <rule-name>)`
    - reactivates rules

Execution of a Program

- `(reset)` prepares (re)start of a program:
  - all previous facts are deleted
  - initial facts are asserted
  - rules matching these facts are put on the agenda
- `(run [<limit>])` starts the execution
- breakpoints
  - `(set-break [<rule-name>])`
    - stops the execution before the rule fires,
    - continue with `(run)`
  - `(remove-break [<rule-name>])`
  - `(show-breaks)`

Watching

- watching the execution
  - `(watch <watch-item>)` prints messages about activities concerning a `<watch-item>`
    - `(facts, rules, activations, statistics, compilation, focus, all)`
  - `(unwatch <watch-item>)`
    - turns the messages off

Watching Facts, Rules and Activations

- facts
  - assertions (add) and retractions (delete)
  - of facts
- rules
  - message for each rule that is fired
- activations
  - activated rules: matching antecedents
  - these rules are on the agenda

More Watching ...

- statistics
  - information about the program execution
  - (number of rules fired, run time, ...)
- compilation (default)
  - shows information for constructs loaded by `(load)`
    - `Defining deftemplate: ...`
    - `Defining defrule: ... +j=j`
      - +j, =j indicates the internal structure of the compiled rules
        - +j join added
        - =j join shared
      - important for the efficiency of the Rete pattern matching network
- focus
  - used with modules
  - indicates which module is currently active

User Interface

- menu-based version
  - most relevant commands are available through windows and menus
- command-line interface
  - all commands must be entered at the prompt
  - (don’t forget enclosing parentheses)

Limitations of CLIPS

- single level rule sets
  - in LOOPS, you could arrange rule sets in a hierarchy, embedding one rule set inside another, etc
- loose coupling of rules and objects
  - rules can communicate with objects via message passing
  - rules cannot easily be embedded in objects, as in Centaur
- CLIPS has no explicit agenda mechanism
  - the basic control flow is forward chaining
  - to implement other kinds of reasoning you have to manipulate tokens in working memory

Alternatives to CLIPS

- JESS
  - see below
- Eclipse
  - enhanced, commercial variant of CLIPS
  - has same syntax as CLIPS (both are based on ART)
  - supports goal-driven (i.e., backwards) reasoning
  - has a truth maintenance facility for checking consistency
  - can be integrated with C++ and dBase
  - new extension RETE++ can generate C++ header files
  - not related to the (newer) IBM Eclipse environment
- NEXPERT OBJECT
  - another rule- and object-based system
  - has facilities for designing graphical interfaces
  - has a 'script language' for designing user front-end
  - written in C, runs on many platforms, highly portable

[Jackson 1999]

JESS

- JESS stands for Java Expert System Shell
- it uses the same syntax and a large majority of the features of CLIPS
- tight integration with Java
  - can be invoked easily from Java programs
  - can utilize object-oriented aspects of Java
- some incompatibilities with CLIPS
  - COOL replaced by Java classes
  - a few missing constructs
    - more and more added as new versions of JESS are released

Post-Test

CLIPS Summary

- notation
  - similar to Lisp, regular expressions
- facts
  - `(deftemplate), (deffacts), assert / retract`
- rules
  - `(defrule ...)`, agenda
- variables, operators, functions
  - advanced pattern matching
- input/output
  - `(printout ...), (read ...), (load ...)`
- program execution
  - `(reset), (run), breakpoints`
- user interface
  - command line or GUI

Important Concepts and Terms

- agenda
- antecedent
- assert
- backward chaining
- consequent
- CLIPS
- expert system shell
- fact
- field
- forward chaining
- function
- inference
- inference mechanism
- instance
- If-Then rules
- JESS
- knowledge base
- knowledge representation
- pattern matching
- refraction
- retract
- rule
- rule header
- salience
- template
- variable
- wild card