CLIPS

1

CLIPS PROGRAMMING

- Basic Commands
- Symbols, Data Types, Syntax
- Templates, Facts, Rules
- Variables and Pattern Matching
- Basic I/O, File I/O, Built-in Functions
- Math/Logical Expressions

2

History of CLIPS

- **Stands for "C Language Integrated Production System"**
- Developed at NASA (1986)
- Implemented in C
- Influenced by OPS5 and ART languages
- Initial version was only a production rule interpreter.
- Latest version is named COOL (CLIPS Object-Oriented Language)

3

CLIPS Programming Tool

- It is a classical Rule-Based (Knowledge-Based) expert system shell: Empty tool, to be filled with knowledge.
- It is a Forward Chaining system: Starting from the facts, a solution is developed.
- Its inference engine internally uses the Rete Algorithm for pattern-matching : Find fitting rules and facts.

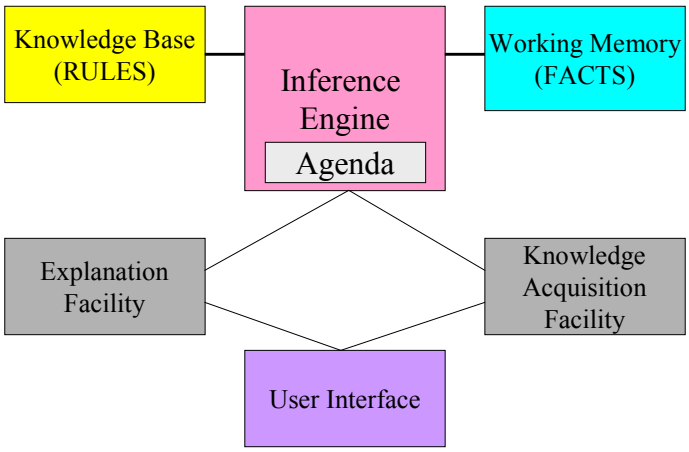
4

Advantages of CLIPS

- It is a high-level production rule interpreter (shell).
- Syntax is similar to LISP.
- Facts and rule-base is similar to Prolog.
- Higher-level compared to LISP or Prolog.
- Runs on UNIX, Linux, DOS, Windows, MacOS.
- A public-domain and well-documented software.
- Includes object-oriented constructs (COOL).

5

Components of a Rule-Based Expert System



6

Components of a Rule-Based Language (1)

- FACT BASE or fact list represents the initial state of the problem. *This is the data from which inferences are derived.*
- RULE BASE or Knowledge Base (KB) contains a set of rules which can transform the problem state into a solution. It is the set of all rules.
- CLIPS supports only forward chaining rules.

7

Components of a Rule-Based Language (2)

- INFERENCE ENGINE controls overall execution. It matches the facts against the rules to see what rules are applicable. It works in a *recognize-act cycle:*
  1) match the facts against the rules
  2) choose which rules instantiation to fire
  3) execute the actions associated with the *winning* rule

8

Basic CLIPS Commands (1)

- (exit) to exit from CLIPS
- (clear) to clear the environment from facts, rules, and other active definitions
- (reset) to set the fact base to its initial state (clears existing facts; sets (initial-fact), and all (deffacts) constructs in the program). Perform (reset) before each program run!
- (run) executes a program currently loaded into the CLIPS interpreter against currently defined rule-bases and fact-bases.

9

Basic CLIPS Commands (2)

- (load “filename.clp”) to load a CLIPS program into the interpreter from the file named filename.clp . This also does syntax check and makes constructs in the file defined.
- (facts) to display a list of currently active facts in the fact base.
- (rules) to display a set of rules currently in the rule base.
- (agenda) to display all potential matches of active facts for all rules.

10

Syntax Notation

- symbols, characters, keywords
  - entered exactly as shown:
  - (example)
- square brackets [...]
  - contents are optional:
  - (example [test])
- pointed brackets < ... >
  - replace contents by an instance of that type
  - (example <char>)
- star *
  - replace with zero or more instances of the type
  - <char>*
- plus +
  - replace with one or more instances of the type
  - <char>+ (is equivalent to <char> <char>*)
- vertical bar |
  - choice among a set of items:
  - true | false

11

Primitive Data Types

- **float**: decimal point (`61.275`) or
  exponential notation (`3.7e10`)
- **integer**: `[sign] <digit>+`
- **symbol**: `<printable ASCII character>+`
  - e.g. `this-is-a-symbol, wrzlbrmft, !?@*+`
- **string**: delimited by double quotes
  - e.g. `"This is a string"`
- **external address**
  - address of external data structure returned by user-defined functions

12

Comments

- Program comments begin with a semicolon "**;**".

```
;This is a comment example
```

- Construct comments are used as a part of the CLIPS constructs (e.g. deftemplate, defrule, etc) quotations.

```
(defrule  my-rule  "my comment"
      (initial-fact)
  =>
      (printout t "Hello" crlf)
)
```

13

EXAMPLE-1:

```
(defrule basic
 =>
   (printout  t  "Hello, world!"  crlf)
)
```

EXAMPLE-2: (Same as above)

```
(defrule basic
  (initial-fact)
 =>
   (printout  t  "Hello, world!"  crlf)
)
```

14

To Make It Run

- Type the code in a file and save it (e.g. hello-world.clp)
- Start CLIPS
- Type (load "hello-world.clp")
- When the file is loaded CLIPS will display the followings:
  defining defrule basic +j
  TRUE
- Type (reset)
- Type (run)

  Tip: You can also use the menu or hot keys for these commands:

  ^E for reset

  ^R for run

To **exit** CLIPS use the menu, **^Q** or (exit)

15

EXAMPLE

```
(defrule is-it-a-duck
  (animal-has  webbed-feet)
  (animal-has  feathers)
  =>
  (assert (animal-is  duck)))
```

```
(defrule duck
  (animal-is  duck)
  =>
  (assert (sound-is  quack))
  (printout t "It is a duck!" crlf))
```

After loading the above program, let's enter the following commands:

```
CLIPS> (reset)
CLIPS> (assert (animal-has  webbed-feet))
CLIPS> (assert (animal-has  feathers))
CLIPS> (run)
```

Rules are fired automatically and the following output is generated:

It is a duck!

16

A PROLOG PROGRAM

```
%% Facts:
father(tom, john).    %% tom is father of john
mother(susan, john).  %% susan is mother of john
father(george, tom).  %% george is father of tom

%% Rules:
parent(X, Y) :- father(X, Y) , mother(X, Y).
grandparent(X, Z) :- parent(X, Y) , parent(Y, Z).
grandfather(X, Z) :- father(X, Y) , parent(Y, Z).
grandmother(X, Z) :- mother(X, Y) , parent(Y, Z).
```

17

A PROLOG SESSION

```
Welcome to SWI-Prolog (Version 5.0.10)
Copyright (c) 1990-2002 University of Amsterdam.

?- consult (ornek.pl).
Yes
?- parent(A, B).
A = tom
B = john ;

A = george
B = tom ;

A = susan
B = john ;

?- grandparent(A, B).
A = george
B = john ;

?- grandfather(A, B).
A = george
B = john ;

?-
```

18

A CLIPS PROGRAM

```
; Facts:
(deffacts families
  (father tom john)    ; tom is father of john
  (mother susan john)  ; susan is mother of john
  (father george tom)) ; george is father of tom

; Rules:
(defrule parent-rule
   (or (father ?x ?y) (mother ?x ?y))
 =>
   (assert (parent ?x ?y)))

(defrule grandparent-rule
   (and (parent ?x ?y) (parent ?y ?z))
 =>
   (assert (grandparent ?x ?z)))

(defrule grandfather-rule
   (and (father ?x ?y) (parent ?y ?z))
 =>
   (assert (grandfather ?x ?z)))
```

19

A CLIPS SESSION

```
CLIPS> (load "ornek.clp")

Defining deffacts: families
Defining defrule: parent-rule +j+j
Defining defrule: grandparent-rule +j+j
Defining defrule: grandfather-rule =j+j
TRUE

CLIPS> (reset)
CLIPS> (run)
CLIPS> (facts)

f-0     (initial-fact)
f-1     (father tom john)
f-2     (mother susan john)
f-3     (father george tom)
f-4     (parent george tom)
f-5     (parent susan john)
f-6     (parent tom john)
f-7     (grandparent george john)
f-8     (grandfather george john)
For a total of 9 facts.

CLIPS>
```

20

Fields

- There are seven data types (types of tokens) called fields in CLIPS.
  - float: [+\-] <digit>* [.<digit>*] [e|E[+|-]<digit>*]
  - integer: [+|-] <digits> *
  - symbol: <char>+
  - string: “<char>* “ (e.g. “John”, “848-3000”)
  - external address
  - instance name
  - instance address
- a word CANNOT *start* with these:
  `< | & $ ? + - ( ) ;`
- a word CANNOT *contain* any of these:
  `< | & ( ) ;`

21

Expressions (1)

- Examples of valid words
  - computer
  - emergency-fire
  - activate_sprinkler_system
  - shut-down-electrical-junction-387
  - !?#$^*
- CLIPS is case-sensitive
  - Computer, COMPUTER, Computer are all different

22

Expressions (2)

- Examples of valid strings
  - “Activate the sprinkler system.”
  - “Shut down electrical junction 387.”
  - “!?#$^”
  - “<-;( ) +-”
- Spaces act as delimiters to separate fields
  - These are different strings "fire", "fire ", " fire", " fire " but would be the same with no quotes
- Valid numbers
  - 1 1.5 .7 +3 -1 65 3.5e10

23

Facts

Fact is a chunk of information consisting of a relation name, zero or more slots and slot values.

Examples:

(Fire)

(speed 50 km)

(cost 78 dollars 23 cents)

(pers-name “Shahram Rahimi ”)

(person (name “John Dıllards”) (age 24))

(person (name “Raheel Ahmad”) (age 25))

24

Fact examples

- Example 1
  - (fire)
  - (flood)
  - (Tuesday)
  - (Wednesday)

Example 2
- (emergency-fire)
- (emergency-flood)
- (day-Tuesday)
- (day-Wednesday)

- Example 3 - describes relationships
  - (emergency fire)
  - (emergency flood)
  - (day Tuesday)
  - (day Wednesday)

25

Commands for Facts

- Facts can be asserted

  CLIPS> (assert (emergency fire))

  <Fact-0>

- Facts can be listed

  CLIPS> (facts)

  f-0 (emergency fire)

- Facts can be retracted

  CLIPS> (retract 0)

  CLIPS> (facts)

26

Initial Facts

- `deffacts` used to define initial groups of facts.
- Facts from `(deffacts)` are automatically asserted using `(reset) command.`

```
(deffacts Expert_Systems “Class List"
  (person (name “Mike Lambert”) (age 18))
  (person (name “John Sayfarth”) (age 20))
  (person (name “Bill Parker”) (age 18))
  (person (name “Matuah Matuah”) (age 23))
)
```

27

Example:

```
(deffacts status “Some facts about the emergency”
   (emergency   fire)
   (fire-class   A)
   (fire-class   B))
```

Facts entered using deffacts are automatically asserted with (reset) command.

```
CLIPS> (reset)
CLIPS> (facts)
f-0  (initial-fact)
f-1  (emergency   fire)
f-2  (fire-class   A)
f-3  (fire-class   B)
CLIPS>
```

28

IDENTICAL FACTS:

EXAMPLE-1: By default, identical facts are not allowed.

```
CLIPS> (reset)
CLIPS> (assert (name Shahram))
<Fact-1>
CLIPS> (assert (name Shahram))
FALSE
CLIPS> (facts)
f-0     (initial-fact)
f-1     (name Shahram)
For a total of 2 facts.
```

EXAMPLE-2:: Identical facts can be defined in deffacts.However, only one of them is put on working memory.

```
CLIPS> (deffacts isimler
                (name Shahram)
                (name Shahram))
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (name Shahram)
For a total of 2 facts.
```

29

EXAMPLE-3: CLIPS can allow the definition of identical facts, when the following command is used first..

```
CLIPS> (set-fact-duplication TRUE)
TRUE
CLIPS> (deffacts isimler
                (name Shahram)
                (name Shahram))
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (name Shahram)
f-2     (name Shahram)
For a total of 3 facts.
```

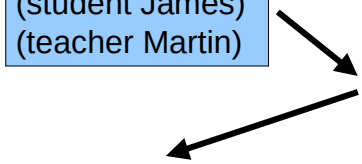
30

Example:

Instead of deffacts, facts can be saved and loaded to/from a seperate fact file.

"names.dat" file

(student George)
(student James)
(teacher Martin)



```
CLIPS> (load-facts "names.dat")
CLIPS> (facts)
f-0     (student George)
f-1     (student James)
f-2     (teacher Martin)
For a total of 3 facts.
CLIPS> (assert (teacher Peter))
<Fact-3>
CLIPS> (facts)
f-0     (student George)
f-1     (student James)
f-2     (teacher Martin)
f-3     (teacher Peter)
For a total of 4 facts.
CLIPS> (save-facts "names.dat")
```


"names.dat" file

(student George)
(student James)
(teacher Martin)
(teacher Peter)

31

Fact templates and instances

`deftemplate` construct is used to define the structure of a fact.

```
(deftemplate <relation_name> [<comment>]
  <slot-definition>*
)
```

`<slot-definition>` is one of the followings:

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

32

(deftemplate) Example

```
CLIPS> (deftemplate course “course information ”
                    (slot number)
                    (slot name))
CLIPS> (assert (course (number CS420)
                       (name “Distributed Computing”))
               (course (number CS401)
                       (name “Computer Architecture”)))
CLIPS> (facts)
f-0 (course (number CS420) (name “Distributed Computing”))
f-1 (course (number CS401) (name “Computer Architecture”)))
For a total of 2 facts

CLIPS> (retract 1)
CLIPS> (facts)
f-0 (course (number CS420) (name “Distributed Computing”))
For a total of 1 fact
```

33

Commands for 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

34

Modifying Facts

To modify a fact:

(modify <fact-index> <slot-modifier>***)**

<slot-modifier> is (<slot-name> <slot-value>)

Example:

```
CLIPS> (modify 0 (number CS520))
CLIPS> (facts)
f-0 (course (number 520) (name “Distributed Computing”))
for a total of 1 fact
```

35

Duplicating Facts

To create a duplicate of a fact:

Example:

```
CLIPS> (duplicate 0 (number CS420))
<fact-1>

CLIPS> (facts)
f-0 (course (number CS520) (name “Distributed Computing”))
f-1 (course (number CS420) (name “Distributed Computing”))
For a total of 2 facts
```

Note: (duplicate) modifies a fact without deleting (retracting) the original, whereas (modify) does.

36

Retracting Facts

- Facts can be removed or retracted using
  `(retract <fact-index>)`
  `(retract 2)`
- Retract can be used for more than one fact
  `(retract 1 2)`

37

(deftemplate) as a record

- Templates can be extended to hold more information like records:

(deftemplate <deftemplate-name> [<optional comment>]
(slot <slot-name> (type <data-type>) [(default <value>)])

- Similar to **struct** in C language

- **Example:**

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

38

Example

- After the template declaration and adding:

```
(deffacts students
   (student (name John))
   (student (name Steve) (age 19)))
```

- The result is:

```
(student (name John) (age 18))
(student (name Steve) (age 19))
```

Ordered Facts

- Ordered facts
  - Relation name with no corresponding deftemplate
  - Have single, implied multifield slot
  - Implied deftemplate automatically created
  - E.g., `(number-list 7 9 3 4) ==`

```
(deftemplate number-list (multislot values))
(number-list (values 7 9 3 4))
```

  - Good for facts with single slot

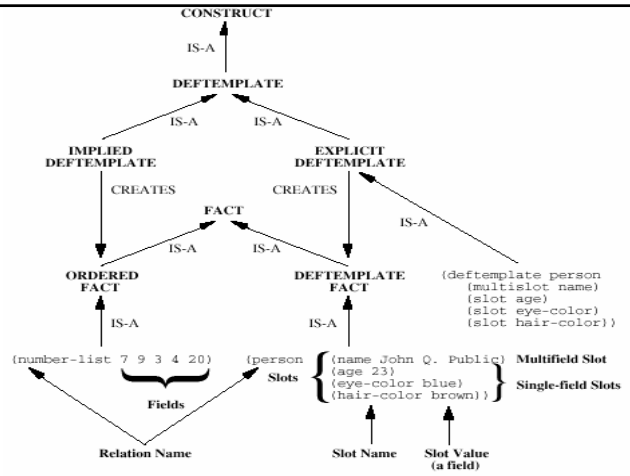


Figure 7.1 Deftemplate Overview

183

Rules (1)

- Syntax:

```
(defrule <rule-name> [<comment>]
    [<declaration>] ; salience
    <patterns>*   ; LHS, premises, patterns,
                  ; conditions, antecedent
=>
    <actions>*)  ; RHS, actions, consequent
```

Rules (2)

- Example:

```
(defrule emergency-type-earthquake
    (emergency earthquake)
=>
    (printout t “There is a earthquake!” crlf))
```

- Rules can have more than one pattern/premise:

```
(defrule emergency-type-earthquake
    (emergency earthquake)
    (degree B)
=>
 (assert (disaster serious))
 (printout t “There is a serious earthquake!” crlf))
```

41

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

42

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

43

Variables (1)

- Variable name is made of ? and one or more characters:
- Example:
  ```
  (course (number ?cmp))
  ```
- Variables are used for
  - Pattern matching
  - I/O
  - As pointers to facts (fact indices)

44

Variables (2)

- 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)
```

45

Variable Example

```
(defrule grandfather
    (is-a-grandfather ?name)
 =>
    (assert (is-a-father ?name))
    (assert (is-a-man ?name))
    (printout t ?name " is a grandfather" crlf)
 )
```

46

Example

```
(deffacts animal-fact-base
   (animal dog)
   (animal cat)
   (animal duck)
   (animal turtle))

(defrule list-animals
     (animal ?name)
=>
    (printout  t  ?name " found" crlf))
```

This program will produce the following results:

CLIPS>**(reset)**
CLIPS>**(run)**
turtle found
duck found
cat found
dog found

CLIPS>

47

Defining Global Variables

```
(defglobal
    ?*income* = 250000
    ?*name* = “John Lambert"
)
```

48

Fact Address

- To remove a fact from the fact-list use `(retract)`
- Before a fact can be retracted it must be specified to CLIPS by its index.
- Rules can be used to modify the fact base. To achieve it variables have to be bound to fact addresses using '`<-`':
  `?num <- (course (number ?cmp))`
- This appears in the LHS of the rule, and can be referred to in either LHS and RHS.
- **Example:**

```
(defrule remove-grandfather-fact
   ?adr <- (is-a-grandfather Jamie)
  =>
   (retract ?adr))
```

49

Wildcards (1)

To specify a general pattern we can use wildcards:

- For single field variables: wildcard ?
- For multifield variables (zero or more) wildcard $?

Examples:

(courses (numbers $?course_nums))
(printout t "Your courses are" $?course_nums crlf))
(list ? $? c ?)

- can match these:
  (list a b f c e), (list a d c b)
- but not these:
  (list c), (list c d), (list a c d b)

50

Wildcards (2)

- The fact
  `(do picnic on Sunday)`
- will match any of the following
  - `(do ? ? Sunday)`
  - `(do ? on ?)`
  - `(do ? on ?when)`
  - `(do $? )`
  - `(do $? Sunday)`
  - `(do ?chore $?when)`

51

EXAMPLE: Usage of $?

```
(deffacts ESFact
  (ES gives soul to programs))

(defrule R-1
  (ES $?what)
=>
  (printout t "$?what : " $?what crlf ))

(defrule R-2
  (ES $?what)
=>
  (printout t "?what : " ?what crlf ))

(defrule R-3
  (ES ?  $?what2)
=>
  (printout t "$?what2: " $?what2 crlf ))
```

CLIPS SESSION:

```
CLIPS> (load “prog.clp”)
CLIPS> (reset)
CLIPS> (run)
$?what : (gives soul to programs)
?what : (gives soul to programs)
$?what2 : (soul to programs)
```

52

Retracting Facts Using Wildcards

```
(defrule change-grandfather-fact
   ?old-fact <- (is-a-grandfather ?name)
=>
   (retract ?old-fact)
   (assert (has-a-grandchild ?name)
   (is-a-man ?name))
)
```

- You can retract several facts:
  (retract ?fact1 ?fact2 ?fact3)
- Or you can retract all of them at once:
  (retract *)

53

To Display Constructs

- To display constructs:

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

- To display the text of definitions of the constructs:

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

54

To Delete Constructs

- To ”undefine“ a given construct:

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

55

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

56

Field Constraints

- NOT **~** `(number ~comp672)`
- OR **|** `(number comp672|comp674)`
- AND **&**
  - `(number ?course_n&comp674|comp675)`
    - Variable `?course_n` will be bound to **both**
  - `(number ?course_n&~comp674&~comp672)`
    - Variable `?course_n` will be bound to **none** of the two

57

Field Constraints Examples

```
(deffacts animals
(animal Fido    dog    domestic)
(animal Sam     cat    domestic)
(animal Donald  duck   wild)
(animal Ninja   turtle wild))
```

- **Example of NOT:**

```
(defrule animal-not-dog
      (animal ?name ~dog ?)
=>
      (printout t ?name " is not a dog" crlf))
```

- **Example of OR:**

```
(defrule dog-or-cat
      (animal ?name dog|cat ?)
=>
      (printout t ?name " is a dog or cat" crlf))
```

- **Example of AND:**

```
(defrule dog-or-cat
      (animal ?name ?type&dog|cat ?)
=>
      (printout t ?name " is a " ?type crlf))
```

58

Example: Agriculture

Problem Description: Write a CLIPS program that is capable of recommending a herbicide and the appropriate application rate for that herbicide for a given field situation. Information concerning the herbicides and their application guidelines are contained in the following table. The crops are corn (C) and soybeans (S) and the weeds are grass (G) and broadleaf (B).

			Organic Matter		
Herbicide	**Weed**	**Crop**	**< 2%**	**2-4%**	**> 4%**
Sencor	**B**	**C or S**	**Do Not Use**	**3/4 pt/ac**	**3/4 pt/ac**
Lasso	**B or G**	**C or S**	**2 qt/ac**	**1 qt/ac**	**0.5 qt/ac**
Bicep	**B or G**	**C**	**1.5 qt/ac**	**2.5 qt/ac**	**3 qt/ac**

59

```
(defrule Sencor-1
  (weed B)
  (crop C | S)
  (organic-matter 1)
=>
  (printout t crlf "Do not use Sencor!" crlf))

(defrule Sencor-2
  (weed B)
  (crop C | S)
  (organic-matter 2 | 3)
=>
  (printout t crlf "Use 3/4 pt/ac of Sencor" crlf))

(defrule Lasso-1
  (weed B | G)
  (crop C | S)
  (organic-matter 1)
=>
  (printout t crlf "Use 2 pt/ac of Lasso" crlf))
```

60

```
(defrule Lasso-2
  (weed B | G)
  (crop C | S)
  (organic-matter 2)
=>
  (printout t crlf "Use 1 pt/ac of Lasso" crlf))

(defrule Lasso-3
  (weed B | G)
  (crop C | S)
  (organic-matter 3)
=>
  (printout t crlf "Use 0.5 pt/ac of Lasso" crlf))

(defrule Bicep-1
  (weed B | G)
  (crop C)
  (organic-matter 1)
=>
  (printout t crlf "Use 1.5 pt/ac of Bicep" crlf))
```

61

```
(defrule Bicep-2
  (weed B | G)
  (crop C)
  (organic-matter 2)
=>
  (printout t crlf "Use 2.5 pt/ac of Bicep" crlf))

(defrule Bicep-3
  (weed B | G)
  (crop C)
  (organic-matter 3)
=>
  (printout t crlf "Use 3 pt/ac of Bicep" crlf))
```

62

```
(defrule input
   (initial-fact)
   =>
   (printout t crlf "What is the crop? (C: corn, S: soybean) ")
   (assert (crop =(read)))  ;this line reads what the user types
   (printout t crlf "What is the weed problem? (B: broadleaf, G: grass) ")
   (assert (weed =(read)))
   (printout t crlf "What is the percentage of organic matter content?
      (1: <2%, 2: 2-4%, 3: > 4%) ")
   (assert (organic-matter =(read)))
   (printout t crlf "RECOMMENDATIONS:" crlf))
```

63

CLIPS SESSION:

```
CLIPS> (load "herbicide.clp")
Defining defrule: Sencor-1 +j+j+j
Defining defrule: Sencor-2 =j=j+j
Defining defrule: Lasso-1 +j+j+j
Defining defrule: Lasso-2 =j=j+j
Defining defrule: Lasso-3 =j=j+j
Defining defrule: Bicep-1 =j+j+j
Defining defrule: Bicep-2 =j=j+j
Defining defrule: Bicep-3 =j=j+j
Defining defrule: input +j
TRUE

CLIPS> (reset)
CLIPS> (run)
What is the crop? (C: corn, S: soybean) C
What is the weed problem? (B: broadleaf, G: grass) B
What is the percentage of organic matter content? (1: <2%, 2: 2-4%, 3: > 4%) 2

RECOMMENDATIONS:
Use 3/4 pt/ac of Sencor
Use 1 pt/ac of Lasso
Use 2.5 pt/ac of Bicep
CLIPS> (dribble-off)
```

64

Math Expressions

- CLIPS maths expressions are written in the prefix format, just like in LISP or Scheme:
  `(+ 2 3)` evaluates to `5`
  - Operators are: '+' addition, '-' subtraction, '*' multiplication, ' /' division, '**' exponentiation
    `(+ 2 (* 3 4))` evaluates to `14`
    `(* (+ 2 3) 4)` evaluates to `20`
    (evaluation is from the inside out)

65

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

66

Pattern Logical OR (1)

Suppose the following three rules are given.

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

  (defrule shut-off-electricity-2
      (disaster C)
  =>
      (printout t “Shut off the electricity” crlf))

  (defrule shut-off-electricity-3
      (sprinkler-systems active)
  =>
      (printout t “Shut off the electricity” crlf))
```

67

Pattern Logical OR (2)

- The three previous rules can be replaced by one rule by using OR:

```
(defrule shut-off-electricity
   (or  (emergency flood)
        (disaster C)
        (sprinkler-systems active))
  =>
   (printout t “Shut off the electricity” crlf))
```

68

Pattern Logical AND

- It is the **default** (implicit) operator.
- It requires that **all the patterns** of the LHS of the rule to be matched to facts in order to trigger the rule.

```
(defrule shut-off-electricity
     (and (emergency flood)
          (disaster C)
          (sprinkler-systems active))
  =>
     (printout t "Shut off the electricity" crlf))
```

69

Pattern Logical NOT

- The logical NOT can only be used to negate a single pattern:

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

70

Universal Conditional Elements

- EXISTS conditional element (∃)
- FORALL conditional element (∀)

71

EXAMPLE-1:
(no conditions)

```
(deftemplate person
        (slot name)  (slot age)  (slot height))

(deffacts people
   (person (name Andrew) (age 24) (height 1.85))
   (person (name Cyril) (age 23) (height 1.70))
   (person (name James) (age 20) (height 1.72))
   (person (name Albert) (age 19) (height 1.80)))

(defrule check-each-person
  (person (name ?n))
=>
  (printout t "All persons have a name data" crlf))
```

This program generates the following output:

```
CLIPS> (reset)
CLIPS> (run)
All persons have a name data
All persons have a name data
All persons have a name data
All persons have a name data
```

72

EXAMPLE-2:
(includes *forall* condition)

```
(defrule check-each-person
  (forall   (person (name ?n))
            (person (name ?n))
  )
=>
  (printout t "All persons have a name data" crlf))
```

This program generates the following output:
CLIPS> (reset)
CLIPS> (run)
All persons have a name data

73

EXAMPLE-3:
(includes *exists* condition)

```
(defrule any-people
  (exists   (person (name ?n)))
=>
  (printout t "There is at least one person with a name" crlf)
)
```

This program generates the following output:
CLIPS> (reset)
CLIPS> (run)
There is at least one person with a name

74

Test Control Pattern

- Control pattern can be used in the LHS to test a condition.
- General syntax:
  `(test <predicate-function>)`
- Example:
  `(test (> ?size 1))`

75

EXAMPLE

```
(deffacts people
   (person (name Andrew) (age 24) (height 1.85))
   (person (name Jane) (age 23) (height 1.70)))


(defrule display-tall-persons
  (person (name ?esm) (height ?gad))
  (test (> ?gad 1.80))
=>
  (printout t ?esm " height is " ?gad crlf)
)
```

76

Predicate Functions

- The predicate functions are used to return a value of either true or false - `and, not, or`
- `eq` equal, `neq` not equal

  `(eq <any-value> <any-value>)`
- = equal, != not equal, >= greater than or equal,

  > greater than, <= less than or equal, < less than These are used for numeric values.

  `(<= <numeric-value><numeric-value>)`
- These are used to test the type of a field: `numberp, stringp, wordp, integerp, evenp, oddp`

77

IF and WHILE Functions

```
(deffacts basla
  (phase   check-continue))

(defrule continue-check
  ?phase-adr <- (phase   check-continue)
=>
  (retract ?phase-adr)
  (printout t "Continue (yes/no) ? " )
  (bind ?answer (read))

  (while (and (neq ?answer yes) (neq ?answer no)) do
    (printout t “Invalid answer, continue (yes/no) ? " )
    (bind ?answer (read)))

  (if (eq ?answer yes)
     then (assert (phase   continue))
     else (halt))
)
```

78

SWITCH Function

```
(defrule start
=>
 (printout t crlf crlf crlf)
  (printout t "        *********************************" crlf)
  (printout t "        *       List of your choices     *" crlf)
  (printout t "        *            Choose one          *" crlf)
  (printout t "        *********************************" crlf)
  (printout t "        *  1.Your First Choice           *" crlf)
  (printout t "        *  2.Your Second Choice          *" crlf)
  (printout t "        *  3.Your Third Choice           *" crlf)
  (printout t "        *  4.Stop Rules Execution        *" crlf)
  (printout t "        *********************************" crlf)
 (printout t "             Your choice? ")
 (bind ?menu (read))
 (switch ?menu
   (case 1 then
        (assert (choice one)))
   (case 2 then
        (assert (choice two)))
   (case 3 then
        (assert (choice three)))
   (case 4 then
        (printout t crlf "stoping..." crlf)
        (halt))
   (default then
        (printout t crlf "What did you choose?" crlf)
        (reset)
        (run))))
```

```
(defrule one
 (choice one)
=>
 (printout t crlf "This is your first choice:..." crlf))

(defrule two
 (choice two)
=>
 (printout t crlf "This is your second choice:..." crlf))

(defrule three
 (choice three)
=>
 (printout t crlf "This is your third chocie:..." crlf))
```

79

LOOP-FOR-COUNT Function

```
(defrule factorial
=>
 (printout t crlf crlf "Enter a number: ")
 (bind ?N (read))
 (bind ?temp 1)
 (loop-for-count (?i 1 ?N) do
   (bind ?temp (* ?temp ?i))
 )
 (printout t "Factorial=" ?temp crlf)

)
```

80

Implicit Looping Example

```
(deffacts initial-information
   (rectangle 10 6)
   (rectangle 7 5)
   (rectangle 6 8)
   (rectangle 2 5)
   (rectangle 9 4)
   (sum 0)
   (count 0))

(defrule sum-rectangles
   (declare (salience 30))
   (rectangle ?height ?width)
=>
   (assert (add-to-sum = (* ?height ?width))))
```

81

```
(defrule sum-areas
   (declare (salience 20))
   ?sum-adr <- (sum ?total)
   ?new-area-adr <- (add-to-sum ?area)
   ?count-adr <- (count ?counter)
=>
   (retract ?sum-adr  ?new-area-adr ?count-adr)
   (assert (sum = (+ ?total ?area)))
   (assert (count = (+ ?counter 1))))

(defrule average
   (declare (salience 10))
   ?sum-adr <- (sum ?total)
   ?count-adr <- (count ?counter)
=>
 (printout t crlf "Here is the average area" (/ ?total ?counter) crlf crlf))
```

82

Avoiding Infinite Loop

- You can get into an infinite loop if you are not careful enough.

```
(deffacts myfact
     (loop-fact))

(defrule simple-loop
     ?old-fact  <-  (loop-fact)
 =>
     (printout t "Looping!" crlf)
     (retract ?oldfact)
     (assert (loop-fact)))
```

- Use Control-C (or another interrupt command) to get break out of the loop.

83

Example: Finding largest number

```
(deffacts numbers
  (number   56)
  (number   -32)
  (number   7)
  (number   96)
  (number   24))

(defrule  find-largest-number
  (number  ?x)
  (not   (number   ?y &: (>   ?y   ?x)))
  =>
  (printout   t   "Largest number is "   ?x   crlf))
```

84

Defining Functions

Syntax: (deffunction function-name (arg ... arg)
    action ... action)

Example:

```
(deffunction hypotenuse (?a ?b)
    (sqrt (+ (* ?a ?a) (* ?b ?b))))

(defrule hesapla
   =>
  (printout t “hypotenuse for 8 and 5: “ (hypotenuse 8 5) crlf))
```

85

EXAMPLE:CALCULATING FACTORIAL

```
(deffunction factorial(?X)
  (bind ?temp 1)
  (loop-for-count (?i 1 ?X) do
    (bind ?temp (* ?temp ?i))
  )
  (return ?temp)
)

(defrule example
=>
 (printout t crlf crlf "N factorial: ")
 (bind ?N (read))
 (printout t "Factorial=" (factorial ?N) crlf)
)
```

86

Standard I/O Functions

- To print to screen: (printout t …)
  For the new line use: crlf
- To read from keyboard use:(read)
- **Example-1:**

```
(defrule to-start
   (phase choose-name)
 =>
   (printout t “Enter your name” crlf)
   (assert (your-name =(read))))   ; ’=’ is optional
```

- **Example-2:**

```
(defrule to-start
 =>
   (printout t “Enter something: ”)
   (bind ?something (read))
   (printout t “You have entered ” ?something crlf))
```

87

A more advanced example:

```
(defrule continue-check
      ?phase <- (phase check-continue)
=>
      (retract ?phase)
      (printout t “Do you want to continue?” crlf)
      (bind ?answer (read))
      (if      (or (eq ?answer yes) (eq ?answer y))
       then  (assert (phase continue))
       else   (halt))
)
```

88

- 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>)`
  - e.g. (load "B:\\clips\\example.clp")

89

Input Example-1

```
(deffacts initial-phase
   (phase choose-player))

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

(defrule good-player-choice
   ?phase  <-  (phase choose-player)
   ?choice  <-  (player-select  ?player&c|h)
   =>
   (retract ?phase ?choice)
   (assert (player-move  ?player)) )
```

90

```
(defrule bad-player-choice
   ?phase  <-  (phase choose-player)
   ?choice  <-  (player-select  ?player&~c&~h)
   =>
   (retract ?phase ?choice)
   (assert (phase choose-player))
   (printout t “Choose c or h.” crlf) )
```

- This is useful for error checking.

Input Example-2

- (read) - is used to input a single field.
- It can also be used with the bind command.

```
(defrule continue-check
  ?phase <- (phase check-continue)
 =>
   (retract ?phase)
  (printout t “Do you want to continue?” crlf)
  (bind ?answer (read))
  (if (or (eq ?answer yes) (eq ?answer y))
      then (assert (phase continue))
      else (halt) )  )
```

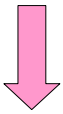
File I/O Functions

```
(defrule open-files
=>
   (open "indata.dat" indata "r")
   (open "outdata.txt" outdata "w")
   (assert (phase read-from-file)))

(defrule read-data
   ?phase-adr <- (phase read-from-file)
=>
   (retract ?phase-adr)
   (bind ?name (read indata))
   (if (eq ?name EOF)
      then (close indata)
           (close outdata)
           (halt))
   (bind ?age (read indata))
   (if (> ?age 50)
      then (printout t ?name crlf)
           (printout outdata ?name "  " ?age crlf))
   (assert (phase read-from-file)))
```

indata.dat file

"Jamie Lambert"	74
"Norm Carve"	83
"Fred Petry"	45
"Bill White"	80



outdata.txt file

```
Jamie Lambert  74
Norm Carver  83
Bill White 80
```

93

BUILT-IN FUNCTIONS

- String Functions
- Multi-field Functions
- Math Functions
- Utility Functions

94

Examples of String Functions

• **CLIPS> (str-assert "student 12345 \"Lisa Gandy\" 1985")**
This command asserts the following fact:
(student 12345 "Lisa Gandy" 1985)

• **(str-cat "index-" 9 ".txt")**
This command produces following output:
indexa-9.txt

• **(str-index "red" "blueredgreen")**
This command produces following output as position:
5

• **(sub-string 2 5 "Shahram")**
This command produces following output:
hahr

95

Examples of Multi-field Functions

• **CLIPS> (mv-append a b c "red")**
This command constructs the following multifield value:
(a b c "red")

• **(mv-delete 3 (mv-append a b c "red")**
This command produces the following multifield value:
(a b "red")

• **(length (mv-append a b c "red"))**
This command returns the number of fields in a multifield value:
4

•**(nth 2 (mv-append a b c "red"))**
This command returns the specified field:
b

96

• **(member c (mv-append a b c "red")**
This command returns the index of specified filed value:
3

• **(subset (mv-append a b) (mv-append b a c))**
This command checks if first set is a subset of second set:
TRUE

• **(subset (mv-append a d c) (mv-append c a))**
This command returns zero:
FALSE

97

Examples of Math Functions

• **CLIPS> (pi)**
3.14159274

• **(cos (pi))**
-1

• **(cos (rad-deg(90)))**
1

• **(min 3 1 8 7)**
1

•**(max 3 1 8 7)**
8

• **(sqrt 16)**
4

• **(trunc 15.3)**
15

• **(abs -25)**
25

• **(mod 17 2)**
1

•**(** 3 2)**
9

98

Examples of Utility Functions

•**system:**

```
(defrule  list-the-directory
   =>
   (system   "ls "   "*.txt"))
```

When this rule is executed, the specified UNIX file names appear on screen:

•**batch:**

When we enter the following command, all commands in batch file are executed.

```
CLIPS>  (batch  "mybatch.bat")
  (load "pro1.clp")
  (load "pro2.clp")
  (reset)
  (run)
```

99

gensym Function

Symbol generation commands generate a unique word every time called:

Example-1:

```
 CLIPS> (gensym)
gen1

• CLIPS> (gensym)
gen2

• CLIPS> (setgen  15)
• CLIPS> (gensym)
gen15

• CLIPS> (gensym)
gen16
```

Example-2:

```
CLIPS>(deffacts list
   (name John (gensym))
   (name Ken (gensym))
   (name Bob (gensym)))

CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (name John gen1)
f-2     (name Ken gen2)
f-3     (name Bob gen3)
For a total of 4 facts.
```

100

Agenda

- If the pattern(s) in the LHS of the rule match asserted facts, the rule is activated and put on the agenda.
- Rules are ordered on the agenda according to their **salience** (priority).
- When the agenda is empty the program stops.
- **Refraction**: each rule is fired only once for a specific set of facts => use `(refresh)`

101

EXAMPLE:

```
(deffacts myFact
  (name John)
  (name Ken)
  (school SIUC)
  (school SIUE)
)

(defrule print-names
  (name ?n)
=>
  (printout t ?n crlf))

(defrule print-schools
  (school ?s)
=>
  (printout t ?s crlf))
```

102

CLIPS SESSION:

```
CLIPS> (load "orn.clp")
Defining deffacts: factler
Defining defrule: print-names +j
Defining defrule: print-schools +j
TRUE
CLIPS> (facts)
CLIPS> (agenda)
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (name John)
f-2     (name Ken)
f-3     (school SIUC)
f-4     (school SIUE)
For a total of 5 facts.
CLIPS> (agenda)
0      print-schools: f-4
0      print-schools: f-3
0      print-names: f-2
0      print-names: f-1
For a total of 4 activations.
```

```
CLIPS> (run)
SIUE
SIUC
Ken
John
CLIPS> (facts)
f-0     (initial-fact)
f-1     (name John)
f-2     (name Ken)
f-3     (school SIUC)
f-4     (school SIUE)
For a total of 5 facts.
CLIPS> (agenda)
CLIPS> (assert (name Selim))
<Fact-5>
CLIPS> (facts)
f-0     (initial-fact)
f-1     (name John)
f-2     (name Ken)
f-3     (school SIUC)
f-4     (school SIUE)
f-5     (name Selim)
For a total of 6 facts.
```

```
CLIPS> (agenda)
0      print-names: f-5
For a total of 1 activation.
CLIPS> (run)
Selim
CLIPS> (agenda)
```

103

Salience

- Salience is used to determine the order of execution of rules (-10,000 to 10000).
- Normally the agenda acts like a stack.
- The most recent activation placed on the agenda is the first rule to fire.
- Salience allows more important rules to stay at the top of the agenda regardless of when they were added.
- If you do not explicitly say, CLIPS will assume the rule has a salience of 0.

104

Example-1 (No salience)

```
(defrule kural-A
  =>
 (printout t "kural-A is fired" crlf))

(defrule kural-B
  =>
  (printout t "kural-B is fired" crlf))

(defrule kural-C
  =>
  (printout t "kural-C is fired" crlf))
```

In this program, rules have equal priorities. Therefore the following output is generated:

```
CLIPS> (reset)
CLIPS> (run)

kural-A is fired
kural-B is fired
kural-C is fired
```

105

Example-2 (No salience)

```
(defrule kural-A
   (myfact A)
  =>
 (printout t "kural-A is fired" crlf))

(defrule kural-B
   (myfact B)
 =>
  (printout t "kural-B is fired" crlf))

(defrule kural-C
   (myfact C)
  =>
  (printout t "kural-C is fired" crlf))
```

Now suppose we assert the following three facts in this order:

```
CLIPS> (reset)
CLIPS> (assert (myfact A))
CLIPS> (assert (myfact B))
CLIPS> (assert (myfact C))
CLIPS> (run)
```

The following output is generated. The last asserted rule is fired first!

```
kural-C is fired
kural-B is fired
kural-A is fired
```

106

Example-3 (No salience)

```
(deffacts gercekler
  (myfact A)
  (myfact B)
  (myfact C))

(defrule kural-A
   (myfact A)
 =>
 (printout t "kural-A is fired" crlf))

(defrule kural-B
   (myfact B)
 =>
  (printout t "kural-B is fired" crlf))

(defrule kural-C
   (myfact C)
 =>
  (printout t "kural-C is fired" crlf))
```

This example is a modified version of Example-2. Instead of asserting facts during run-time, they are defined as initial facts in this program.

CLIPS> (reset)
CLIPS> (run)

Output is the same as Example-2:

kural-C is fired
kural-B is fired
kural-A is fired

107

Example-4 (salience)

```
(defrule kural-A
   (declare (salience 20))
=>
   (printout t "kural-A is fired" crlf))

(defrule kural-B
   (declare (salience 30))
=>
   (printout t "kural-B is fired" crlf))

(defrule kural-C
   (declare (salience 10))
=>
   (printout t "kural-C is fired" crlf))
```

In this program the following output is generated: Saliences always enforce the firing order between rules.

CLIPS> (reset)
CLIPS> (run)

kural-B is fired
kural-A is fired
kural-C is fired

108

Example-5 (salience)

```
(defrule kural-A
   (declare (salience 20))
   (myfact A)
  =>
   (printout t "kural-A is fired" crlf))

(defrule kural-B
   (declare (salience 30))
   (myfact B)
 =>
   (printout t "kural-B is fired" crlf))

(defrule kural-C
   (declare (salience 10))
  (myfact C)
  =>
   (printout t "kural-C is fired" crlf))
```

Now suppose we assert the following three facts in this order:

```
CLIPS> (reset)
CLIPS> (assert (myfact A))
CLIPS> (assert (myfact B))
CLIPS> (assert (myfact C))
CLIPS> (run)
```

In this case, the order of assertion commands does not effect the output, because saliences enforce the firing order between rules.

```
kural-B is fired
kural-A is fired
kural-C is fired
```

109

Conflict Resolution Strategies

- Recency
  - Rules which use more recent data are preferred.
- Specificity
  - Rules with more conditions are preferred to more general rules that are easier to satisfy.
- Refractoriness
  - A rule should not be allowed to fire more than once for the same data.
  - Prevents loops

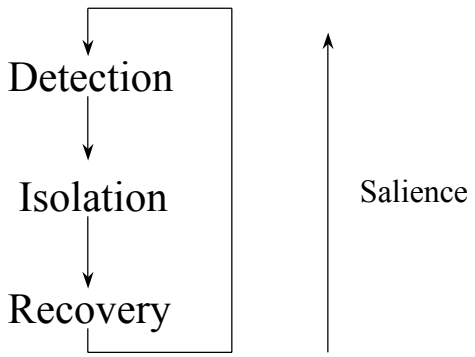
110

Conflict Resolution in CLIPS

- First, CLIPS uses salience to sort the rules. Then it uses the other strategies to sort rules with equal salience.
- Rule-based expert system is opportunistic
- Some procedural aspect normal; not more than 7 level
  - E.g., human's move vs. computer's move

111

Salience Mechanism



Different phases for fault Detection, Isolation, and Recovery Program

112

Control Rules Example

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

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

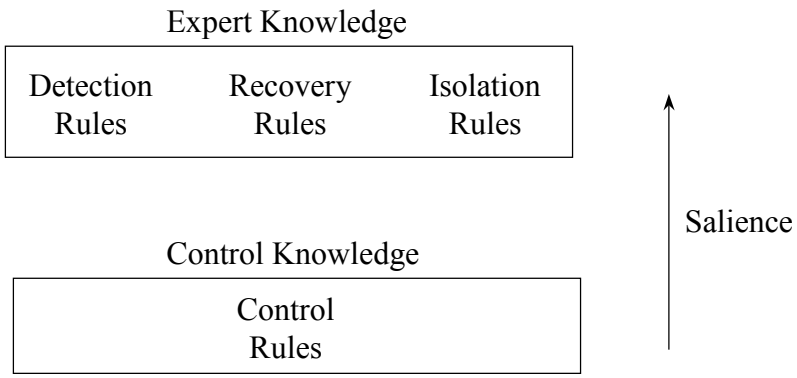
113

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

(defrule find-fault-location-and-recovery
        (phase recovery)
        (recovery-solution switch-device ?replacement on)
   =>
        (printout t “Switch device” ?replacement “on” crlf))
```

114

Separation of Expert/Domain Knowledge & Control Knowledge



115

The previous control rules can be written in a more general form

```
(deffacts control-information
   (phase detection)
   (phase-after detection isolation)
   (phase-after isolation recovery)
   (phase-after recovery detection))

(defrule change-phase
   (declare (salience -10))
   ?phase <- (phase ?current-phase)
   (phase-after ?current-phase ?next-phase)
=>
   (retract ?phase)
   (assert (phase ?next-phase)))
```

116

It can also be written as a sequence of phases to be cycled through

```
(deffacts control-information
   (phase detection)
   (phase-sequence isolation recovery detection))

(defrule change-phase
   (declare (salience -10))
   ?current-adr <- (phase ?current-phase)
   ?phase-adr <- (phase-sequence ?next-phase $?other-phases)
=>
   (retract ?current-adr ?phase-adr)
   (assert (phase ?next-phase))
   (assert (phase-sequence $?other-phases ?next-phase)))
```

117

Debugging

- Rules and facts can be watched for debugging purposes:
  - `(watch rules)`
  - `(watch facts)`
- Make save output to a file:
  - To start logging: `(dribble-on "output.log")`
  - To stop: `(dribble-off)`

118

Commands for Debugging

- (watch {facts, rules, activations, all})
- (unwatch {facts, rules, activations, all})
- (matches <rule-name>)
- (set-break <rule-name>)
- (show-breaks)
- (remove-break [<rule-name>])
- (dribble-on <“file-name”>)
- (dribble-off)

119

Limitations of CLIPS

- single level rule sets
  - you can not arrange rule sets in a hierarchy, embedding one rule set inside another, etc
- 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

120

Alternatives to CLIPS

- JESS (Java Expert System Shell)
  - has same syntax as CLIPS
  - can be invoked from Java programs
  - COOL is replaced by Java classes
- Eclipse
  - has same syntax as CLIPS
  - supports goal-driven (i.e., backwards) reasoning
  - can be integrated with C++ and dBase
- NEXPERT OBJECT
  - rule and object-based system
  - has a script language for designing user front-end
  - written in C, runs on many platforms

121