

Chapter 7: Introduction to CLIPS

Expert Systems: Principles and
Programming, Fourth Edition

Objectives

- Learn what type of language CLIPS is
- Study the notation (syntax) used by CLIPS
- Learn the meaning of a field and what types exist
- Learn how to launch and exit from CLIPS
- Learn how to represent, add, remove, modify, and duplicate in CLIPS

2

Objectives

- Learn how to debug programs using the watch command
- Learn how to use the deffacts construct to define a group of facts
- Learn how to use the agenda command and execute CLIPS programs
- Learn about commands that can manipulate constructs

3

Objectives

- Learn how to use the printout command
- Learn how to use multiple rules
- Learn how to use the set-break command
- Learn how to use the load and save constructs
- Learn how to use variables, single and multifield wildcards, and comment constructs

4

What is CLIPS?

- CLIPS is a multiparadigm programming language that provides support for:
 - Rule-based
 - Object-oriented
 - Procedural programming
- Syntactically, CLIPS resembles:
 - Eclipse
 - CLIPS/R2
 - JESS

5

Other CLIPS Characteristics

- CLIPS supports only forward-chaining rules.
- The OOP capabilities of CLIPS are referred to as CLIPS Object-Oriented Language (COOL).
- The procedural language capabilities of CLIPS are similar to languages such as:
 - C
 - Ada
 - Pascal
 - Lisp

6

CLIPS Characteristics

- CLIPS is an acronym for **C** Language **I**ntegrated **P**roduction **S**ystem.
- CLIPS was designed using the C language at the NASA/Johnson Space Center.
- CLIPS is portable – PC → CRAY.

7

CLIPS Notation

- Symbols other than those delimited by `<>`, `[]`, or `{ }` should be typed exactly as shown.
 - E.g.: (example) should be entered as shown
- `[]` mean the contents are optional and `<>` mean that a replacement is to be made.
 - E.g.: (example <integer>) → (example 10)
- `*` following a description means that the description can be replaced by zero or more occurrences of the specified value.
 - E.g.: (example <integer>*)

8

CLIPS Notation

- Descriptions followed by + mean that one or more values specified by description should be used in place of the syntax description.
 - E.g.: (example <integer>+) → (example 10 5)
- A vertical bar | indicates a choice among one or more of the items separated by the bars.
 - E.g.: all | none | some

9

Fields

- To build a knowledge base, CLIPS must read input from keyboard / files to execute commands and load programs.
- During the process of reading, CLIPS groups symbols together into tokens – groups of characters that have special meanings.
- Fields are a special group of tokens representing certain data units. There are 8 types of fields or primitive data types.

10

Numeric Fields

- The floats and integers make up the numeric fields – simply numbers.
- Integers have only a sign and digits.
 - E.g. 1 +3 -15
- Floats have a decimal and possibly “e” for scientific notation.
 - E.g. 0.7 23.7 9.4e+3

11

Symbol Fields

- Symbols begin with printable ASCII characters followed by zero or more characters, followed by a delimiter.
 - E.g. Space-Station
fire
notify-fire-department
G
- CLIPS is case sensitive.

12

String Fields

- Strings must begin and end with double quotation marks.
 - E.g. "John Q. Public "
- Spaces within the string are significant.
 - E.g. "spaces" ≠ "space "
- The actual delimiter symbols can be included in a string by preceding the character with a backslash.
 - E.g. "\"fine\"" → "fine"

13

Address Fields

- External addresses represent the address of an external data structure returned by a user-defined function.
- Fact address fields are used to refer to a specific fact.
- Instance name / address field – instances are complex data representation and are created from classes. Instances can be referred to either by names or by addresses.

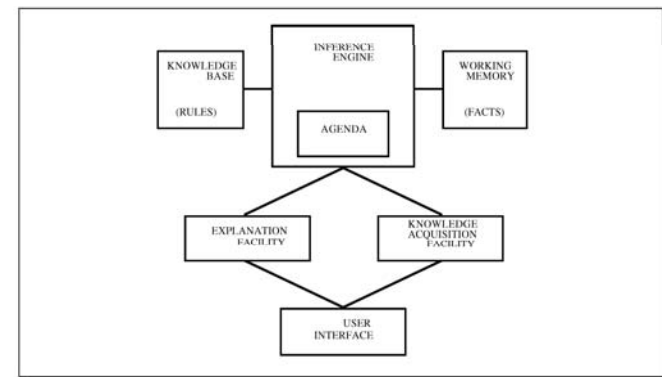
14

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.`

15

Figure 1.6 Structure of a Rule-Based Expert System




16

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**.
- Fact is the smallest unit of knowledge.
- Facts consist of:
 - Relation name (symbolic field)
 - Zero or more slots with associated values
- The order of slots is irrelevant

17

Example Fact in CLIPS



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

18

Deftemplate

- Before facts can be constructed, CLIPS must be informed of the list of valid slots for a given relation name.
- A deftemplate construct is used to describe groups of facts sharing the same relation name and contain common information.
- The deftemplate construct is analogous to “typedef struct” in C.

19

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))
```

20

Deftemplate vs. typedef struct

- The following C code first defines a template of a person structure, and then two person structure variables:

```
typedef struct {
    char forename[20];
    char surname[20];
    float age;
    int childcount;
} person;

person jimmy, flossie;
```

21

Deftemplate vs. Ordered Facts

- Facts with a relation name defined using deftemplate are called *deftemplate facts*.
- Whenever possible deftemplate facts should be used because the slot names make the facts more readable and easier to work with
- Facts with a relation name that does not have a corresponding deftemplate are called *ordered facts* – having a single implied multifield slot for storing all the values of the relation name.
 - E.g.: (food-groups meat dairy bread fruit)

22

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")))
```

23

Displaying Facts

- CLIPS> (facts) ←



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

For a total of 1 fact.

24

Removing Facts

- Just as facts can be added, they can also be removed.
- Removing facts results in gaps in the fact identifier list.
- To remove a fact with index 2 (f-2):

```
CLIPS> (retract 2) ↵
```

25

Modifying Facts

- 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> )
```

26

Modifying Facts

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.

27

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.

```
CLIPS > (duplicate 4 (name "Jennifer Summers")) ↵
```

28

Watch Command

- The *watch* command is useful for debugging purposes.
- The syntax: (watch <watch-item>)
where <watch-item> can be:
 facts, activations, rules, statistics, etc.
- If facts are “watched”, CLIPS will automatically print a message indicating an update has been made to the fact list whenever facts are asserted, retracted, modified, duplicated, etc :

```
CLIPS > (watch facts) ↵
```

29

Deffacts Construct

- The *deffacts* construct can be used to assert a group of facts, especially those that are known to be true before running the program.
- Groups of facts representing knowledge can be defined as follows:

```
(deffacts <deffacts name> [<optional comment>  
    <facts> * )
```

- The *reset* command is used to remove all existing facts and assert the facts in a deffacts statement.

30

Deffacts Construct Example

```
CLIPS > (deffacts people “some people we know”  
    person (name “Jon Q. Public”) (age 24)  
        (eye-color blue) (hair-color black))  
    person (name “Jack S. Public”) age 24)  
        (eye-color blue) (hair-color black))) ↵
```

```
CLIPS > (reset) ↵
```

```
CLIPS > (facts) ↵
```

```
f-0   (initial-fact)  
f-1   (person (name “Jon Q. Public”) (age 24)  
        (eye-color blue) (hair-color black))  
f-2   (person (name “Jack S. Public”) (age 24)  
        (eye-color blue) (hair-color black))
```

31

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
```

32

Rule Components

- First, we need to create the deftemplate for the types of facts related to the rule:
`(deftemplate emergency (slot type))`
 - Type would be fire, flood, power outage, etc.
- Similarly, we must create the deftemplate for the types of responses:
`(deftemplate response (slot action))`
 - action would be “activate the sprinkler” or other responses to be taken.

33

Rule Components

- The rule would be shown as follows:
CLIPS >
`(defrule fire-emergency “An example rule”
 (emergency (type fire))
 =>
 (assert (response
 (action activate-sprinkler-system))))`
- Before entering any rule construct, type (clear) command to remove deftemplates and deffacts created from the previous section.

34

Rule Components

- General format of a rule:
`(defrule <rule name> [<comment>]
 <patterns>* ; left-hand side of rule
 =>
 <actions>*) ; right-hand side of rule`

35

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 zero or more conditional elements – pattern CEs
- Each pattern consists of 1+ constraints intended to match the fields of the deftemplate fact
- CLIPS tries to match the patterns of the rules against the facts in the fact list.

36

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.
- The last part of the rule is the list of **actions** that will execute when the rule **fires**.
- The term fire means that CLIPS executes the actions of a rule from the agenda.

37

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.

38

Execution

- When the program runs, the rule with the highest **saliency** on the agenda is fired.
- Saliency – priority of a rule (unspecified now)
- 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.

39

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 – o.w. the same rule will fire repeatedly if old facts are not removed.
- 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.

40

What is on the Agenda?

```
CLIPS > (reset) ↵
CLIPS > (assert ( emergency (type fire))) ↵
<Fact-1>
CLIPS > (agenda) ↵
0   fire-emergency: f-1
For a total of 1 activation.
CLIPS > (run) ↵
CLIPS > (agenda) ↵
CLIPS > (refresh fire-emergency) ↵
CLIPS > (agenda) ↵
0   fire-emergency: f-1
For a total of 1 activation.
```

41

Command for Manipulating Constructs

- The *list-defrules* command is used to display the current list of rules maintained by CLIPS.
- The *list-deftemplates* displays the current list of deftemplates.
- The *list-deffacts* command displays the current list of deffacts.
- The *ppdefrule*, *ppdeftemplate* and *ppdeffacts* commands display the text representations of a defrule, deftemplate, and a deffact, respectively.

42

Commands

- The *undefrule*, *undeftemplate*, and *undeffacts* commands are used to delete a defrule, a deftemplate, and a deffact, respectively.
- The *clear* command clears the CLIPS environment and adds the *initial-fact* deftemplate and deffact to the CLIPS environment.
- The *printout* command can also be used to print information.

```
(defrule fire-emergency
  (emergency (type fire) =>
  (printout t "Activate the sprinkler system" crlf))
```

43

Other Commands

- *Set-break* – allows execution to be halted before any rule from a specified group of rules is fired.
CLIPS > (set-break fire-emergency)
- *Load* – allows loading of rules from an external file.
CLIPS > (load "e:fire.clp")
- *Save* – opposite of load, allows saving of constructs to disk
CLIPS > (save "e:fire.clp")

44

Commenting and Variables

- **Comments** – provide a good way to document programs to explain what constructs are doing.
 - Any text that begins with a semicolon and ends with a carriage return
- **Variables** – store values, syntax requires preceding with a question mark (?) and followed by a symbolic field
 - E.g. ?speed
 - ?color

45

Fact Addresses, Single-Field Wildcards, and Multifield Variables

- A variable can be used on the LHS of a rule to be bound to a slot value that can later be used on the LHS or RHS of the rule.

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

46

Fact Addresses, Single-Field Wildcards, and Multifield Variables

- In order to manipulate facts from the RHS of a rule, 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 "<-".

```
(defrule process-moved-information
  ?f1 <- (moved (name ?name) (address ?address))
  ?f2 <- (person (name ?name))
=>
  (retract ?f1)
  (modify ?f2 (address ?address)))
```

47

Fact Addresses, Single-Field Wildcards, and Multifield Variables

- **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.

```
(defrule print-social-security-numbers
  (print-ss-number-for ?last-name)
  (person (name ? ? ?last-name)
    (social-security-number ?ss-number))
=>
  (printout t ?ss-number crlf))
```

48

Fact Addresses, Single-Field Wildcards, and Multifield Variables

- **Multifield variables** (e.g. \$?name) and **wildcards** (\$?) allow matching against zero or more fields in a pattern.

```
(defrule print-social-security-numbers
  (print-ss-number-for ?last-name)
  (person (name $? ?last-name)
          (social-security-number ?ss-number))
=>
  (printout t ?ss-number crlf))
```

49

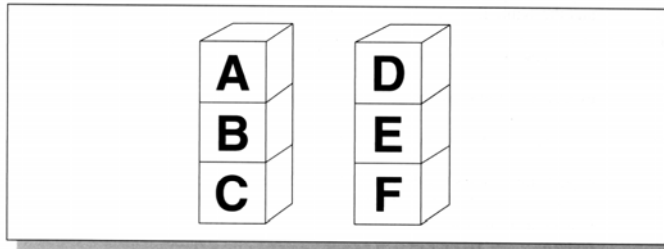
Blocks World Example

- A good example of planning
- The goal is to move one block on top of another with the minimum number of moves
 - 1: only one block may be moved at a time
 - 2: the goal must not have already been achieved.
- To move block x on top of block y, we need to move all blocks (if any) on top of x to the floor and all blocks (if any) on top of block y to the floor and then move block x on top of block y.

50

Blocks World Example

Figure 7.2 Blocks World Initial Configuration



51

Blocks World Example

RULE MOVE-DIRECTLY

IF The goal is to move block ?upper on top of block ?lower and
 block ?upper is the top block in its stack and
 block ?lower is the top block in its stack,
THEN Move block ?upper on top of block ?lower.

52

Blocks World Example

RULE CLEAR-UPPER-BLOCK

IF The goal is to move block ?x and
block ?x is not the top block in its stack and
block ?above is on top of block ?x,
THEN The goal is to move block ?above to the floor.

53

Blocks World Example

RULE CLEAR-LOWER-BLOCK

IF The goal is to move another block on top of
block ?x and
block ?x is not the top block in its stack and
block ?above is on top of block ?x,
THEN The goal is to move block ?above to the floor.

54

Blocks World Example

RULE MOVE-TO-FLOOR

IF The goal is to move block ?upper on top of
the floor and
block ?upper is the top block in its stack,
THEN Move block ?upper on top of the floor.

55

Blocks World Example

- Ordered facts are used to specify the initial stack configuration
(stack A B C)
(stack D E F)
- A deftemplate fact is used to describe the block-moving goal
(deftemplate goal (slot move) (slot on-top-of))
(goal (move C) (on-top-of E))

56

Blocks World Example

- Multifield wildcards and variables can be used to implement the rules in a easier fashion.

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

57

Blocks World Example

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

58

Blocks World Example

```
(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 " moved on top of "
            ?block2 "." crlf))
```

59

Blocks World Example

```
(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 " moved on top of floor." crlf))
```

Refer to the file *block.clp* on the CD-ROM packaged with the book for the complete listing of the Blocks World program.

60

Summary

- In this chapter, we looked at the fundamental components of CLIPS.
- Facts make up the first component of CLIPS, made up of fields – symbol, string, integer, float, fact address, etc. (8 types of fields)
- The deftemplate construct was used to assign slot names to specific fields of a fact.
- The deffacts construct was used to specify facts as initial knowledge.

61

Summary

- Rules make up the second component of a CLIPS system.
- Rules are divided into LHS – IF portion and the RHS – THEN portion.
- Rules can have multiple patterns and actions.
- The third component is the inference engine – rules having their patterns satisfied by facts produce an activation that is placed on the agenda.

62

Summary

- Refraction prevents old facts from activating rules.
- Variables are used to receive information from facts and constrain slot values when pattern matching on the LHS of a rule.
- Variables can store fact addresses of patterns on the LHS of a rule so that the fact bound to the pattern can be retracted on the RHS of the rule.
- We also looked at single-field wildcards and multifield variables.

63