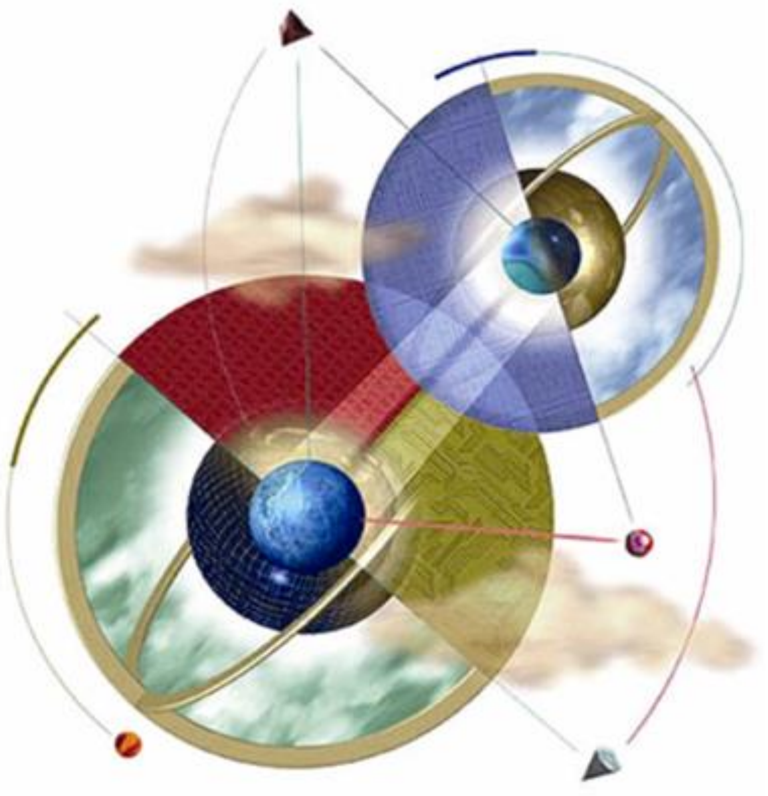


Chapter 7: Introduction to CLIPS

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 exit
- Learn how to launch and exit from CLIPS
- Learn how to represent, add, remove, modified, and duplicated in CLIPS

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

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

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

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

CLIPS Characteristics

- CLIPS is an acronym for C Language Integrated Production System.
- CLIPS was designed using the C language at the NASA/Johnson Space Center.
- CLIPS is portable – PC $\rightarrow$ CRAY.

CLIPS Notation

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

CLIPS Notation

- Descriptions followed by + mean that one or more values specified by description should be used in place of the syntax description.
- A vertical bar | indicates a choice among one or more of the items separated by the bars.

Fields

- To build a knowledge base, CLIPS must read input from keyboard / files to execute commands and load programs.
- During the execution process, CLIPS groups symbols together into tokens – groups of characters that have the same meaning.
- A field is a special type of token of which there are 8 types.

Numeric Fields

- The floats and integers make up the numeric fields – simply numbers.
- Integers have only a sign and digits.
- Floats have a decimal and possibly “e” for scientific notation.

Symbol Fields

- Symbols begin with printable ASCII characters followed by zero or more characters, followed by a delimiter.
- CLIPS is case sensitive.

String Fields

- Strings must begin and end with double quotation marks.
- Spaces w/in the string are significant.
- The actual delimiter symbols can be included in a string by preceding the character with a backslash.

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 similar to facts addresses but refer to the instance rather than a fact.

Entering / Exiting CLIPS

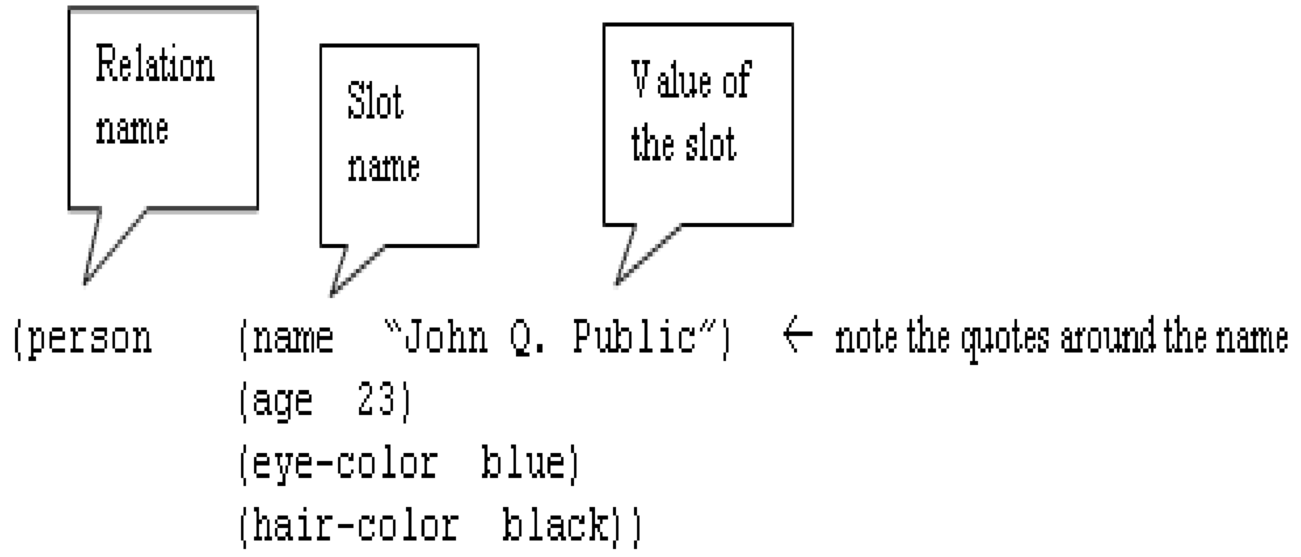
- The CLIPS prompt is: CLIPS>
- This is the type-level mode where commands can be entered.
- To exit CLIPS, one types: CLIPS> (exit) ↵
- CLIPS will accept input from the user / evaluate it / return an appropriate response:

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

Facts and CLIPS

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

Example Fact in CLIPS

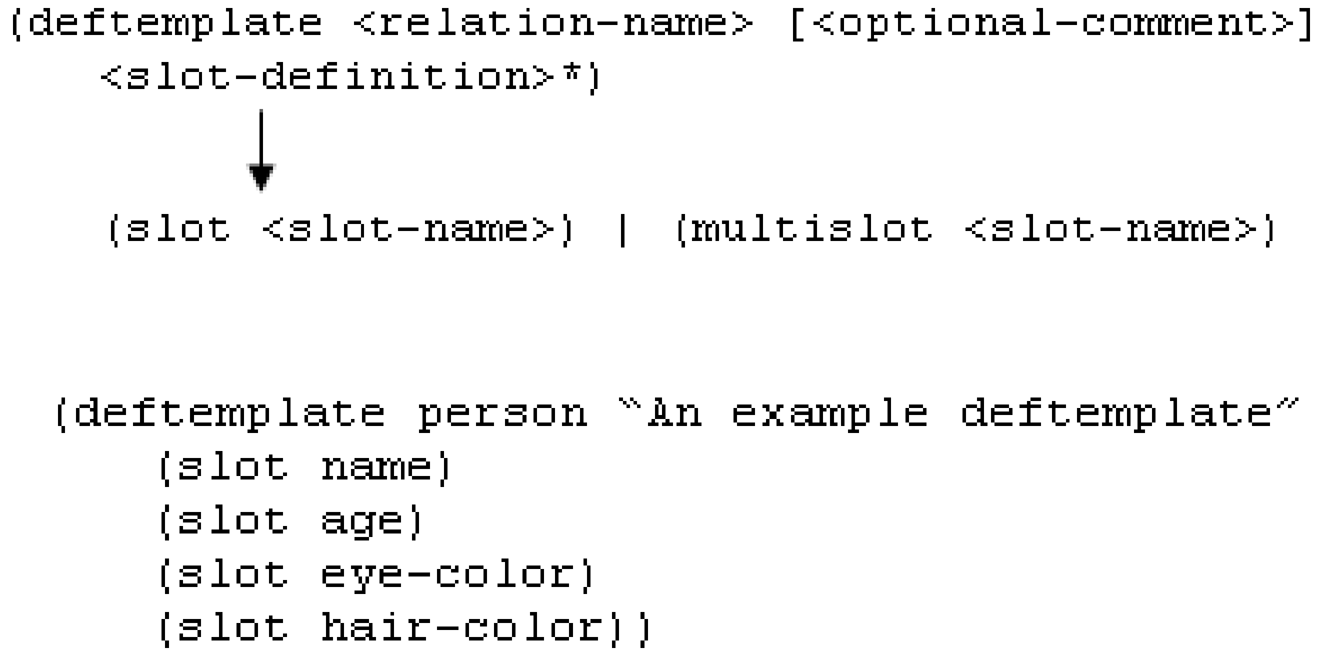


```
(person    (name   "John Q. Public")
           (age   23)
           (eye-color  blue)
           (hair-color  black))
```

Deftemplate

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

Deftemplate General Format



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

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

Deftemplate vs. Ordered Facts

- Facts with a relation name defined using deftemplate are called *deftemplate facts*.

- Facts with a relation name that does not have a corresponding deftemplate are called *ordered facts* – have a single implied multifield slot for storing all the values of the relation name.

Adding Facts

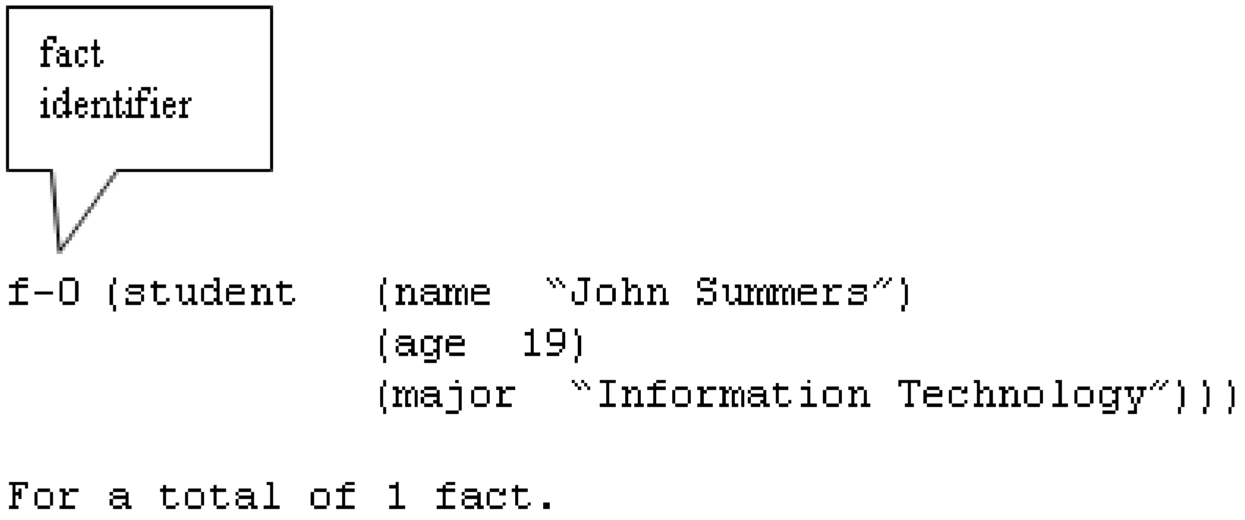
- CLIPS store all facts known to it in a fact list.
- To add a fact to the list, we use the *assert* command.

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

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

Displaying Facts

- CLIPS> (facts) ↵



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

For a total of 1 fact.
```

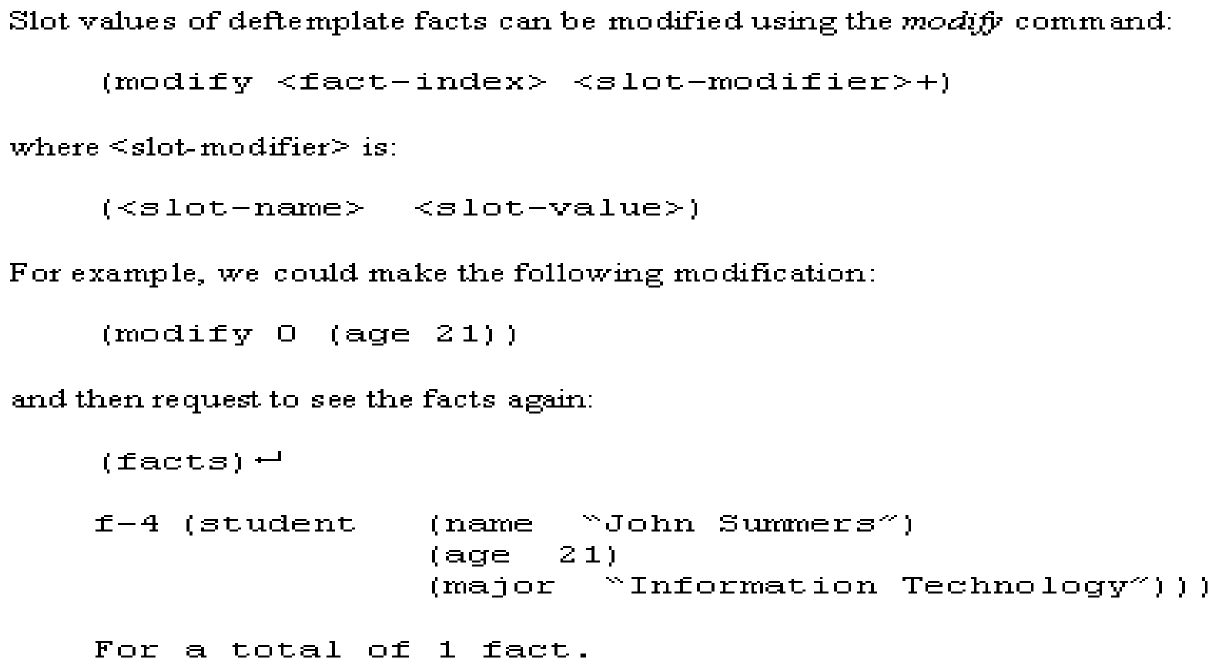
Removing Facts

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

```
CLIPS> (retract 2)↵
```

Modifying Facts

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



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

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

where <slot-modifier> is:

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

For example, we could make the following modification:

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

and then request to see the facts again:

```
(facts)↵

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

For a total of 1 fact.
```

Results of Modification

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

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

Watch Command

- The *watch* command is useful for debugging purposes.
- If facts are "watched", CLIPS will automatically print a message indicating an update has been made to the fact list whenever either of the following has been made:
  - Assertion
  - Retraction

Deffacts Construct

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

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

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

The Components of a Rule

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

```
IF the emergency is a fire
THEN the response is to activate the sprinkler system
```

Rule Components

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

  (deftemplate emergency (slot type))

  -- type would be fire, flood, etc.

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

  (deftemplate response (slot action))

  -- action would be "activate the sprinkler"

Rule Components

- The rule would be shown as follows:

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

Analysis of the Rule

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

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 Agenda and Execution

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

```
CLIPS>  (run  [<limit>])↵
```

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

Execution

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

What is on the Agenda?

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

```
CLIPS> (agenda)↵
```

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

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.

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 *initialfact-defacts* to the CLIPS environment.
- The *printout* command can also be used to print information.

Other Commands

- *Set-break* – allows execution to be halted before any rule from a specified group of rules is fired.
- *Load* – allows loading of rules from an external file.
- *Save* – opposite of load, allows saving of constructs to disk

Commenting and Variables

- Comments – provide a good way to document programs to explain what constructs are doing.
- Variables – store values, syntax requires preceding with a question mark (?)

Fact Addresses, Single-Field Wildcards, and Multifield Variables

- 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 "<–".
- Single-field wildcards can be used in place of variables when the field to be matched against can be anything and its value is not needed later in the LHS or RHS of the rule.
- Multifield variables and wildcards allow matching against more than one field in a pattern.

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, or float.
- 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.

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.

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.