CLIPS Programming 1

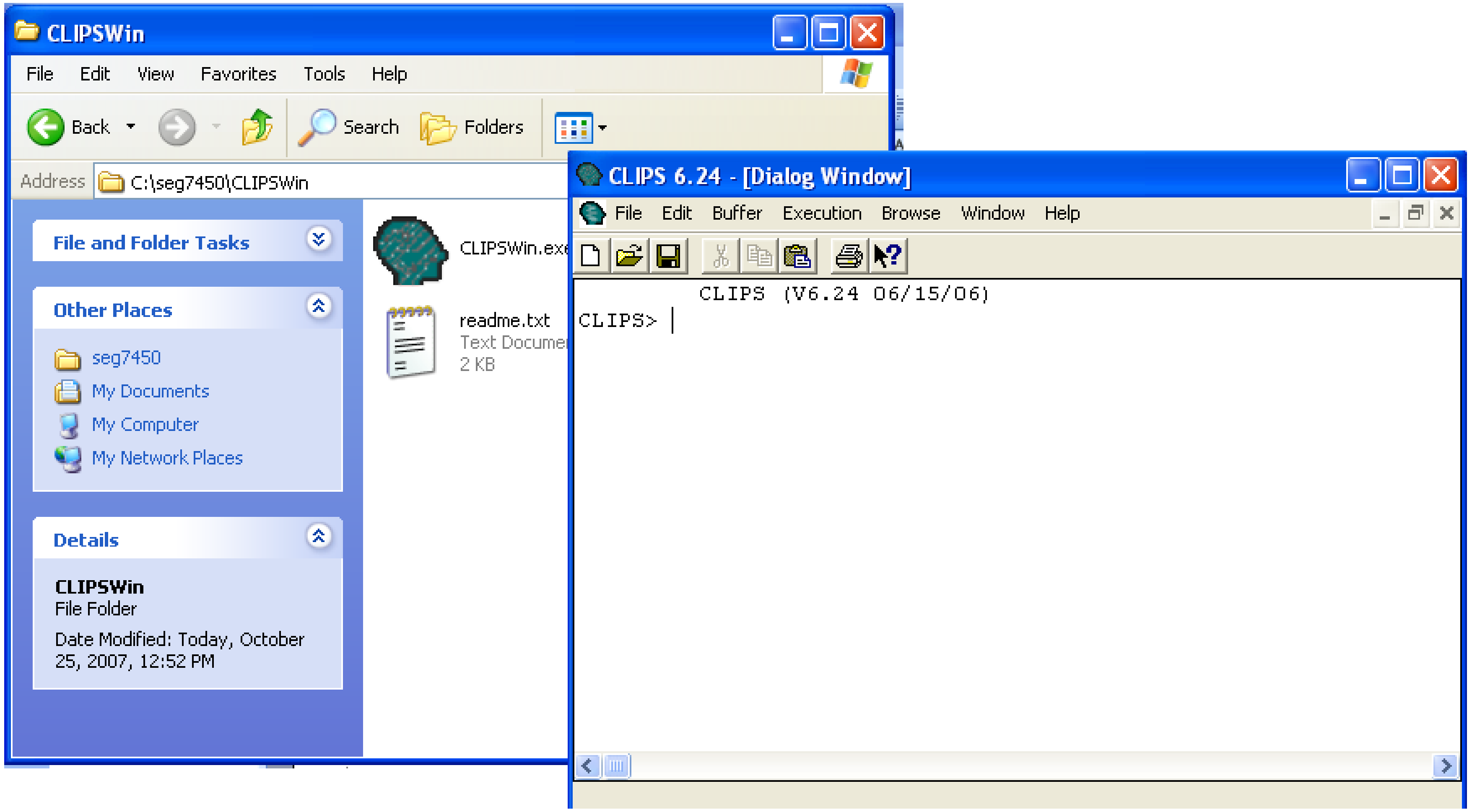
Introduction to CLIPS

CLIPS in PC

- CLIPS is a tool for building expert systems.
- CLIPS is a multiparadigm programming language that provides support for rule-based, object-oriented, and procedural programming.
- You can download it from:
  http://www.se.cuhk.edu.hk/~seem5750/clipswin_executable_6241.zip
  - Download **clipswin_executable_6241.zip**
  - Extract the zipped file.

CLIPS in PC

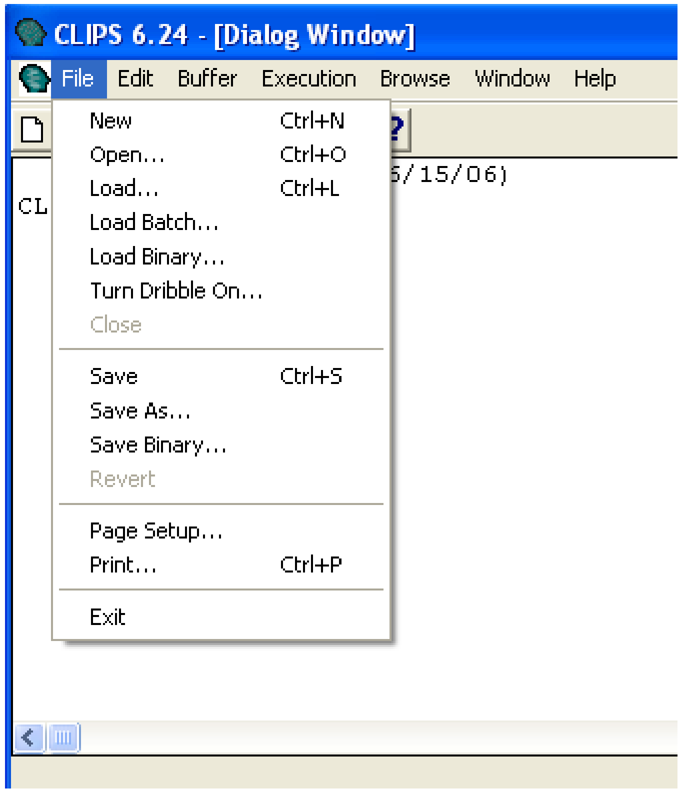
- Double click the program “CLIPSwin.exe”


CLIPSWin
File Edit View Favorites Tools Help
Back Search Folders
Address C:\seg7450\CLIPSWin
File and Folder Tasks
Other Places
seg7450
My Documents
My Computer
My Network Places
Details
CLIPSWin
File Folder
Date Modified: Today, October 25, 2007, 12:52 PM
CLIPSWin.exe
readme.txt
Text Documen
2 KB
CLIPS 6.24 - [Dialog Window]
File Edit Buffer Execution Browse Window Help

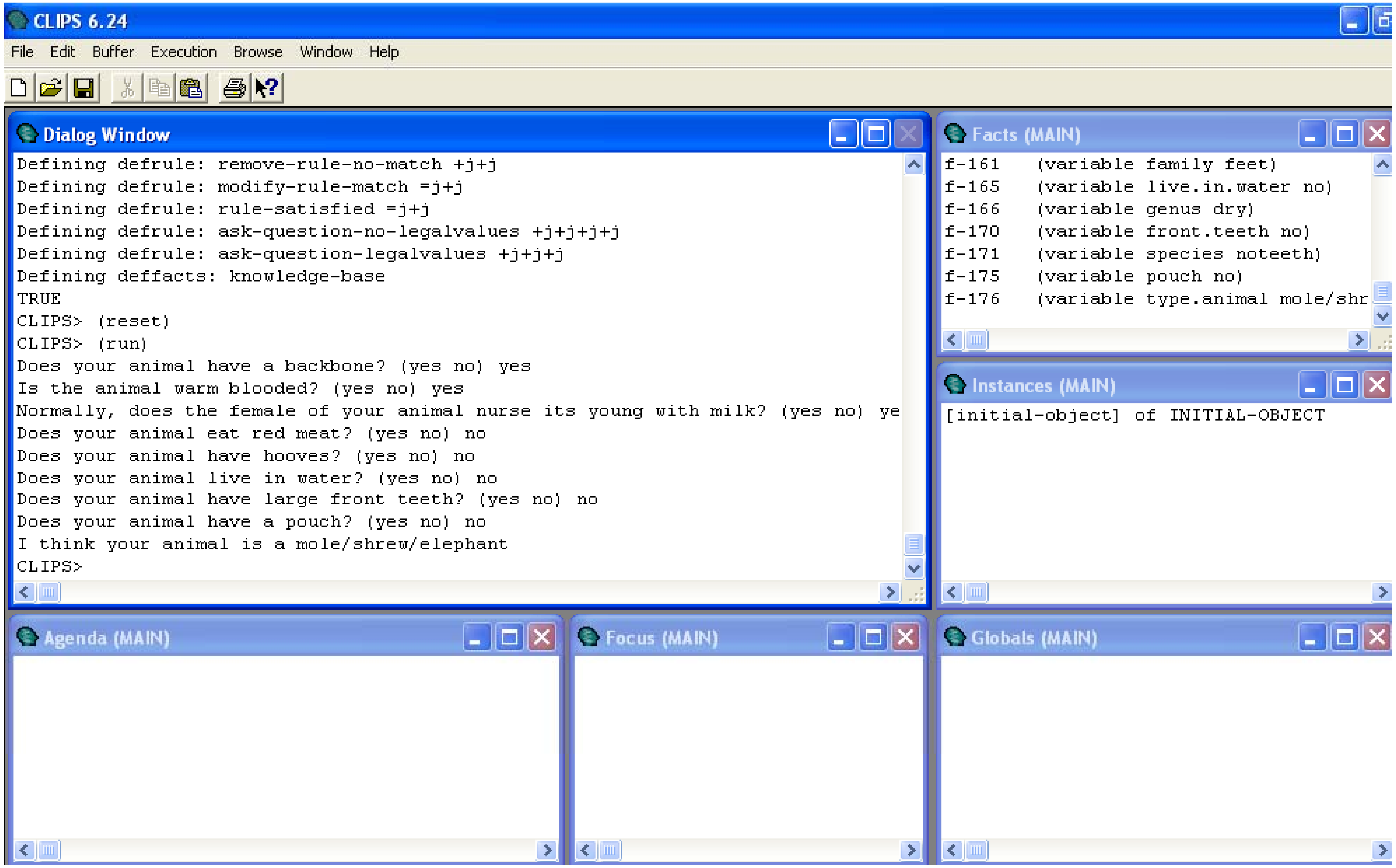
```
          CLIPS (V6.24 06/15/06)
CLIPS>
```

CLIPS in PC

- Select **File** -> **Load** to load a clips program
- Select **Execution** -> **Reset** to reset memory
- Select **Execution** -> **Run** to execute the program
- Select **Execution** -> **Clear CLIPS** to clear the memory
- Select **Browse** -> **Deffacts Manager** to list the name of deffacts construct
- Select **Browse** -> **Deftemplate Manager** to list the name of Deftemplate construct
- Select **Window** -> **Facts** to list the facts in memory
- Select **Window** -> **Agenda** to list the rule to be fired at the current step
- Select **Window** -> **Globals** Window to list the globals variance



CLIPS in PC



```
Defining defrule: remove-rule-no-match +j+j
Defining defrule: modify-rule-match =j+j
Defining defrule: rule-satisfied =j+j
Defining defrule: ask-question-no-legalvalues +j+j+j+j
Defining defrule: ask-question-legalvalues +j+j+j
Defining deffacts: knowledge-base
TRUE
CLIPS> (reset)
CLIPS> (run)
Does your animal have a backbone? (yes no) yes
Is the animal warm blooded? (yes no) yes
Normally, does the female of your animal nurse its young with milk? (yes no) ye
Does your animal eat red meat? (yes no) no
Does your animal have hooves? (yes no) no
Does your animal live in water? (yes no) no
Does your animal have large front teeth? (yes no) no
Does your animal have a pouch? (yes no) no
I think your animal is a mole/shrew/elephant
CLIPS>
```

```
f-161    (variable family feet)
f-165    (variable live.in.water no)
f-166    (variable genus dry)
f-170    (variable front.teeth no)
f-171    (variable species noteeth)
f-175    (variable pouch no)
f-176    (variable type.animal mole/shr
```

```
[initial-object] of INITIAL-OBJECT
```

CLIPS Programming

- **Basic elements of an Expert System**
  - Fact-list : Global memory for data
  - Knowledge-base : Contain all the rules
  - Inference Engine : Control overall execution
- **A program in CLIPS consists of**
  - Facts
  - Rules

Fields

- Tokens represent groups of characters that have special meaning to CLIPS.
- **The group of tokens** known as fields is of particular importance.
- There are eight types of fields, also called the CLIPS primitive data types:
  - float,
  - integer,
  - symbol,
  - string,
  - external address,
  - fact address,
  - instance name, and
  - instance address.
- CLIPS is case-sensitive.

Fields

- Floats: 1.5 1.0 9e+1 3.5e10
- Integers: 1 +3 -1 65
- A symbol, E.g.
    - Space-Station
    - Fire
    - activate_sprinkler_system
  - is a field that starts with a printable ASCII character and is followed by zero or more characters.
    - The end of a symbol is reached when a delimiter is encountered.
    - Delimiters include any nonprintable ASCII character. (spaces, tabs, returns, line feeds) and " ( ) & | < ~ ; ? $
    - Symbols cannot contain delimiters.
- A string must begin and end with double quotation marks, which are part of the field.
  - E.g. "Active the sprinkler system."
- External addresses represent the address of an external data structure returned by a user-defined function (not in the scope of introduction to CLIPS).
- A fact address is used to refer to a specific fact.
- An instance address is like a fact address, but it refers to an instance rather than a fact.
- A series of zero or more fields contained together is referred to as a multifield value. A multifield value is enclosed by left and right parentheses.
  - E.g.
    - ( )
    - (this that)

Entering and Exiting CLIPS

- The CLIPS prompt appear as follows:

```
CLIPS>
```

- Commands can be entered directly to CLIPS; this mode is called the *top level.*
- *The normal mode of leaving CLIPS is the exit command.*

```
(exit)
```

- Each CLIPS command must have a matching number of left and right parentheses.
- The final step in executing a CLIPS command after it has been entered with properly balanced parentheses is to press the return key.

```
CLIPS> (+ 3 4) ┘
7
CLIPS> (exit) ┘
```

Entering and Exiting CLIPS

- A symbol surrounded by parentheses is considered to be a command or function call.
- The input (+ 3 4) is a call to the + function.
- The input (exit) invokes the *exit* command.

Facts

- To solve a problem, a CLIPS program must have data or information with which it can reason.
- A "chunk" of information in CLIPS is called *a fact.*
- Facts consist of a relation name followed by zero or more slots and their associated values.
- The following is an example of a fact:

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

- The symbol *person* is the fact's relation name and the fact contains four slots: *name, age, eye-color,* and *hair-color.*
  - The value of the name slot is "John Q. Public"
  - The value of the eye-color slot is blue.
  - The value of the hair-color slot is black.

Facts

- The order in which slots are specified is irrelevant. The following fact is treated as identical to the first person fact by CLIPS.

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

The Deftemplate Construct

- Before facts can be created, CLIPS must be informed of the list of valid slots for a given relation name.
- Groups of facts that share the same relation name and contain common information can be described using the deftemplate construct.
- The general format of a deftemplate is:

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

- The syntax description <slot-definition> is defined as:

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

- The deftemplate for the person fact:

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

Multifield Slots

- Multislot keyword in their corresponding deftemplates are allowed to contain zero or more values

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

is illegal if *name* is a single-field slot, but legal if *name* is a multifield slot.

Ordered Facts

- Facts with a relation name that does not have a corresponding deftemplate are called *ordered facts.*
- For example, a list of numbers could be represented with the following fact:
  - `(number-list 7 9 3 4 20)`
- There are two cases in which ordered facts are useful.
  - Facts consisting of just a relation name are useful as flags and look identical regardless of whether a deftemplate has been defined. For example,
    - `(all-orders-processed)`
  - For facts containing a single slot, the slot name is usually synonymous with the relation name. For example, the facts:

```
(time 8:45)
(food-groups meat dairy bread
                fruits-and-vegetables)
```

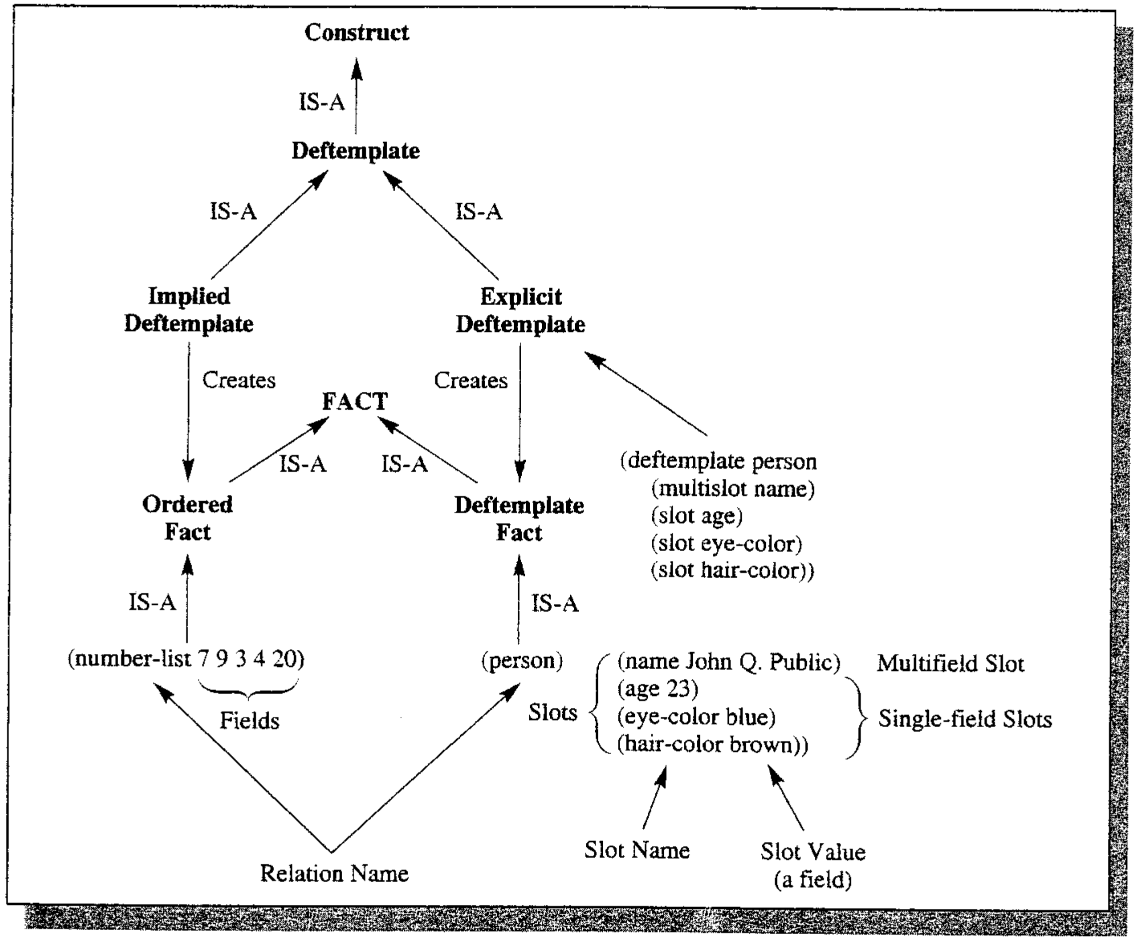
    - are as meaningful as:

```
(time (value 8:45))
(food-groups (values meat dairy bread
                fruits-and-vegetables))
```

Ordered Facts

- A deftemplate fact is a non-ordered fact.
- Ordered versus Non-ordered
  - Ordered facts encode information with fixed position which is not convenient to access
  - Clarity and slot order independence are virtues of deftemplate facts
  - Extra command (e.g. modify, duplicate) for non-ordered facts

Deftemplate Overview



Manipulation of Facts -- assert

- Basic commands to operate on facts:
  - **assert, retract, modify, duplicate, facts**
- **Adding Facts**
  - New facts can be added to the fact list using the assert command.

    ```
    (assert <fact>+)
    ```
  - As an example,

    ```
    CLIPS>
    (deftemplate person
       (slot name)
       (slot age)
       (slot eye-color)
       (slot hair-color))
    CLIPS>
    (assert (person (name "John Q. Public")
             (age 23)
             (eye-color blue)
             (hair-color black)))
    <Fact-0>
    CLIPS>
    ```

Manipulation of Facts -- assert

- More than one fact can be asserted using a single *assert* command. For example, the command:

```
 (assert (person (name "John Q. Public")
             (age 23)
             (eye-color blue)
             (hair-color black))
         (person (name" Jane Q. Public")
             (age 36)
             (eye-color green)
             (hair-color red)))
```

Manipulation of Facts -- facts

- **Displaying Facts**
  - The facts command can be used to display the facts in the fact list.

    ```
    (facts)
    ```
  - For example,

    ```
    CLIPS> (facts)
    f-0  (person (name "John Q. Public")
           (age 23)
           (eye-color blue)
           (hair-color black))
    For a total of 1 fact.
    CLIPS>
    ```
  - Every fact that is inserted into the fact list is assigned a unique fact identifier starting with the letter *f* and followed by an integer called the *fact index.*

Manipulation of Facts -- facts

- The complete syntax for the *facts* command is:

  ```
  (facts [<start> [<end> [<maximum>]]])
  ```

  where <start>, <end>, and <maximum> are positive integers.
  - If no arguments are specified, all facts are displayed.
  - If the <start> argument is specified, all facts with fact indexes greater than or equal to <start> are displayed.
  - If <start> and <end> are specified, all facts with fact indexes greater than or equal to <start> and less than or equal to <end> are displayed.
  - Finally, if <maximum> is specified along with <start> and <end>. no more than <maximum> facts will be displayed.

Manipulation of Facts -- retract

- Removing facts
  - Removing facts from the fact list is called *retraction* and is done with the **retract** command.
  - The syntax of the *retract* command is:

    ```
    (retract <fact-index>+)
    (retract 0)
    ```
  - A single *retract* command can be used to retract multiple facts at once.

    ```
    (retract 0 1)
    ```

Manipulation of Facts -- modify

- 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>)
    ```
  - For example,

    ```
    CLIPS> (modify 0 (age 24))
    <Fact-2>
    CLIPS> (facts)
    f-2 (person (name "John Q. Public")
            (age 24)
            (eye-color blue)
            (hair-color black))
    For a total of 1 fact.
    CLIPS>
    ```
  - A new fact index is generated for a modified fact.

Manipulation of Facts -- duplicate

- **Duplicating facts**
- **duplicate** command

```
CLIPS> (duplicate 2 (name "Jack S. Public“))
<Fact-3>
CLIPS> (facts)
f-2 (person (name "John Q.      Public")
          (age 24 )
          (eye-color blue)
          (hair-color black))
f-3 (person (name "Jack S. Public")
          (age 24)
          (eye-color blue)
          (hair-color black))
For a total of 2 facts.
CLIPS>
```

- The modify and duplicate commands cannot be used with ordered facts.
- To enable the duplicate command, the following command may be required:

```
(set-fact-duplication TRUE)
```

Manipulation of Facts -- watch

- The **watch** command is useful for debugging programs.

  ```
  (watch <watch-item>)
  ```

  – where <watch-item> is one of the symbols facts, rules, activations, statistics, compilations, focus, deffunctions, globals, generic-functions, methods, instances, slots, messages, message-handlers, or all.
- If facts are being watched, CLIPS will automatically print a message indicating that an update has been made to the fact list whenever facts are asserted or retracted.
- The effects of a *watch* command may be turned off by using the corresponding unwatch command.

  ```
  (unwatch <watch-item>)
  ```

Manipulation of Facts -- watch

```
CLIPS> (facts 3 3)
f-3 (person (name   "Jack S.       Public")
            (age 24)
            (eye-color blue)
            (hair-color black))
For a total of 1 fact.
CLIPS> (watch facts)
CLIPS> (modify 3 (age 2))
<== f-3     (person (name   "Jack S.       Public")
            (age 24)
            (eye-color blue)
            (hair-color black))
==> f-4     (person (name "Jack S. Public")
            (age 25)
            (eye-color blue)
            (hair-color black) )
<Fact-4>
CLIPS>
```

The Deffacts Construct

- It is often convenient to be able to automatically assert a set of facts instead of typing in the same assertions from the top level.
- This is particularly true for facts that are known to be true before running a program (i.e., the initial knowledge).
- Groups of facts that represent initial knowledge can be defined using the deffacts construct.

```
(deffacts people "Some people we know"
  (person (name "John Q. Public") (age 24)
          (eye-color blue) (hair-color black))
  (person (name "Jack S. Public") (age 24)
          (eye-color blue) (hair-color black))
  (person (name "Jane Q. Public") (age 36)
          (eye-color green) (hair-color red)))
```

The Deffacts Construct

- The general format of a deffacts is:

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

- The facts in a deffacts statement are asserted using the CLIPS reset command. The *reset* command removes all facts from the fact list and then asserts the facts from existing deffacts statement.

```
(reset)
```

- Even if you have not defined any deffacts statements, a reset will assert the fact (initial-fact).
  - The fact identifier of the initial-fact is always f-0.

The Components of a Rule

- Rules can be typed directly into CLIPS or loaded in from a file of rules.
- The pseudocode for one of the possible rules in the industrial plant monitoring expert system is shown as follows:

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

- Before converting the pseudocode to a rule, the deftemplates for the types of facts referred to by the rule must be defined.

```
(deftemplate emergency (slot type))
```

  – ;where the *type* field of the *emergency* fact would contain symbols such as fire, flood, and power outage.

```
(deftemplate response (slot action))
```

  – where the *action* field of the *response* fact indicates the response to be taken.

- The rule expressed in CLIPS is:

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

The Components of a Rule

- The general format of a rule is:

```
(defrule <rule name> [<comment>]
  <patterns>* ; Left-Hand Side (LHS) of the rule
  =>
  <actions>* ); Right-Hand Side (RHS) of the rule
```

- The entire rule must be surrounded by parentheses and each of the patterns and actions of the rule must be surrounded by parentheses.
- A rule may have multiple patterns and actions. The parentheses surrounding patterns and actions must be properly balanced if they are nested.
- The header of the rule consists of three parts. The rule must start with the **defrule** keyword, followed by the name of the rule.
- Next comes an optional comment string.

The Components of a Rule

- Example:

```
; Rule header
(defrule fire-emergency “An example rule"
  ; Patterns
  (emergency (type fire))
  ; THEN arrow
  =>
  ; Actions
  (assert (response (action activate-sprinkler-system))))
```

- After the rule header are zero or more conditional elements (CEs). The simplest type of CE is a **pattern CE** or simply **pattern.**
  - Each pattern consists of one or more constraints intended to match the fields of a deftemplate fact.

The Components of a Rule

- CLIPS attempts to match the patterns of rules against facts in the fact list.
  - If all the patterns of a rule match facts, the rule is **activated** and put on the **agenda,** the collection of activated rules.
- The arrow is a symbol representing the beginning of the THEN part of an IF-THEN rule.
  - The part of the rule before the arrow is called the left-hand side (LHS) and the part after the arrow is called the right-hand side (RHS).
- The last part of a rule is the list of actions that will be executed when the rule **fires**.
  - A program normally ceases execution when there are no rules on the agenda.
  - When there are multiple rules on the agenda, CLIPS automatically determines which is the appropriate rule to fire.

The Agenda and Execution

- **Execution -- run**
- A CLIPS program can be made to run with the run command. The syntax of the *run* command is:

```
(run [<limit>])
```

  - where the optional argument <limit> >is the maximum number of rules to be fired.
  - If <limit> is not included or <limit> is -1, rules will be fired until none are left on the agenda.
- **Reset**
  - Because rules require facts to execute, the *reset* command is the key method for starting or restarting an expert system in CLIPS.

Displaying the Agenda

- The list of rules on the agenda can be displayed with the agenda command.

  ```
  (agenda)
  ```

- Example:

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

- The 0 indicates the salience of the rule on the agenda.

Rules and Refraction

- With the *fire-emergency* rule on the agenda, the *run* command will now cause the rule to fire.
- The fact (response (action activate-sprinkler-system» will be added to the fact list as the action of the rule:

```
CLIPS> (run)
CLIPS> (facts)
f-0            (initial-fact)
f-1            (emergency (type fire))
f-2            (response (action activate-
  sprinkler-system))
For a total of 3 facts.
CLIPS>
```

- Rules in CLIPS exhibit a property called refraction, which means they won’t fire more than once for a specific set of facts.
- **Refresh**
  – The **refresh** command can be used to make the rule fire again.
    - `(refresh <rule-name>)`

Watching Activations, Rules, and Statistics

- Examples of watching activations:

```
CLIPS> (reset)
CLIPS> (watch activations)
CLIPS> (assert (emergency (type fire)))
==> Activation 0      fire-emergency: f-l
<Fact-1>
CLIPS> (agenda)
0   fire-emergency:   f-l
For a total of 1 activation.
CLIPS> (retract 1)
<== Activation 0      fire-emergency: f-1
CLIPS> (agenda)
CLIPS>
```

Watching Activations, Rules, and Statistics

- Rules being watched:

```
CLIPS> (reset)
CLIPS> (watch rules)
CLIPS> (assert (emergency (type fire)))
==> Activation 0      fire-emergency: f-l
<Fact-1>
CLIPS> (run)
FIRE     1 fire-emergency: f-l
CLIPS> (agenda)
CLIPS>
```

Watching Activations, Rules, and Statistics

- If statistics are watched, informational messages will be printed at the completion of a run.

```
CLIPS> (unwatch all)
CLIPS> (reset)
CLIPS> (watch statistics)
CLIPS> (assert (emergency (type fire)))
<Fact-1>
CLIPS> (run)
1 rules fired
3 mean number of facts (3 maximum)
1 mean number of instances (1 maximum)
1 mean number of activations (1 maximum)
CLIPS> (unwatch statistics)
CLIPS>
```

Watching Activations, Rules, and Statistics

- CLIPS keeps statistics on the number of facts, activations, and instances.
  - The mean number of facts is the sum of the total number of facts in the fact list after each rule firing divided by the number of rules fired.
- The mean and maximum numbers of activations statistics indicate the average number of activations per rule firing and the largest number of activations on the agenda for anyone rule firing.

Commands for Manipulating Constructs

- Displaying the List of Members of a Specified Construct
  - `(list-defrules)`
  - `(list-deftemplates)`
  - `(list-deffacts)`
- Displaying the Text Representation of a Specified Construct Member
  - `(ppdefrule <defrule-name>)`
  - `(ppdeftemplate <deftemplate-name>)`
  - `(ppdeffacts <deffacts-name>)`
- Deleting a Specified Construct Member
  - `(undefrule <defrule-name>)`
  - `(undeftemplate <deftemplate-name>)`
  - `(undeffacts <derfacts-name>)`
- Clearing All Constructs from the CLIPS Environment
  - `(clear)`

The PRINTOUT Command

- Besides asserting facts in the RHS of rules, the RHS can also be used to print out information using the printout command.

  ```
  (printout <logical-name> <print-items>*)
  ```

  - where <logical-name> indicates the output destination of the *printout* command and <print-items>* are the zero or more items to be printed by this command.
- Example of rule using the printout command:

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

- The logical name *t* tells CLIPS to send the output to the standard output device of the computer, usually the terminal.

Using Multiple Rules

- In addition to the *fire-emergency* rule, the expert system monitoring the industrial plant might include a rule for emergencies in which nooding has occurred.

```
(defrule fire-emergency
  (emergency (type fire))
  =>
  (printout t "Activate the sprinkler system"
               crlf) )
(defrule flood-emergency
  (emergency (type flood))
  =>
  (printout t "Shut down electrical equipment"
               crlf) )
```

Using Multiple Rules

- Rules with more than one pattern could be used to express these conditions.

```
(deftemplate extinguisher-system
   (slot type)
   (slot status))
(defrule class-A-fire-emergency
   (emergency (type class-A-fire))
   (extinguisher-system (type water-sprinkler)
           (status off))
   =>
   (printout t "Activate water sprinkler" crlf))
(defrule class-B-fire-emergency
   (emergency (type class-B-fire))
   (extinguisher-system (type carbon-dioxide)
           (status off))
   =>
   (printout t "Use carbon dioxide extinguisher"
           crlf) )
```

Using Multiple Rules

- Any number of patterns can be placed in a rule.
  - The important point to realize is that the rule is placed on the agenda only if ***all*** the patterns are satisfied by facts.
  - This type of restriction is called an ***and conditional element***.

The SET-BREAK Command

- CLIPS has a debugging command called *set-break* that allows execution to be halted before any rule from a specified group of rules is fired.
  - A rule that halts execution before being fired is called a *breakpoint.*

```
(set-break <rule-name>)
```

  - Example with the following rules:

```
(defrule first
  =>
  (assert (fire second)))

(defrule second
  (fire second)
  =>
  (assert (fire third)))

(defrule third
  (fire third)
  => )
```

The SET-BREAK Command

- With set-break

```
CLIPS> (set-break second)
CLIPS> (reset)
CLIPS> (run)
FILE    1 first: f-0
Breaking on the second
CLIPS>
```

- The show-breaks command can be used to list all breakpoints. Its syntax is:

```
(show-breaks)
```

- The remove-break command can be used to remove breakpoints. Its syntax

```
(remove-break [<rule-name>])
```

Loading and Saving Constructs

- **Loading Constructs from a File**

  ```
  (load <file-name>)
  ```

  - where <file-name> is a string or symbol containing the name of the file to be loaded.
  - E.g. (load “fire.clp”)
- **Saving Constructs to a File**
  - The save command allows the set of constructs stored in CLIPS to be saved to a disk file.

    ```
    (save <file-name>)
    ```

  - E.g. (save “fire.clp”)

Commenting Constructs

- A comment in CILPS is any text that begins with a semicolon and ends with a carriage return.
- It is good to include comments describing the constructs, and giving information regarding the program.

```
;***************************
;* Programmer: G. D. Riley *
;* Title: The Fire Program *
;* Date: 05/17/04          *
; Deftemplate
     (deftemplate emergency “template #1”
      (slot type))     ; What type of emergency

     (deftemplate response “template #2”
       (sloto type))   ; How to respond

; The purpose of this rule is to activate
; the sprinkler if there is a fire
     (defrule fire-emergency “An example rule”  ;IF
              (emergency (type fire))
     =>       ; THEN
     ;Activate the sprinkler system
     (assert (response (action activate-sprinker-system))))
```