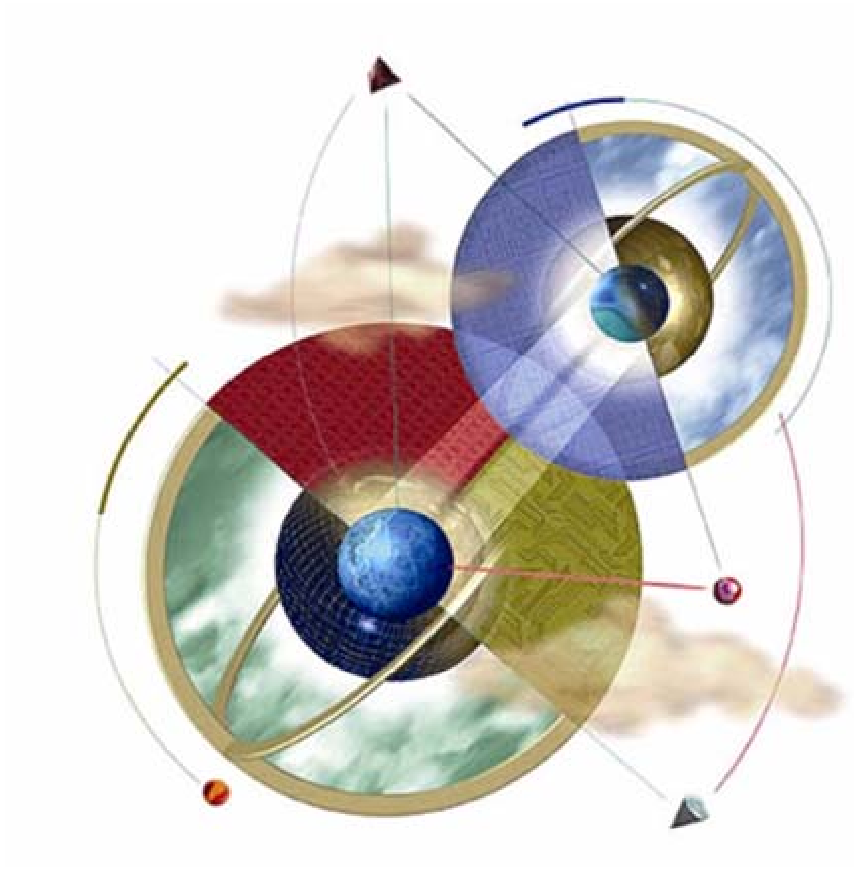
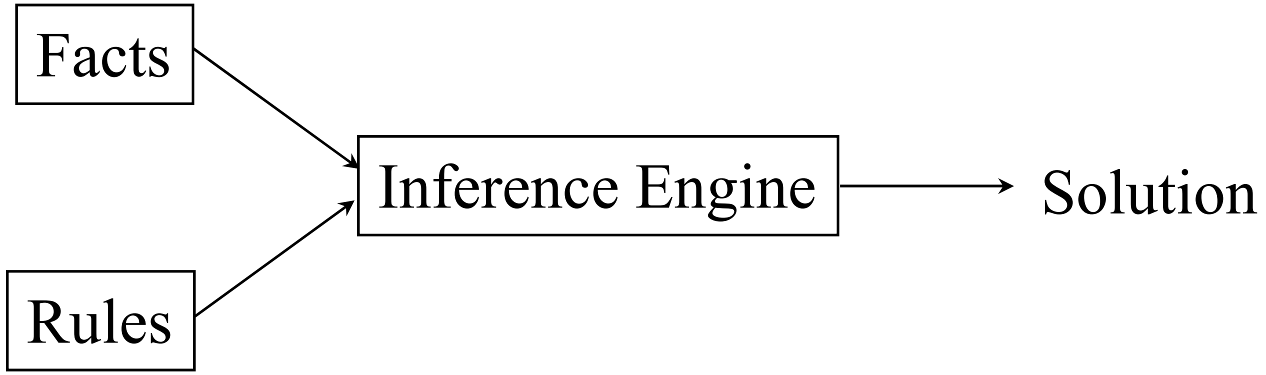


Chapter 7: Introduction to CLIPS

Expert Systems: Principles and Programming, Fourth Edition

What is CLIPS?

- ***C L**anguage **I**ntegrated **P**roduction **S**ystem (CLIPS)*
- CLIPS was designed using the C language at the NASA/Johnson Space Center
- Three basic components:
  - fact list
  - knowledge base (rules)
  - inference engine



CLIPS Characteristics

- CLIPS is a multiparadigm programming language that provides support for:
  - Rule-based programming
    - CLIPS supports only forward-chaining rules
  - Object-oriented programming
    - The OOP capabilities of CLIPS are referred to as CLIPS Object-Oriented Language (COOL)
  - Procedural programming
    - The procedural language capabilities of CLIPS are similar to C languages

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 (fields).
- A field is a special type of token of which there are 8 types.
  - Float, integer, symbol, string, external address, fact address, instance name.

Numeric Fields & Symbol Fields

- The floats and integers make up the numeric fields – simply numbers.
  - Integers have only a sign and digits.
    - 20 or -17
  - Floats have a decimal and possibly “e” for scientific notation.
    - -1.5 or 3.5e10
- Symbols begin with printable ASCII characters followed by zero or more characters.
  - CLIPS is case sensitive

String Fields

- Strings must begin and end with double quotation marks (“).
  - “Joe Smith”
- Spaces within the string are significant.
  - “space “ ≠ “ space”
- The actual delimiter symbols can be included in a string by preceding the character with a backslash (\).
  - “\”tokens\”” or “\\token\\”

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

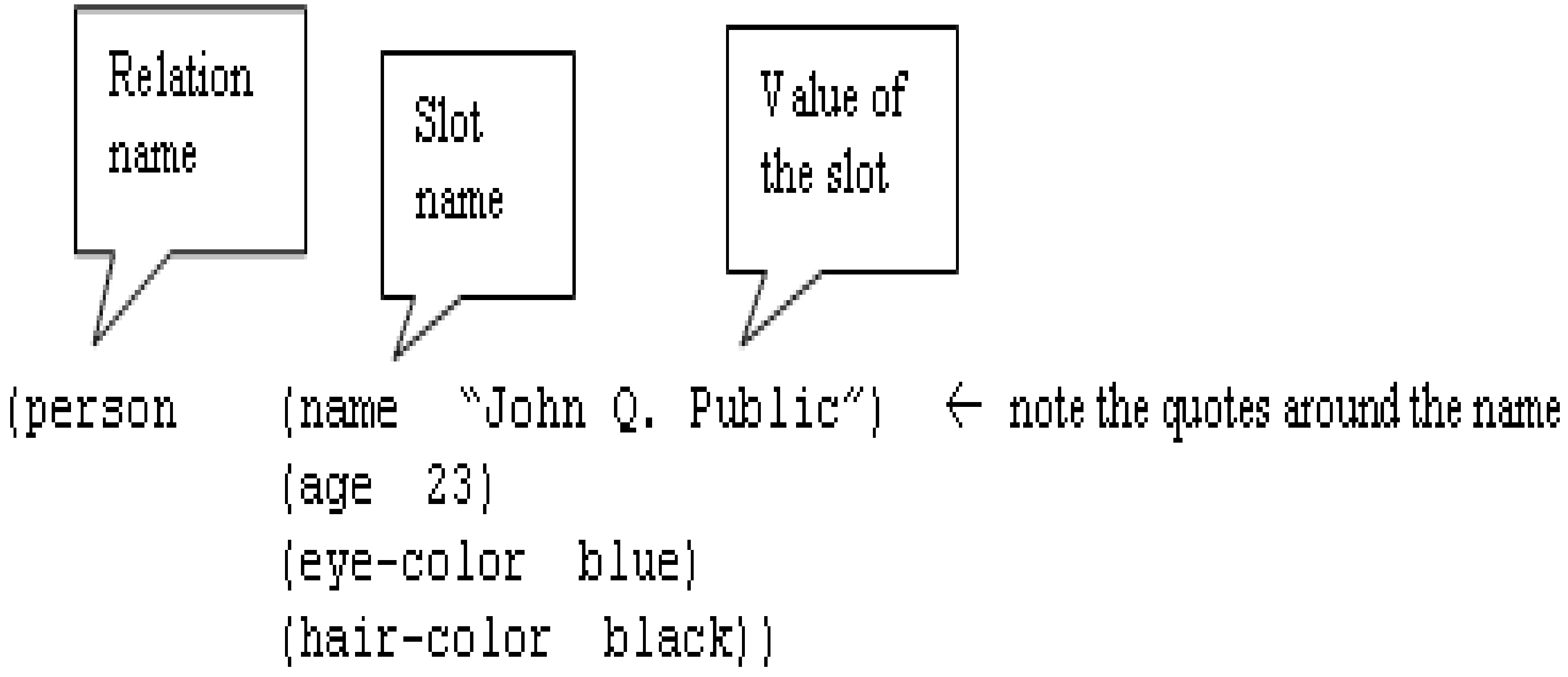
- The CLIPS prompt is: CLIPS>
- This is the type-level mode where commands can be entered.
- To exit CLIPS, one types: CLIPS> (exit) ↵
- CLIPS will accept input from the user / evaluate it / return an appropriate response:

  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 (a symbol field)
  - Zero or more slots with associated values

Example Fact in CLIPS

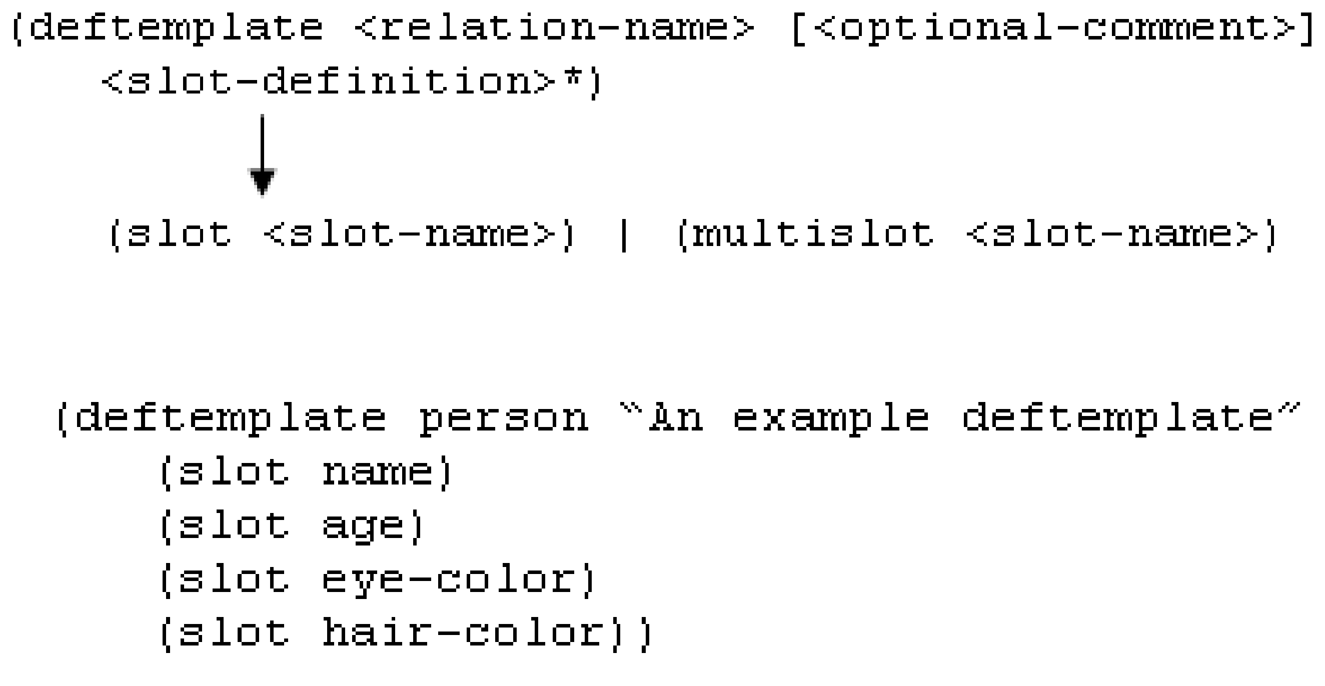


```
(person    (name   "John Q. Public")
           (age   23)
           (eye-color  blue)
           (hair-color  black))
```

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))
```

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.
  - (example <integer>)
    - (example 1) or (example 400)
- * following a description means that the description can be replaced by zero or more occurrences of the specified value.
  - <integer>*
    - 1 or 1 2 or   1 2 3

CLIPS Notation

- Descriptions followed by + mean that one or more values specified by description should be used in place of the syntax description.
  - <integer>+   equals
  - <integer> <integer>*
- A vertical bar | indicates a choice among one or more of the items separated by the bars.
  - All | none | some
    - All
    - None
    - Some

Deftemplate & Facts

- (person (name Peter) (age 12) (height 140) (weight 40))
  (person (name John) (age 30) (height 175) (weight 65))

person	name	age	height	weight
	Peter	12	140	40
	John	30	175	65

- (deftemplate <relation-name> <slot-def>*)
  - (deftemplate person (slot name) (slot age) (slot height) (slot weight))
  - (deftemplate team (slot id) (multislot members))
    - (team (id 01) (members Joe Mary Peter))
    - (team (id 02) (members Frank John Sue David Kevin))

Deftemplate vs. Ordered Facts

- Facts with a relation name defined using deftemplate are called *deftemplate facts*.
  - (person (name Joe) (age 20))
- 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.
  - (person Joe John Kevin)
  - When to use ordered facts?
    - (all-orders-processed) ➔ as flags
    - (time 8:45) → same slot name and relation name
    - (food-groups meat dairy bread fruits-and-vegetables)

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) ↵

fact identifier

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

For a total of 1 fact.
```

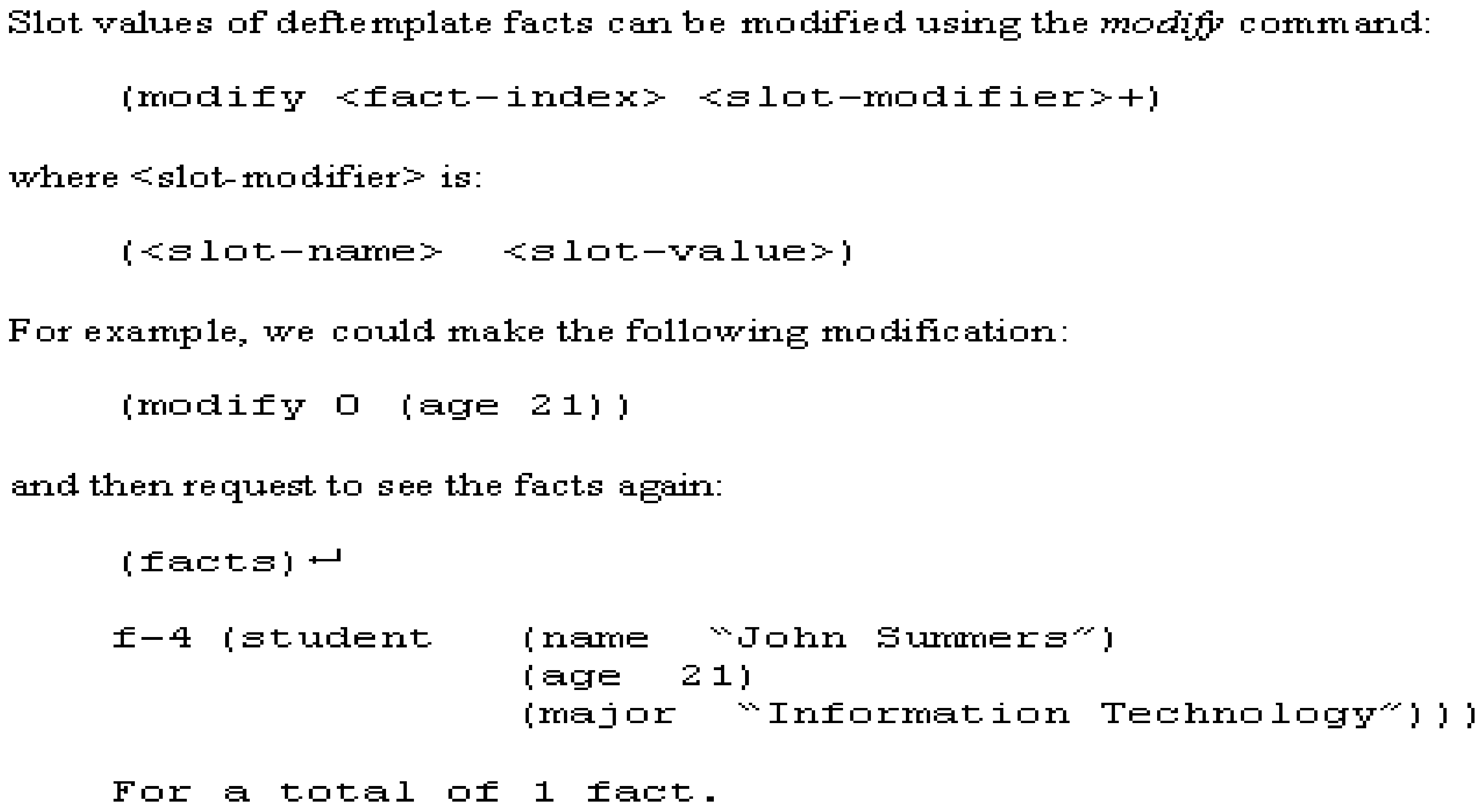
Removing Facts

- Just as facts can be added, they can also be removed.
- Removing facts results in gaps in the fact identifier list.
  - (retract <fact-index>+)
- 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.
  - (watch <watch-item>)
    - watch-item can be facts, rules, activations, or compilations
  - (unwatch <watch-item>)
- 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> <facts>*)

- The *reset* command is used to assert the facts in a deffacts statement.
  - but all existing facts will be removed (reset)

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

Rules

- (defrule <rule-name>

      <patterns>*   ------LHS
      =>
      <actions>*)   ------RHS

  - (defrule fire-emergency-1

        (emergency (type fire))
        =>
        (assert (response (action sprinkle))))

  - (defrule fire-emergency-2

        (emergency (type fire))
        (fired-object (material oil))
        =>
        (printout t “The fire should be excluded from air.” crlf))

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 (RHS).
- 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.

agenda

10 *fire-emergency-1 :f-1*
```
  0 fire-emergency-2  :f-1,f-4
-10 find-height-175    :f-2
-10 find-name-John     :f-3,f-5
           :
```

(run 1) $\longrightarrow$

agenda

```
  0 fire-emergency-2 :f-1,f-4
-10 find-height-175    :f-2
-10 find-name-John     :f-3,f-5
           :
```

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.
  - (refresh <rule-name>)

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.
  - (ppdefrule <rule-name>)

Commands

- The *undefrule*, *undeftemplate*, and *undeffacts* commands are used to delete a defrule, a deftemplate, and a deffact, respectively.
  - (undefrule <rule-name>)
- The *clear* command clears the CLIPS environment
  - (clear)

Commands (cont.)

- The *printout* command can be used to print information.
  - (printout <logical-name> <print-items>*)
    - the logical name t for standard output device
    - crlf forces a line feed
    - (printout t “The fire should be excluded from air.” crlf))
- *Set-break* – allows execution to be halted *before* any rule from a specified group of rules is fired.
  - (set-break <rule-name>)
    - a debugging command to set a breakpoint
  - (remove-break [<rule-name>])
    - remove one or all breakpoints

Commands (cont.)

- *Load* – allows loading of rules from an external file.
  - (load <file-name>)
    - (load “c:\\homework\\ex1.clp”)
- *Save* – opposite of load, allows saving of current constructs stored in CLIPS to disk
  - (save <file-name>)
    - (save “c:\\homework\\ex2.clp”)

Execute a CLIPS Program

$$\begin{tikzcd} \boxed{\text{Load}} \arrow[r] & \boxed{\text{Reset}} \arrow[r] & \boxed{\text{Run}} \end{tikzcd}$$

1. Edit a CLIPS program file in text format
   – include deftemplate, deffacts, and defrule
2. Open CLIPS system and load the program file
   – (load <file-name>) or [File] → [Load]
3. Reset the CLIPS system
   – (reset) or [Execution] → [Reset]
4. Execute
   – (run) or [Execution] → [Run]
5. Clear the CLIPS system
   – (clear) or [Execution] → [Clear CLIPS]

Fire-Emergency Expert System

- A fire-emergency expert system for dealing with four types of fire-emergency:
  - Type A： the firing material is paper, wood, or cloth
  - Type B： the firing material is oil or gas
  - Type C： the firing material is battery
  - Type D： the firing material is chemicals
- Dealing with the four types of fire-emergency
  - Type A：general extinguisher or water
  - Type B：foam extinguisher or carbon dioxide extinguisher
  - Type C：dry chemicals extinguisher or carbon dioxide extinguisher
  - Type D：graphitized coke

The CLIPS Program

```
(deftemplate fire (slot type))

(deftemplate fired-object (slot material))

(defrule fire-type-A-1
        (fired-object (material paper))
        =>
        (assert (fire (type A))))

(defrule fire-type-A-2
        (fired-object (material wood))
        =>
        (assert (fire (type A))))

(defrule fire-type-A-3
        (fired-object (material cloth))
        =>
        (assert (fire (type A))))

(defrule fire-type-B-1
        (fired-object (material oil))
        =>
        (assert (fire (type B))))

(defrule fire-type-B-2
        (fired-object (material gas))
        =>
        (assert (fire (type B))))

(defrule fire-type-C-1
        (fired-object (material battery))
        =>
        (assert (fire (type C))))

(defrule fire-type-D-1
        (fired-object (material chemicals))
        =>
        (assert (fire (type D))))
```

The CLIPS Program (cont.)

- (defrule deal-with-type-A
  ```
  (fire (type A))
  =>
  (printout t “general extinguisher or water” crlf))
  ```
- (defrule deal-with-type-B
  ```
  (fire (type B))
  =>
  (printout t “foam extinguisher or carbon dioxide extinguisher” crlf))
  ```
- (defrule deal-with-type-C
  ```
  (fire (type C))
  =>
  (printout t “dry chemicals extinguisher or carbon dioxide extinguisher” crlf))
  ```
- (defrule deal-with-type-D
  ```
  (fire (type D))
  =>
  (printout t “graphitized coke” crlf))
  ```
  - input fact: (assert (fired-object (material paper)))
  - or default fact: (deffacts initial (fired-object (material paper)))

Commenting and Variables

- Comments – provide a good way to document programs to explain what constructs are doing.
  - – begins with a semicolon (;) and ends with a carriage return
- Variables – store values, syntax requires preceding with a question mark (?) or ($?)
  - – ?var
    - single-field variable <=> slot
  - – $?var
    - multiple-field variable <=> multislot

Single-Field Wildcards, Multifield Wildcards, and Fact Address

- 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 wildcards allow matching against more than one field in a pattern.
  - $?
- 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 "<-".
  - ?abc <- (person (name David) (age 18) (weight 70))

Examples

- (defrule find-height-175

  (person (name ?name) (age ?) (height 175) (weight ?w))

  =>

  (printout t ?name “is 175 cm tall and “ ?w “ kg weight.”))
  - f-12 (person (name David) (age 18) (height 175) (weight 70))
  - f-15 (person (name John) (age 30) (height 175) (weight 65))
- (defrule find-height-175-and-is-father

  (person (name ?name) (age ?) (height 175) (weight ?w))

  (father-child (father ?name) (child $?))

  =>

  (printout t ?name “is 175 cm tall and “ ?w “ kg weight.”))
  - f-12 (person (name David) (age 18) (height 175) (weight 70))
  - f-15 (person (name John) (age 30) (height 175) (weight 65))
  - f-20 (father-child (father John) (child Mary Sue Joe))

Single-Field Wildcards

```
• (deftemplate person
   (multislot name)
   (slot social-security-number))
• (deffacts some-people
   (person (name John Q. Public)
         (social-security-number 483-98-9083))
   (person (name Jack R. Public)
         (social-security-number 483-98-9084)))
• (defrule print-social-security-numbers
         (print-ss-numbers-for ?last-name)
         (person (name ?first-name ?middle-name ?last-name)
                  (social-security-number ?ss-number))
  =>
         (printout t ?ss-number crlf))
```

If the first-name and middle-name are not important, they can be replaced by question marks (?)

Fact Addresses

A variable can be bound to the ***fact address*** of the fact matching a pattern on the LHS of a rule by using the pattern ***binding operator*** "<-".

-
  ```
  (deftemplate person
    (slot name)
    (slot address))
  ```

-
  ```
  (deftemplate moved
    (slot name)
    (slot address))
  ```

-
  ```
  (defrule process-moved-information
    ?f1 <- (moved (name ?name)

            (address ?address))
    ?f2 <- (person (name ?name))
    =>
    (retract ?f1)
    (modify ?f2 (address ?address)))
  ```

-
  ```
  (deffacts example
    (person (name “John Hill”)
             (address “25 Mulberry Lane”))
    (moved (name “John Hill”)
             (address “37 Cherry Lane”)))
  ```

Retract Facts

- (Retract <fact-address>+)
- (defrule add-sum

```
    ?data <- (data (item ?value))
    ?old-total <- (total ?total)
    =>
    (retract ?old-total ?data)
    (assert (total (+ ?total ?value))))
```

  - f-2 (total 0)
  - f-4 (data (item 20))
  - f-5 (data (item 15))
  - f-8 (data (item 12))

Infinite Loop

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

- The program executes an infinite loop
  - it asserts a new person fact after modification
  - reactivate the process-moved-information rule

Multifield Variables

- a multifield variable is referred to on the RHS
  - not necessary to include the $ as part of the variable name on the RHS
  - the $ is only used on the LHS to indicate the zero or more fields
  - (defrule print-children

    (person (name $?name) (children $?children))

    =>

    (printout t ?name " has children " ?children crlf))
  - the multifield values are surrounded by parentheses when printed
    - (John H. Smith) has children (Joe Paul Mary)

Using Multifield Wildcards

```
• (deftemplate person
    (multislot name)
    (slot social-security-number))
• (deffacts some-people
    (person (name John Q. Public)
          (social-security-number 483-98-9083))
    (person (name Jack R. Public)
          (social-security-number 483-98-9084)))
• (defrule print-social-security-numbers
          (print-ss-numbers-for ?last-name)
          (person (name ?first-name ?middle-name ?last-name)
                      (social-security-number ?ss-number))
  =>
          (printout t ?ss-number crlf))
```

Using $? To replace this place

Example

- (deftemplate person
  - (multislot name)
  - (multislot children)
- (deffacts some-people

```
(person (name John Q. Public)
               (children Jane Paul Mary))
(person (name Jack R. Public)
               (children Rick)
```

- (defrule print-children

```
   (print-children $?name)
   (person (name $?name)
               (children $?children))
=>
   (printout t ?name “ has children “ ?children crlf))
```

Example

- (defrule find-child
```
    (find-child ?child)
    (person (name $?name)
            (children $?before ?child $?after))
 =>
    (printout t ?name " has child " ?child crlf)
    (printout t "Other children are ?before " " ?after crlf))
```
- (assert (find-child Paul))
- What if we assert a fact that Joe Fiveman has three children named Joe, Joe, and Joe?
  - (assert (person (name Joe Fiveman)
            (children Joe Joe Joe)))

  (assert (find-child Joe))

Exercise

- Write one or more rules that will generate all the permutations of a base fact and print them out. For example, the fact:

(base-fact red green blue)

Should generate the output

```
Permutation is (red green blue)
Permutation is (red blue green)
Permutation is (green red blue)
Permutation is (green blue red)
Permutation is (blue red green)
Permutation is (blue green red)
```

- Given a natural number N, generate a sequence from 1 to n and all their permutations.

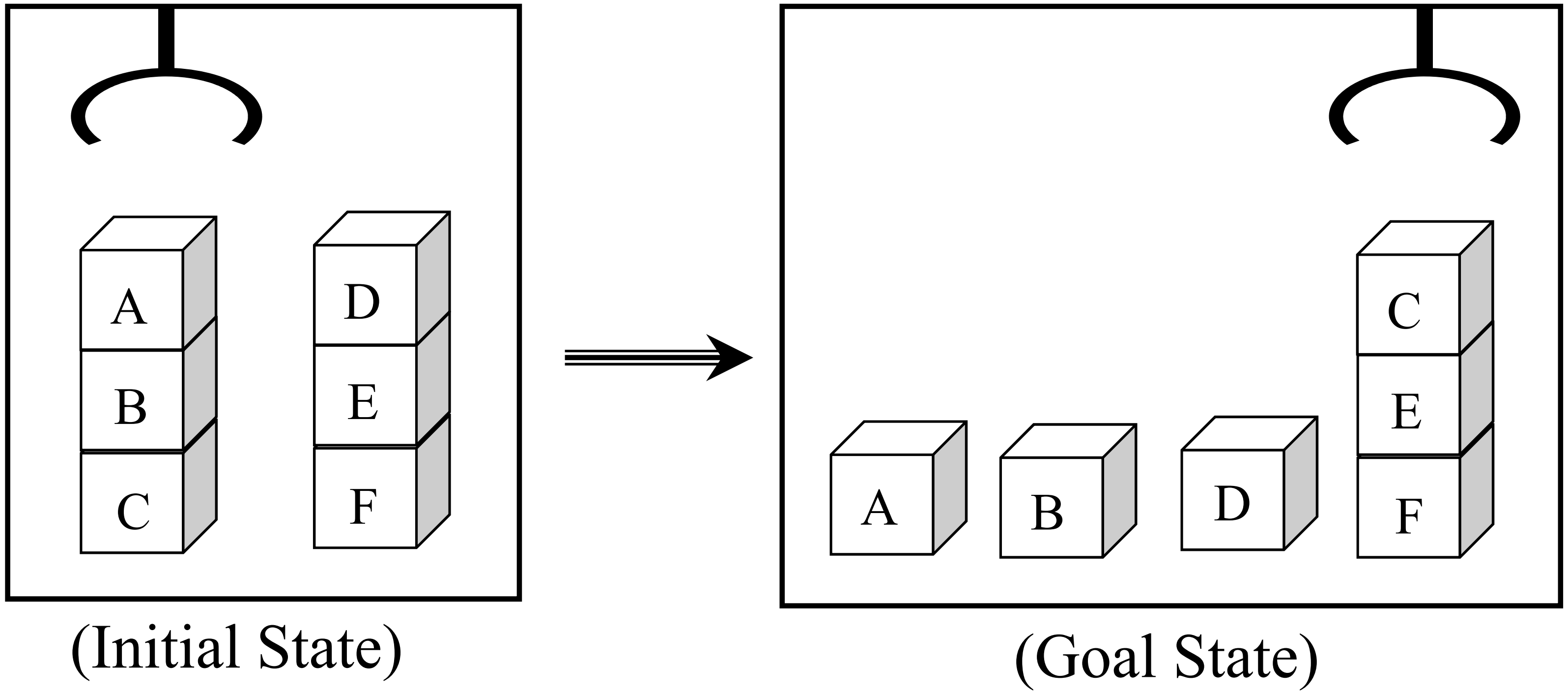
Implementing a Stack

```
(defrule push-value
     ?push-value <- (push-value ?value)
     ?stack <- (stack $?rest)
=>
     (retract ?push-value ?stack)
     (assert (stack ?value $?rest))
     (printout t "pushing value " ?value crlf))
(defrule pop-value-valid
     ?pop-value <- (pop-value)
     ?stack <- (stack ?value $?rest)
=>
     (retract ?pop-value ?stack)
     (assert (stack $?rest))
    (printout t "Popping value " ?vaue crlf)))
```

```
(defrule pop-value-invalid
     ?pop-value <- (pop-value)
     (stack)
=>
     (retract ?pop-value)
     (printout t "Popping from empty
     stack " crlf))
```

Blocks World

- goal: to move one block on top of another block
  - to move C on top of E



The CLIPS Program for Blocks World

```
(deffacts initial-state
   (block A)
   (block B)
   (block C)
   (block D)
   (block E)
   (block F)
   (on-top-of (upper nothing) (lower A))
   (on-top-of (upper A) (lower B))
   (on-top-of (upper B) (lower C))
   (on-top-of (upper C) (lower floor))
   (on-top-of (upper nothing) (lower D))
   (on-top-of (upper D) (lower E))
   (on-top-of (upper E) (lower F))
   (on-top-of (upper F) (lower floor))
   (goal (move C) (on-top-of E))
)
```

```
(deftemplate on-top-of (slot upper) (slot lower))

(deftemplate goal (slot move) (slot on-top-of))

(defrule move-directly
   ?goal <- (goal (move ?b1) (on-top-of ?b2))
   (block ?b1)
   (block ?b2)
   (on-top-of (upper nothing) (lower ?b1))
   ?stack1 <- (on-top-of (upper nothing) (lower ?b2))
   ?stack2 <- (on-top-of (upper ?b1) (lower ?b3))
 =>
   (retract ?goal ?stack1 ?stack2)
   (assert (on-top-of (upper ?b1) (lower ?b2)))
   (assert (on-top-of (upper nothing) (lower ?b3)))
   (printout t ?b1 " move on top of " ?b2 "." crlf)
)
```

The CLIPS Program for Blocks World (cont.)

```
(defrule move-to-floor
   ?goal <- (goal (move ?b1) (on-top-of floor))
   (block ?b1)
   (on-top-of (upper nothing) (lower ?b1))
   ?stack <- (on-top-of (upper ?b1) (lower ?b2))
 =>
   (retract ?goal ?stack)
   (assert (on-top-of (upper ?b1) (lower floor)))
   (assert (on-top-of (upper nothing) (lower ?b2)))
   (printout t ?b1 " move on top of floor." crlf)
)
```

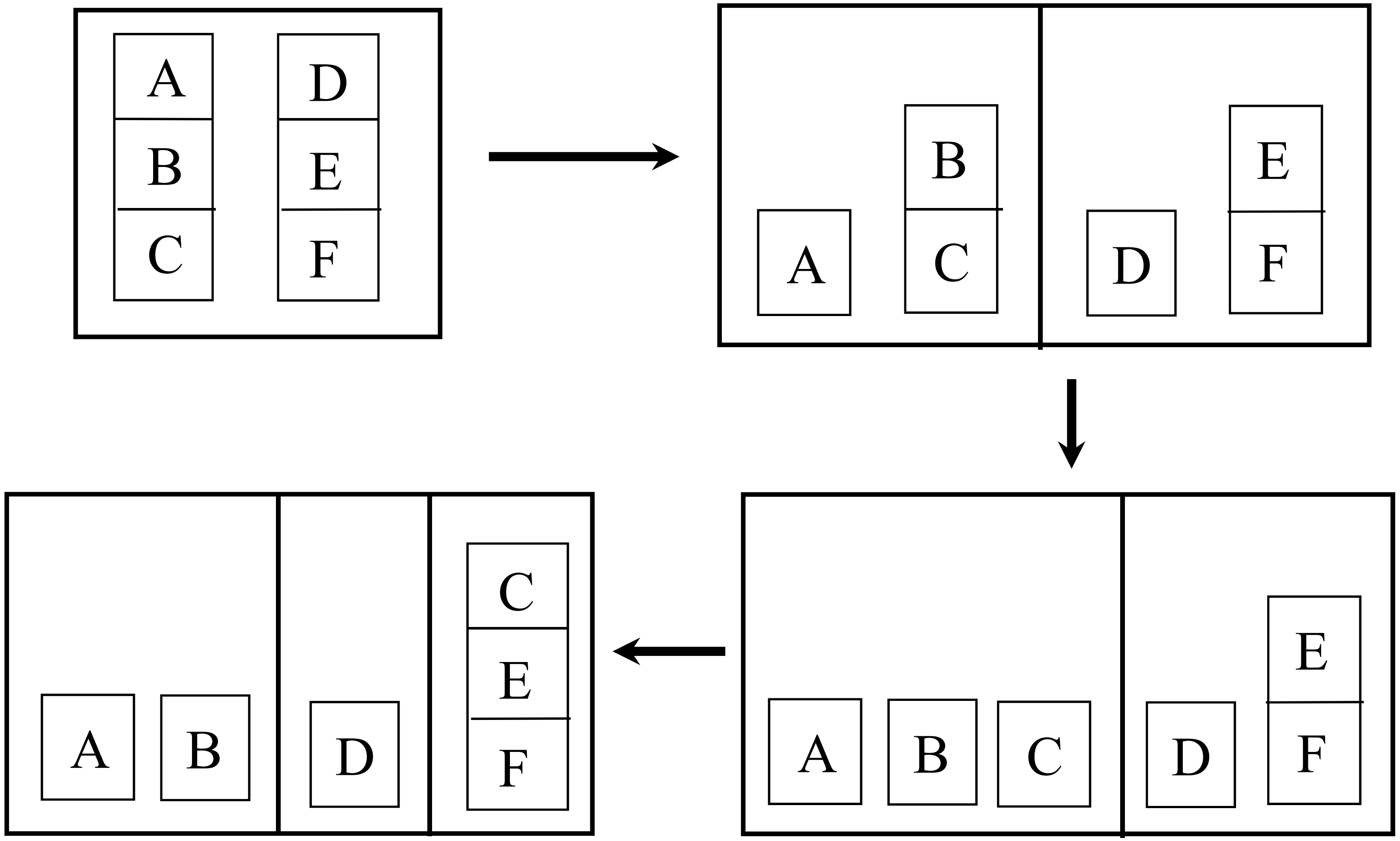
```
(defrule clear-move-block
   (goal (move ?b1) (on-top-of ?))
   (block ?b1)
   (block ?b2)
   (on-top-of (upper ?b2) (lower ?b1))
 =>
   (assert (goal (move ?b2) (on-top-of floor)))
)
```

```
(defrule clear-place-block
   (goal (move ?) (on-top-of ?b1))
   (block ?b1)
   (block ?b2)
   (on-top-of (upper ?b2) (lower ?b1))
 =>
   (assert (goal (move ?b2) (on-top-of floor)))
)
```

Results:

```
A move on top of floor.
B move on top of floor.
D move on top of floor.
C move on top of E.
```

Trace

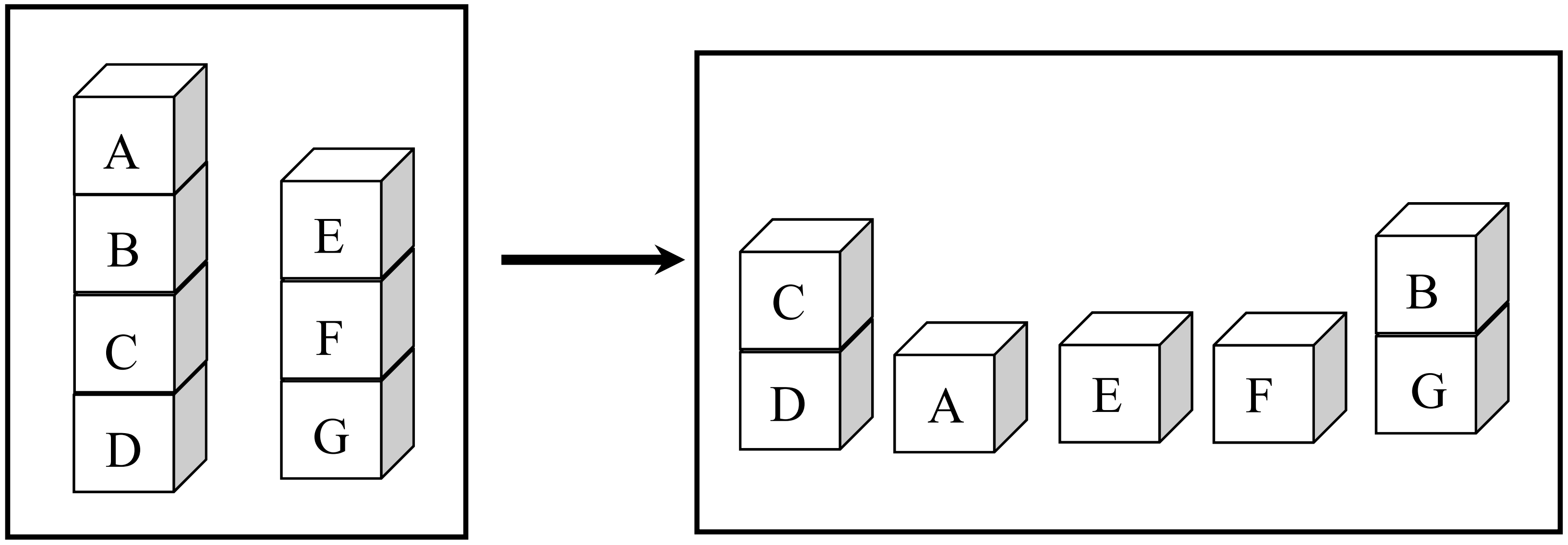


Trace (cont.)

Cycle	Stack-1	Stack-2	Goal	Rule
initial	ABC	DEF	C→E	
1			B→Floor D→Floor	clear-move-block clear-place-block
2			A→Floor	clear-move-block
3	~~ABC~~ A, BC		~~A→Floor~~	move-to-floor
4	~~BC~~ B, C		~~B→Floor~~	move-to-floor
5		~~DEF~~ D, EF	~~D→Floor~~	move-to-floor
6	~~C~~	~~EF~~ CEF	~~C→E~~	move-directly

Exercise

- goal: to move B on top of G
  - only trace is required



(Initial State) (Goal State)

Blocks World Revisited

- Using multifield variables and wildcards to re-implement the blocks world problem

```
(deftemplate goal (slot move) (slot on-top-of))
(deffacts initial-state
     (stack A B C)
     (stack D E F)
     (goal (move C) (on-top-of E))
)
(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 " move on top of " ?block2 "." crlf))
```

Blocks World Revisited (cont.)

```
(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 “ move on top of floor.” crlf))
```

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

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