The Control Flow in CLIPS

Opposite to conventional programming languages, where the control flow is fully explicit, CLIPS offers two levels of data-driven control:

- The core-level control, which corresponds to the built-in rule application mechanism;
- The user-level control that provides mechanisms for controlling the core-level engine.

Core-level control

The basic CLIPS machine is similar to the Markov Algorithmic Machine. The central part is a data structure called the **Agenda** which stores all the activation records for the rules which can be activated at a given moment in time. Therefore, the basic CLIPS control of rule application is data-driven. Indeed, observe that a rule can be activated by different groups of facts from the factual knowledge base of the running program. For example, taken the rule

```
(defrule test
      (fact $?a $?b) => (printout t $?a $?b crlf))
```

and the fact **(fact 1 2)**, the rule **test** can be activated in three different modes, each activation corresponding to a different binding of the variables **$?a** and **$?b**, as in the table below. A line from the table corresponds to an *activation record* of the rule **test**. At the same moment in time, there can be different activation records for the same rule and for different groups of facts that satisfy the patterns of the rule.

Activation	$?a	$?b
#1 for fact	**empty list**	**1 2**
#2 for fact	**1**	**2**
#3 for fact	**1 2**	**empty list**

From the activation records stored in the **Agenda** a single record has to be selected and the associated rule applied. When the rule is applied, the rule variables are subject to the bindings designated by the activation record. The algorithm that controls the **Agenda** is called the **Agenda_algorithm** and behaves as shown below.

```
Agenda_algorithm (Rules,Facts) {
   do {
         Agenda := rule activation records derived from matching
                   the Rules against the Facts;
         if Agenda is not empty
         then {
                sort Agenda decreasingly, according to the salience which
                             the activation records inherit from the rules;
                ar = the activation record from the top of Agenda;
                Agenda = Agenda - {ar};
                Facts = new facts resulted from the application of rule(ar);
                if rule(ar) is terminal then break;
         }
   } while (Agenda is not empty);
}
```

The **Agenda_algorithm** differs from the MAM control in some respects. In MAM the rules ordering in an algorithm is explicit, by force of the rule position. In CLIPS the ordering is implicit, according to the salience of the rules and, moreover, can be only partial and can change dynamically during the execution of the program. Moreover, CLIPS uses a conflict resolution strategy for selecting the rule to be applied, provided the **Agenda** stores several activation records with the same highest salience.

In MAM the traversal of the Data Register for selecting the beginning of a string in order to determine the applicability of a rule is left-to-right. In CLIPS the order in which the facts are matched against the patterns of a rule is random[1] (it certainly depends on the order of facts in the factual knowledge, subject to implementation).

In MAM a correct algorithm must terminate by executing a terminal rule. In CLIPS a program can terminate when the **Agenda** is empty and there are no rules to be applied (case that corresponds to the blocking of an MA algorithm).

In order to observe how close to the MAM control algorithm is the **Agenda_algorithm**, consider the CLIPS variant of the Markov algorithm for reversing a string of symbols. The algorithm is:

```
reverse(A); A g1,g2;
      1: ag1g2 -> g2ag1;
      2: ag1 -> bg1;
      3: abg1 -> g1a;
      4: a ->.;
      5: -> a;
end reverse
```

The CLIPS program is:

```
(deftemplate DR (multislot contents))

(defrule r1
   (declare (salience -1))
   ?f <-  (DR (contents $?s1 a ?g1&~a&~b ?g2&~a&~b $?s2))
   =>
   (modify ?f (contents $?s1 ?g2 a ?g1 $?s2)))

(defrule r2
   (declare (salience -2))
   ?f <-  (DR (contents $?s1 a ?g1&~a&~b $?s2))
   =>
   (modify ?f (contents $?s1 b ?g1 $?s2)))

(defrule r3
   (declare (salience -3))
   ?f <-  (DR (contents $?s1 a b ?g1&~a&~b $?s2))
   =>
   (modify ?f (contents $?s1 ?g1 a $?s2)))

(defrule r4
   (declare (salience -4))
   ?f <- (DR (contents $?s a))
   =>
   (modify ?f (contents $?s)) (halt))

(defrule r5
   (declare (salience -5))
   ?f <- (DR (contents $?s))
   =>
   (modify ?f (contents a $?s)))
```

There is a one to one correspondence between the MA and the CLIPS program. The only additional information in CLIPS is related to the domain of variables. For example, in the rule **r1**

[1] There is an implicit mechanism that will be discussed later, based on the age of facts, which can be used to simulate somewhat the left-to-right traversal of DR.

the values bound to the variables **?g1** and **?g2** are explicitly tested. They must be different from **a** and **b**, symbols that are "local variables" using the terminology of MA.

The major difference between the **Agenda_algorithm** and the MAM control is the *principle of refraction*. It stops the creation of the same activation record endlessly.

A rule can be applied once only for a given combination **F** of facts that makes the rule applicable and for a given set of bindings of rule variables against slot values of the facts **F**. As long as facts **F** do not change the rule is non-applicable for these facts.

Conflict Resolution Strategies

There can be several activation records with the same highest salience in the **Agenda**. For selecting the activation record to be applied CLIPS uses a *conflict resolution strategy*. Such a strategy can be explicitly enforced from a set of predefined strategies: depth, breadth, simplicity, complexity, lex, mea, and random. Only the simplest strategies are mentioned below, whereas the *mea* strategy is discussed in the next lecture. For all the other strategies see the CLIPS reference manual.

- According to the *depth* strategy, the activation record with the highest salience and that is the most recently created is selected. The **Agenda** behaves as a LIFO structure (a stack).

- The *breadth* strategy selects the activation record with the highest salience that is the oldest in the **Agenda**. The **Agenda** behaves as a FIFO structure (a queue).

- The *random* strategy selects randomly an activation record from the set of activation records with the same highest salience.

```
; The shortest path between two given vertices of a directed graph

(deftemplate edge (slot from) (slot to))
(deftemplate shortest_path (slot start) (slot stop))

(defrule initial_path
   (shortest_path (start ?x))
   =>
   (assert (path ?x))
   (set-strategy breadth)) ; Enforce the conflict resolution strategy
                           ; used by the Agenda_algorithm
(defrule extend_path
   (path $?n ?y)
   (edge (from ?y) (to ?z & ~?y & :(not (member ?z $?n))))
   =>
   (assert (path $?n ?y ?z)))

(defrule complete_path
   (declare (salience 10))
   (shortest_path (stop ?x))
   (path $?n ?x)
   =>
   (printout t crlf "shortest path " (create$ $?n ?x) crlf)
   (halt))

(defrule load_data
  =>
  (printout t "File: ")
  (load-facts (read)))
```

Some problems fit directly some predefined conflict resolution strategy. For instance, finding the path with fewest edges between two given vertices of a graph is solved by the simple program above, where the enforced conflict resolution strategy is *breadth*.

However, the CLIPS core-level control, including the conflict resolution strategies, must not be mistaken as being the problem solving control flow. A bunch of rules written at random may not lead to the correct result. The **Agenda_algorithm** is only the basis onto which the problem solving control flow is modeled using user-level control mechanisms. From this point of view it is better to consider that the selection of the applied rule is random and, therefore, to consider the **Agenda_algorithm** as non-deterministic.

User-level control

It would be quite difficult to think the control flow of a problem in terms of the scant **Agenda_algorithm**. The programmer should be able to enforce a rule activation strategy that fits the solving problem strategy. This task can be done in CLIPS in several ways:

- implicitly, via control facts;
- explicitly, by denying the application of the refraction principle for specific rules;
- explicitly, by modularizing the CLIPS program.

Using control facts

Assume that in a program a group of rules **Group1** has to be executed after a condition **Cond** holds. If **Cond** is not satisfied then another group of rules **Group2** has to be considered. Here, executed means testing the rules for application and applying the rules if possible. Therefore, what must be implemented is close to the conventional selection:

```
if Cond then Group1 else Group2
```

One way to do it is by control facts. Assume that the rule which tests the condition **Cond** asserts the fact **(Cond is true)** if the condition is satisfied and the fact **(Cond is false)** otherwise. Then the rules from **Group1** must include in their patterns the pattern **(Cond is true)**, while the rules from **Group2** must contain as an additional pattern **(Cond is false)**, as sketched below. Testing for **Cond** or **not Cond** directly within the rules is also possible but less efficient if **Cond** is complex.

```
(defrule testCond
       Cond => (assert (Cond is true))

(defrule testCond
       not Cond => (assert (Cond is false))

(defrule rule_from_Group1
       (Cond is true) ... other patterns
       => actions )

(defrule rule_from_Group1
       (Cond is false) ... other patterns
       => actions )
```

It is easy to see how the above mechanism works. However, for complex cases it leads to an overhead of control facts, and the housekeeping of these facts can clutter the program. A cleaner and powerful means to control the execution flow of a CLIPS program is via program modules.

Refreshed rules

A rule can trigger a specific action that unlocks the application of other rules that may be blocked due to the refraction principle. The action **(refresh $rule_1$ … $rule_n$)** unlocks the application of the rules **$rule_1$ … $rule_n$**. After being refreshed, rules can be applied for the same facts they were already applied. For example, in the short program below the rule **read_fact** is fired continuously until the end of input file is reached. The rule is equivalent to a loop the body of which reads data from a file.

```
; Variant 1
(deftemplate Book (multislot title) (multislot author) (multislot editor) (slot year))

(defrule get-input-file
 =>
     (printout t "Input file: ") ; ask for the name of the input file
     (bind ?file (readline))     ; read the file name and bind it to the variable ?file
     (open ?file data "r")       ; open for reading the logical file called data
     (assert (input_file: data)))

(defrule read_fact
     ?f <- (input_file: ?file)
 =>
    (bind ?fact (readline ?file))   ; read from data a fact contained on a single line
    (if (eq ?fact EOF)
        then (close ?file)          ;end of file, close the data file
             (retract ?f)
        else (assert-string ?fact)
             (refresh read_fact)))  ; read the next fact using the same rule

; Variant 2
(deftemplate Book (multislot title) (multislot author) (multislot editor) (slot year))

(defrule get-input-file
 =>
     (printout t "Input file: ") ; ask for the name of the input file
     (bind ?file (readline))     ; read the file name and bind it to the
variable ?file
     (open ?file data "r")       ; open for reading the logical file called data
     (assert (input_file: data)))

(defrule read_fact
     ?f <- (input_file: ?file)
 =>
    (assert (new-fact (readline ?file)))) ; read a fact  on a single line

(defrule test-eof
    ?f <- (input_file: ?file)
    ?g <- (new-fact EOF)
 =>
    (close ?file)
    (retract ?f ?g))

(defrule new-fact
    ?g <- (new-fact ?fact&~EOF)
 =>
    (assert-string ?fact)
    (retract ?g)
    (refresh read_fact))  ; read the next fact
```

Program modules

The factual knowledge base and, moreover, the rule base, of a CLIPS program can be split to form modules. By using modules, the factual knowledge base of a program can be selectively shared among different solving processes, each process corresponding to the activity of a module. The problem solving activity unwinds by being focussed on different modules. Although this technique may seem equivalent to the function or procedure calls from a conventional programming language, module-based processing bears significant differences.

We say that a module is active when its rules are effectively applied. A module is dormant when its rules are not applied, even if they are applicable.

Each module can import programming constructs from other modules and it can export programming constructs. For example, by importing a template **T** from a module **A**, another module **B** can "see" all the facts of module **A** which are instances of the template **T**.

Each module has its own **Agenda** that is *permanently active*, even when the module is dormant. In this way, a module **B** which "sees" facts from another module **A** is sensitive to the changes performed by **A**. In special cases, the module **B** can take control automatically and can become active when one of its rules can be fired due to external facts.

In order to manage the modules of a program CLIPS maintains a LIFO data structure, called the **focus-stack**. The module at the top of the **focus-stack** is active; all the others are dormant. The rules in a module can contain actions that modify the contents of the **focus-stack** and thus force the activation of other modules.

- **(focus M_1 M_2 ... M_n)** pushes the modules M_1 M_2 ... M_n at the top of the **focus-stack**. The new top module is M_1. Therefore, the module M_1 is becoming active, while the module that issued the command changes its status to dormant. However, the control is transferred to M_1 only after the current rule that issued the **focus** action completes its application.

- **(return)** pops the module at top of the **focus-stack**. The next module from the stack becomes active. It is interesting to note that there can be an implicit return from an active module: when the module becomes blocked, i.e. the module private **Agenda** is empty (no rule is applicable). Therefore, **return** activates the shallowest module in the **focus-stack** that has a nonempty **Agenda**. If the **focus-stack** gets empty during the operation, then the program terminates.

- **(clear-focus-stack)** pops all the modules from the **focus-stack**.

Each CLIPS program has a default module called **MAIN**. **MAIN** is always at the top of **the focus-stack** when a CLIPS program starts execution. In addition, it is this module that contains the **initial-fact**. The **initial-fact** should be exported by **MAIN** and imported by all modules that contain rules without patterns. Otherwise these rules could not be applied.

Auto-focus modules and daemons

An important feature of a module is its automatic activation. If the rule of a module **M** is declared auto-focus, by using the special pattern **(declare (auto-focus TRUE))**, then the module will be automatically activated (and inserted at the top of the **focus-stack**) when the auto-focus rule becomes applicable. If the module is already active (i.e. it is the module from the top of the

focus-stack) no action is taken. As an example, consider the two simple modules of the program below. The currently active module is **MAIN** when the rule **set_counter** is applied. The rule asserts the fact **(counter (value 10))** and then locks itself due to the refraction principle. Each new fact **(counter...)** activates automatically the module **PRINT** (that is pushed at the top of the **focus-stack**) and the rule **print_counter** prints the value of the counter. Since the **Agenda** of the module **PRINT** is emptied immediately, the module returns automatically to the next module in the **focus-stack**, i.e. to **MAIN**. The rule **decrease_counter** is now applied. A new fact **(counter...)** replaces the old fact and again the module **PRINT** gets control. These actions are repeated until the value of the counter is **0** and, therefore, no fresh **(counter...)** fact is produced.

```
; Module MAIN increments the counter
(defmodule MAIN (export ?ALL))

(deftemplate counter (slot value))

(defrule set_counter
   => (assert (counter (value 10))))

(defrule decrease_counter
   ?f <- (counter (value ?n&:(> ?n 0)))
   =>
   (modify ?f (value (- ?n 1))))
```

```
; Module PRINT prints the counter
(defmodule PRINT (import MAIN ?ALL))

(defrule print_counter
   (declare (auto-focus TRUE))
   (counter (value ?n))
   =>
   (printout t ?n crlf))
```

A module that automatically activates itself when its execution environment fulfils given conditions is called a daemon. The auto-focus mechanism can be used to program daemon-based applications. Details concerning the program modularizing and the control of the modules can be found in the CLIPS documentation.

An example of using daemons

Consider a program that analyzes a given set of ballots in an election campaign and decides the winner, if any. The structure of the program is illustrated in figure 1.

The **MAIN** module loads the factual knowledge base with the facts from a file.
The **VOTING** module counts the votes and updates the ranking of candidates.
The **RESULT** module decides the winner or the tie (the best candidates and their number of votes).

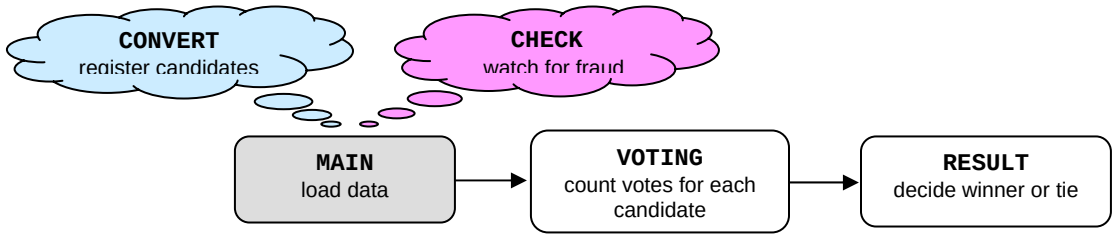

Figure 1. The structure of the ballot program

Apart from these modules, explicitly sequenced by the **focus** action from the rule **input_data** in module **MAIN**, the program contains two additional modules that are off the explicit control track: **CONVERT** and **CHECK**. Both are daemons. The use of daemons as data verifying and data conversion modules, as **CHECK** and **CONVERT** are used here, simplifies the structure of the program. The control of these two modules is fully data driven. There is no need to enforce restrictions on the order of data in the loaded file, and the data validating part can be neatly separated from the rest of the program.

The **CONVERT** module is automatically activated whenever a fact **(candidates c_1 ... c_n)** is created in **MAIN** as effect of loading the initial facts. The module generates specific facts for each of the candidates c_k: **(candidate (who c_k) (votes 0))**

The module **CHECK** checks for fraud. The ballots of any fellow that voted several times are invalidated: the corresponding facts are retracted and the fellow is registered on a black list. This module is automatically activated whenever:

- There are two distinct facts **(votes X Y)** and **(votes X Z)** where, eventually, **Y = Z**, and **X** is not on the black list. In this case the fact **(invalidate X)** is created and the two votes are deleted from the knowledge base.

- There is a fact **(invalidate X)** (i.e. **X** is on the black list) and a new fact **(votes X W)** is created (loaded from the input file) in the module **MAIN**. In this case the fact **(votes X W)** is retracted.

In order for being able to work with multiple identical facts (called duplicated facts in CLIPS slang), the rule **input_data** from **MAIN** uses the predefined CLIPS action:

(set-fact-duplication [TRUE | FALSE])

While the facts are loaded, fact duplication is allowed. Therefore, multiple facts such as **(votes Barney Fido) ... (votes Barney Fido) ...** can occur as distinct facts within the knowledge base. When the loading of the initial facts is completed the fact duplication is disallowed. Any attempt to create a fact that is identical with an existing fact is simply ignored.

In addition, observe that the module **MAIN** contains the dummy facts **(candidates)** and **(votes)**. These facts stand as declarations for implied templates. Otherwise these implied templates are not known and, therefore, cannot be exported from **MAIN**.

```
;_____________________________________________________________
(defmodule MAIN (export deftemplate ?ALL))
; Load facts: {(candidates c1...cn) crlf | (votes who whom) crlf }*

(deftemplate candidate (slot who) (slot votes))
(deffacts ff (candidates) (votes))

(defrule input_data
  =>
  (focus VOTING RESULT)
  (printout t "Input file: ")
  (set-fact-duplication TRUE)
  (load-facts (readline))      ; load facts from a file
  (set-fact-duplication FALSE))

;_____________________________________________________________
(defmodule CONVERT (import MAIN ?ALL))
; Converts a list of candidates into facts for each candidate
; setting the votes counter of the candidate to 0

(defrule register_candidate
  (declare (auto-focus TRUE))
  (candidates $? ?name $?)
  (not (candidate (who ?name)))
  =>
  (assert (candidate (who ?name) (votes 0))))
```

```
;_____________________________________________________________
 (defmodule CHECK (import MAIN ?ALL))
; Looks up for multiple ballots of the same voter or for
; ballots of an already "invalidated" voter

(defrule voting_twice
  (declare (auto-focus TRUE))
  ?f1 <- (votes ?x ?)
  ?f2 <- (votes ?x ?)
  (test (neq ?f1 ?f2))
  (not (invalidate ?x))
  =>
  (printout t ?x " voted several times" crlf)
  (assert (invalidate ?x)))

(defrule invalidate_votes
  (declare (auto-focus TRUE))
  (invalidate ?x)
  ?f <- (votes ?x ?)
  => (retract ?f))

;_____________________________________________________________
 (defmodule VOTING (import MAIN ?ALL))
; Count the (valid) ballots for the existing candidates

(defrule voting_for_non_candidate
  ?f <- (votes ?x ?y)
  (not (candidate (who ?y)))
  => (retract ?f))

(defrule voting_for_candidate
  ?f <- (votes ?x ?y)
  ?c <- (candidate (who ?y) (votes ?v))
  =>
  (modify ?c (votes (+ ?v 1)))
  (retract ?f))

;_____________________________________________________________
 (defmodule RESULT (import MAIN ?ALL))
; Decide if the elections ended in a tie or there is a winner
(deftemplate tie (slot votes))

(defrule ballot_winner
  (candidate (who ?x) (votes ?vx))
  (not (candidate (who ?y & ~?x) (votes ?vy & :(>= ?vy ?vx))))
  =>
  (printout t "Winner " ?x " with " ?vx " votes" crlf))

(defrule tie_detect
  (not (tie))
  (candidate (who ?x) (votes ?vx))
  (not (candidate (who ?y & ~?x) (votes ?vy &:(> ?vy ?vx))))
  (exists (candidate (who ?z & ~?x) (votes ?vx)))
  =>
  (printout t "Tie with " ?vx " votes:" crlf)
  (assert (tie (votes ?vx))))

(defrule tie_result
  (tie (votes ?v))
  (candidate (who ?c) (votes ?v))
  =>
  (printout t "   - " ?c crlf))
```

A session of executing the program is illustrated in the table below. It is interesting to trace the execution of the program, watching the change of **`focus`** (switching control from one module to another) and the moments when it effectively occurs.

Program execution	Data file
`CLIPS> (reset)` **`CLIPS> (run)`** **`Input file:`** **`H:\CPSC-432\CLIPS\Election.dat`** **`Barney voted several times`** **`Bibo voted several times`** **`Tie with 3 votes:`** **`   - Fred`** **`   - Fido`** **`CLIPS> (exit)`**	**`(candidates Bibo Fido Fred)`** **`(votes Bibo Fred)`** **`(votes Bibo Fido)`** **`(votes Fido Fred)`** **`(votes Fred Fred)`** **`(votes Lolo Fido)`** **`(votes Margo Fred)`** **`(votes Barney Fido)`** **`(votes Barney Fido)`** **`(votes Grumpy Fido)`** **`(votes Olive Popayee)`** **`(votes Alice Fido)`**