Non-Determinacy and Other Interesting Features of CLIPS

Non-determinacy is natural in many problems. It is, perhaps, the tradition of the conventional computing machines and, consequently, the explicit sequential control flow of the conventional programming languages that made us think in terms of ordered time steps the solution of our problems. The problems themselves might not require such tight control at every solving level as the sequential programs that solve them do. CLIPS departs from this custom and allows for non-determinacy.

Non-determinacy

Since, opposite to MAM, the rules of a CLIPS program are not ordered, there can be several rules that could be simultaneously applied. The sequential nature of the CLIPS basic machine restricts the application to a single rule. However, the applied rule can be randomly selected. Therefore, the order in which the rules with equal salience are applied may be not predicted. This feature can prove a trap for the programmer but, on the other hand, a potential parallel CLIPS machine can benefit from it.

In the program below the result differs in function of the application order of the rules. If the order is **r3,r2** the message printed is **"the problem cannot be solved"**; if rule **r1** is applied first the message is **"the problem can be solved"**. Note that these two contradictory results can be derived for the same initial data.

```
(deffacts facts (fact))

(defrule r1
       (fact)
       => (printout t "the problem can be solved" crlf)
          (halt))

(defrule r2
       (not(fact))
       => (printout t "the problem cannot be solved" crlf)
          (halt))

(defrule r3
       ?f <- (fact)
       => (retract ?f)
```

The program is in error. In fact, the problem itself is bogus, containing the following contradictory statements:

- if there is a fact **(fact)** then the problem is solved;
- if there is a fact **(fact)** the fact should be deleted;
- if there is no **(fact)** the problem cannot be solved.

For well specified problems non-determinacy leads to simpler programs. Consider the problem of finding whether a given equivalence relation over a finite set of people is total. A relation **R** over a set **A** is a subset of pairs **(x,y)** from **A×A**. The relation is an equivalence relation if it is reflexive, symmetric and transitive. The relation is total over **A** if any pair **(x,y)** from **A×A** is in **R**.

The set of people is described by facts of the form **(people (name *aName*))** that are instances of the template **(deftemplate people (slot name))**. The relation is specified as a set of facts of the form **(knows (who *aName*) (whom *anotherName*))** that are instances of the template **(deftemplate knows (slot who) (slot whom))**. First, the module **EQUIVALECE** expands the

given relation **knows**, using the transitivity and symmetry properties. Finally, the module **MAIN** tests if the relation is total, i.e. for each possible pair **(people (name a))**, **(people (name b))**, there is a fact **(knows (who a) (whom b))**. There is a primitive pattern-match operator, called **forall**[1], which does exactly this kind of test.

```
(defmodule MAIN (export deftemplate ?ALL))

(deftemplate knows (slot who) (slot whom))
(deftemplate person (slot name))

(defrule start
      => (printout t "Input file: ")
         (load-facts (read))
         (focus EQUIVALLENCE))

(defrule partial_equivalence_relation
      (person (name ?x))
      (person (name ?y))
      (not (knows (who ?x) (whom ?y)))
      =>
      (printout t "knows is a partial equivalence relation over people" crlf)
      (halt))

(defrule total_equivalence_relation
     (forall (and (person (name ?x)) (person (name ?y)))
             (knows (who ?x) (whom ?y)))
     =>
     (printout t "knows is a total equivalence relation over people" crlf)
     (halt))

;_____________________________________

(defmodule EQUIVALLENCE
     (import MAIN deftemplate ?ALL))

(defrule transitive-relation
      (knows (who ?x) (whom ?y))
      (knows (who ?y) (whom ?z))
      (not (knows (who ?x) (whom ?z)))
      =>
      (assert (knows (who ?x) (whom ?z))))

(defrule symmetrical-relation
      (knows (who ?x) (whom ?y&~?x))
      =>
      (assert (knows (who ?y) (whom ?x))))
```

All rules in the program have the same salience. Certainly the rules from the module **MAIN** are mutually exclusive, due to their patterns, and initially, due to the empty factual base (only the rule **start** can be applied). However, the rules in the module **EQUIVALENCE** are not mutually exclusive. There can be several activation records for the same rule and/or for different rules that can be simultaneously fired (i.e. the designated rules can be simultaneously applied). The order in which these rules are applied is irrelevant. The solution will be always correct.

Here, non-determinacy can be used for optimizing the execution of the program by concurrently applying the rules. From this point of view, the **EQUIVALENCE** module behaves as a true non-

[1] **(forall p_1 p_2 ...p_n)** is equivalent to **(not (and p_1 (not (and p_2...p_n))))**, where **p_k, k=1,n** are patterns.

deterministic algorithm with positive effect on the complexity of the overall solving process. For a parallel execution the complexity would be reduced from **O(n^2)** to **O(n)**.

Besides the problems that have unique solutions, as the **equivalence** problem does, there are problems that accept several correct solutions. For problems of this kind, the corresponding CLIPS program can compute different valid solutions according to the order in which rules are fired. Such a program - for the well-known *blocks world problem* - is considered later. On the other hand, the **equivalence** example does not rule out those problems where sequential execution is a must, at least partially. It shows only that there are problems where non-determinacy is natural, and that CLIPS is able to make use of this feature.

Declarative versus imperative programming style

Often a CLIPS rule explains what is meant by a certain relationship or by a specific object of the problem. It thus states what are the constraints an object or a relationship must satisfy in order to be created (asserted), modified or destroyed (retracted). The rule does not state how to fulfil these constraints and how to built a particular object that fulfils them. It simply declares what has to be done or what the object or the relationship is. The programming paradigm based on these characteristics is said *declarative* in opposition to the *imperative* programming, which focuses on how to test and enforce a given set of constraints and how to build objects that satisfy the constraints.

As an example, consider that the results of a ballot are represented by instances of the template **(deftemplate candidate (slot name) (slot votes))**. The problem is to find the winner, provided there is one, and to test whether the ballot ended in a tie. The rules below are not procedures to compute the winner or to test for a tie outcome. They are definitions of the two concepts. The rules read:

- **ballot_winner**: a candidate **?x** such that there is no other candidate with an equal or greater number of votes.

- **ballot_tie**: when there is a candidate **?x** with a maximum number of votes (i.e. there is no other candidate with more votes) and, moreover, there exists another candidate with an equal number of votes.

```
(deftemplate candidate (slot name) (slot votes))
(deftemplate ballot_winner (slot name) (slot votes))

(defrule ballot_winner
      (candidate (name ?x) (votes ?vx))
      (not (candidate (name ?y&~?x) (votes ?vy&:(>= ?vy ?vx))))
      =>
      (assert (ballot_winner (name ?x) (votes ?vx))))

(defrule ballot_tie
      (candidate (name ?x) (votes ?vx))
      (not (candidate (name ?y&~?x) (votes ?vy&:(> ?vy ?vx))))
      (exists (candidate (name ?y&~?x) (votes ?vy&:(= ?vy ?vx))))
      =>
      (assert (ballot_tie)))
```

Saying *what* instead of *how* can allow for 'definitions' as those above to be processed in an unspecified order and, eventually, concurrently. Declarative programs exhibit naturally a certain degree of implicit non-determinacy. In an imperative program the groups of statements that can be executed in any order have to be specified explicitly.

Certainly, the ballot example favors the declarative programming style. Nevertheless, there is no pure programming ideal that can be attained without a price to pay. Here the price for the declarative style is time and space complexity. This is the reason why any practical CLIPS program will have some degree of imperative programming style enforced, if not by the nature of the problem, at least by efficiency considerations.

Take, for instance, the problem of generating the full set of **k**-combinations of an **n**-set. An **n**-set is a set with **n**-elements. A **k**-combination of an **n**-set **S** is a **k**-subset of **S**. For example the **3**-combinations of the set **{a,b,c,d}** are **{a,b,c}**, **{a,b,d}**, **{a,c,d}**, and **{b,c,d}**. The declarative CLIPS program is remarkably simple.

```
; An intractable declarative program to compute combinations
; The space complexity is Ω(n!)

(deffacts data (set a b c d e)
               (combination_length 3))

(defrule k-combination
      (combination_length ?k)                    ; generate a k-combination as a
      (set $? $?x&:(= (length$ $?x) ?k) $?)      ; k-subset of the given set
      (not (combination $?y &:(subset $?x $?y))) ; see if already generated
      =>
      (assert (combination $?x))
      (printout t $?x crlf))

(defrule set-permutation
      (set $?x ?a $?y ?b $?z)
      =>
      (assert (set $?x ?b $?y ?a $?z)))
```

Pattern matching is used here to extract **k**-subsets from the given **n**-set. Since the pattern matching is linear, there will be subsets of the **n**-set impossible to extract. For example, if the **n**-set is **(set a b c d e)** the subset with the elements **a c e** could never be extracted since its elements are not contiguous in the fact that represents the set. In order to defeat the linear pattern matching the program generates set permutations, constructing all possible contiguous tuples with **k**-elements. This is the reason why the complexity is huge. Such a program is useless. A more efficient variant is less declarative.

The program below adopts an imperative style to compute **k**-combinations. It starts with **1**-sets and step-wise extends these sets, by adding elements from the initial set, until they grow to disjoint **k**-sets. The program is no longer close to the definition of a **k**-combination. Instead, it shows how the **k**-combinations can be computed starting from the **1**-combinations. The program is clumsier, but more efficient. The space complexity is **Comb(n,1)+…+Comb(n,k)= O(2^n)<O(n!)**

```
; A better imperative program to compute combinations
(deffacts data (set a b c d e)
               (combination_length 3))

(defrule initialize_combination
      (set $? ?x $?) => (assert (combination ?x)))

(defrule extend_combination
      (combination_length ?k)
      (combination $?x&:(< (length$ $?x) ?k))
      (set $? ?e&:(not (member$ ?e $?x)) $?)
      (not (combination $?y&:(and (subset $?y (create$ ?e $?x))
                                  (subset (create$ ?e $?x) $?y))))
      =>
      (assert (combination ?e $?x)))
```

```
(defrule print
      (combination_length ?k)
      (combination $?x&:(= (length$ $?x) ?k))
      =>
      (printout t $?x crlf))
```

The space complexity can be improved further by extending the sets in stages. At stage **i**, **i=1,k-1**, the **i**-sets are extended to form **i+1**-sets. Once all the disjoint **i+1**-sets are build the **i**-sets are deleted.

Temporary Variables

Variables in CLIPS are quite different from variables in conventional programming languages. In conventional programming variables are of paramount importance for data structuring, for the representation of problem universe and, finally, for the computation process itself. In a C program, all useful data are accessed starting from variables (static or dynamic). Storage is allocated by binding variables to pointers and reclaimed starting from the pointers stored in variables. Processing is performed using the values bound to program variables.

In CLIPS the landscape is different. Here the bulk of data that represents the problem universe and the auxiliary solving objects lives by its own. There are no global variables to store pointers to the facts that make up the factual knowledge base of the program. Variables are used only in rules, to constrain the pattern matching and to select parts from the matched facts. Variables have only a temporary role, behaving as the parameters of a function or procedure in a conventional programming language. From this point of view a rule is a procedural abstraction, applied by using a pattern-matching mechanism.

The use of variables in CLIPS, compared to conventional programming, has direct impact on the way the programmer thinks the solution of a problem. There is no need to fragment the problem universe using data structures assigned as values to the program variables and there is no need to pay attention to the a complex pointer-based inter-dependence of these fragments. In CLIPS (as in MAM) the problem universe is represented as a whole, in a uniform and symbolic manner. The program is able to see the factual base in its entirety.

Recursion

Recursion is a basic and natural way of thinking the solution of a problem. Many problems have elegant recursive solutions. How is it possible to use recursion in the realm of rule-based associative programming? For example, in which way the following C Fibonacci function (simple and *inefficient*) could be converted to an *efficient* CLIPS program?

```
int F(int n) {return n < 2 ? n : F(n-1)+F(n-2);}
```

The CLIPS program that corresponds to the **F** function is given below. It avoids redundant computation cacheing the results of **F(x)** as soon as these are computed, as suggested in figure 1.

The program uses an additional interesting control mechanism present in CLIPS: the **Agenda** conflict solving strategy. The activation records in the **Agenda** can be ordered not only according to the salience of the rules (taken as the primary sorting key), but in addition - if explicitly specified - according to a secondary key which takes into consideration the *age* of the facts. Each fact has an age, i.e. the time elapsed from its creation up to the current **Agenda** cycle. There are predefined **Agenda** conflict solving strategies that sort **Agenda** differently in function of this secondary key. The program above uses the **mea** conflict resolution strategy. It is an abbreviation

for Means-Ends Analysis, a general problem solving strategy used in Artificial Intelligence. If the strategy is explicitly set using the action **`(set-strategy mea)`**, the Agenda is ordered as follows:

1. The activation records are first sorted in descending order according to the salience of the rules; the rules with equal salience form a group.

2. Each group of records is sorted in ascending order according to the age of the facts that match the first pattern from each rule.

The rule that fires corresponds to the first activation record of the top group in the **Agenda**. It has the highest salience and its first pattern matches the youngest fact from the facts matched by the other rules in the same group.

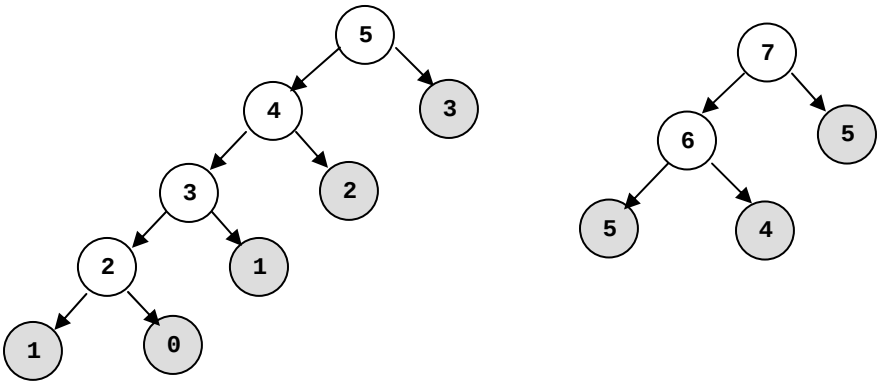

Figure 1 (a) The computation of F(5), (b) The computation of F(7) after F(5)
White nodes stand for effective computations
Grey nodes designate cached results of former computations

```
(defmodule MAIN (export ?ALL))

(deftemplate Fibonacci (slot of) (slot is))
(deffacts F01 (Fibonacci (of 0) (is 0))
              (Fibonacci (of 1) (is 1)))

(defrule init
   (declare (salience 20))
   => (set-strategy mea))

(defrule get_parameter
    (not (F $?))
    =>
    (printout t "Parameter of F: ")
    (assert-string (str-cat "(F " (readline) ")" )))

(defrule cached_result
   ?f <- (F ?x)
   (Fibonacci (of ?x) (is ?a))
   =>
   (printout t "F(" ?x ")= " ?a crlf)
   (retract ?f))

(defrule compute
   ?f <- (F ?x)
   (not (Fibonacci (of ?x)))
   =>
   (focus F_computation))
```

```
(defmodule F_computation (import MAIN ?ALL))

(defrule result_already_computed
   ?f <- (F ?x)
   (Fibonacci (of ?x))
   =>
   (retract ?f))

(defrule compute_F_of_x
   ?f <- (F ?x)
   (not (Fibonacci (of ?x)))
   (Fibonacci (of =(- ?x 1)) (is ?a))
   (Fibonacci (of =(- ?x 2)) (is ?b))
   =>
   (retract ?f)
   (assert (Fibonacci (of ?x) (is (+ ?a ?b))))
   (printout t "F(" ?x ")= " (+ ?a ?b) crlf))

(defrule recursive_call
   (declare (salience -5))
   (F ?x)
   (not (Fibonacci (of ?x)))
    =>
   (assert (F (- ?x 2)) (F (- ?x 1))))
```

As a matter of fact, the **Agenda** control strategies stand for yet another data-driven control mechanism that uses a natural data attribute: the age. To see how the **mea** strategy works, take the rule:

```
(defrule compute_F_of_x
      ?f <- (F ?x)
      (not (Fibonacci (of ?x)))
      (Fibonacci (of =(- ?x 1)) (is ?a))
      (Fibonacci (of =(- ?x 2)) (is ?b))
      =>
      (retract ?f)
      (assert (Fibonacci (of ?x) (is (+ ?a ?b))))
      (printout t "F(" ?x ")= " (+ ?a ?b) crlf))
```

Assume that there are two facts **(F 3)** age 20 and **(F 2)** age 10. Also, consider that there are facts that satisfy the remaining patterns of the rule. In this case, there will be two activation records in the **Agenda** for the rule. Their order is as follows:

First activation record -> for **(F 2)** age 10
Second activation record -> for **(F 3)** age 20

Therefore the rule is first applied for the younger fact **(F 2)** age 10 and later on for the older fact **(F 3)** age 20.

In what follows, the **Agenda** control strategy is used to force a recursive behavior of a set of rules. To see how it works, assume the goal of the Fibonacci program is **(F 5)**, i.e. we have to compute **F(5)**.

If the result of the function has been cached on a previous call, (i.e. if the fact **(Fibonacci (of 5) (is 5))** is present in the factual base of the program) the rule **cached_result** is applied: the result is printed, the goal **(F 5)** is retracted, and the computation of **F(5)** ends.

If the result of the function has not been cached on a previous call (i.e. if the fact **(Fibonacci (of 5) (is 5))** is missing from the factual base of the program) the rule **compute** is applied and the module **F_computation** becomes active.

The two main rules that perform the computation are **compute_F_of_x** and **recursive_call**. The rule **compute_F_of_x** has precedence over **recursive_call** and, if the Fibonacci numbers for **(F 4)** and **(F 3)** exist, the result of **(F 5)** is immediately computed, cached as the fact **(Fibonacci (of 5) (is 5))**, and printed. The goal is retracted and the computation of **(F 5)** ends.

If the rule **compute_F_of_x** cannot be applied (one or both partial results are missing) then the rule **recursive_call** is applied. The rule is interesting since it generates two goals: **(F 4)** age A_{F4} and **(F 3)** age A_{F3}, $A_{F3} < A_{F4}$. The module **F_computation** behaves as if it would be invoked recursively to compute **F(3)** and **F(4)**.

```
CLIPS> (reset)
CLIPS> (run)
Parameter of F: 5
F(2)= 1
F(3)= 2
F(4)= 3
F(5)= 5
Parameter of F: 7
F(6)= 8
F(7)= 13
Parameter of F: 5
F(5)= 5
Parameter of F:
CLIPS> (exit)
```

Figure 2. Executing the Fibonacci program

The rule **compute_F_of_x**, which does the effective computation, cannot be applied for **(F 5)** age A_{F5} as long as there will be facts of the form **(F x)** age A_{Fx} younger than A_{F5}. But this means implicitly that **x < 5**, and it follows that **(F 5)** age A_{F5} cannot be applied as long as all the values for **F(x)**, **x < 5**, are not computed. Similarly, the rule **compute_F_of_x** cannot be applied for facts of the form **(F x)** age A_{Fx} older than A_{F5}, unless **F(5)** is already computed and therefore the fact **(F 5)** is retracted. What we've obtained is a LIFO sequence of activation records for the rule **compute_F_of_x**. The sequence is ordered increasingly according to the age of the facts **(F x)**. These facts correspond to the recursive calls for the computation of **F(x)**. This is what we wanted. We have implemented a stack of activation records that simulate the call of a recursive function.

As far as the rule **result_already_computed** is concerned, it retracts the goals **(F ?x)** which have a cached result. Consider that the current goal is **(F 4)** and that the largest result cached is **(Fibonacci (of 2) (is 1))**. The goal **(F 4)** generates two "recursive goals" **(F 3)** and **(F 2)**. Only **(F 3)** has to be processed; the goal **(F 2)** is destroyed by the rule **result_already_computed**.

An example of program execution is shown in figure 2. It proves that indeed the program avoids the repeated computation of the same result throughout its use. Initially, for **(F 5)** the program computes **(F x)**, **x≤5**, for **(F 7)** it computes only **(F x)**, **x≥6**, based on the results cached etc.

Obviously, the mechanism used to define the **Fibonacci** recursive function can be generalized for any recursive function. For a primitive recursive function

```
F(0)= r
F(n+1) = G(n,F(n))
```

the general CLIPS mechanism used to describe the function is illustrated in figure 3.

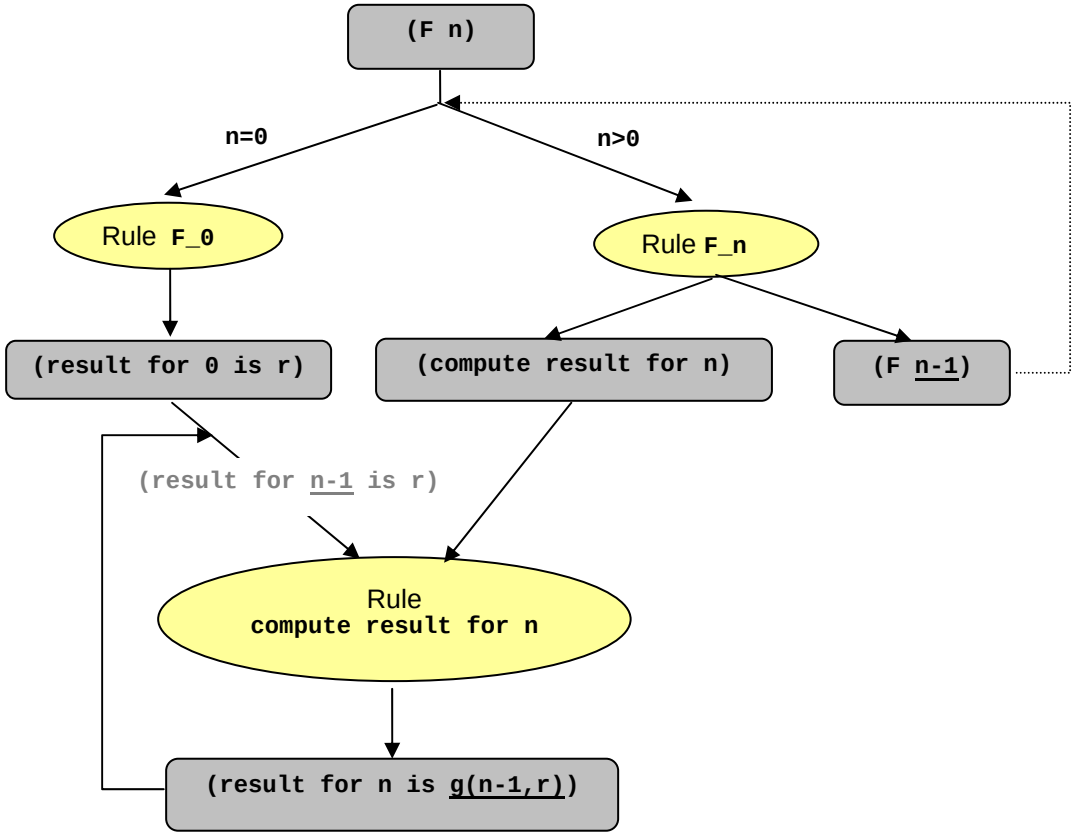

Figure 3. Computing a primitive recursive function in CLIPS

As a direct application of the above mechanism, consider the computation of **n!**. The program works precisely as the recursive function **factorial(x)** would be computed in an imperative programming language. The partial results are not cached.

```
; factorial(0) = 1
; factorial(n) = n * factorial(n-1)

(defrule init
   (declare (salience 20))
   =>
   (set-strategy mea))

(defrule get_param
    (not (factorial $?))
    =>
    (printout t "Your number: ")
    (assert-string (str-cat "(factorial " (readline) ")")))
```

```
(defrule factorial_of_0
    ?f <- (factorial 0)
    =>
    (assert (result for 0 is 1))
    (retract ?f))

(defrule factorial_of_n
    ?f <- (factorial ?n & :(> ?n 0))
    =>
    (assert (compute_result for ?n)
            (factorial (- ?n 1)))
    (retract ?f))

(defrule compute_result
    ?f <- (compute_result for ?n)
    ?g <- (result for ?m is ?r) ; ?m is precisely ?n-1
    =>
    (assert (result for ?n is (* ?n ?r)))
    (retract ?f ?g))

(defrule print_result
     (not (compute_result for ?))
    ?f <- (result for ?n is ?r)
    =>
    (printout t ?n "!= " ?r crlf crlf)
    (retract ?f))
```

```
Your number: 3
==> f-1     (factorial 3)
==> f-2     (compute_result for 3)
==> f-3     (factorial 2)
<== f-1     (factorial 3)
==> f-4     (compute_result for 2)
==> f-5     (factorial 1)
<== f-3     (factorial 2)
==> f-6     (compute_result for 1)
==> f-7     (factorial 0)
<== f-5     (factorial 1)
==> f-8     (result for 0 is 1)
<== f-7     (factorial 0)
==> f-9     (result for 1 is 1)
<== f-6     (compute_result for 1)
<== f-8     (result for 0 is 1)
==> f-10    (result for 2 is 2)
<== f-4     (compute_result for 2)
<== f-9     (result for 1 is 1)
==> f-11    (result for 3 is 6)
<== f-2     (compute_result for 3)
<== f-10    (result for 2 is 2)
3!= 6

<== f-11    (result for 3 is 6)
Your number:
```

Figure 4. A CLIPS trace of computing **`(factorial 3)`**

Automatic management of data relationships

In conventional programs, managing data relationships is the programmer's task. The program states explicitly when objects are to be created or deleted as requested by the logic of the problem. CLIPS is equipped with a simple but effective mechanism that enables the programmer to automatically keep track of logical relationships between the facts of the problem. The mechanism is a very simple form of TMS (Truth Maintenance System) borrowed from the AI programming technology. The mechanism is based on the idea that each fact in the factual knowledge base of a program has a logical support that justifies its existence. Once the logical support is retracted the supported facts have no reason to exist in the factual base. In other words, a fact can be logically dependent on other facts (its logical support) and its lifetime is dependent on the structure and the life span of the logical support.

As an example, take the following statements: I have a millionaire aunt and this makes me rich; with no such aunt I will be rich no more. The fact **(I am rich)** is a consequence of the existence of the fact **(I have a millionaire aunt X)**. Moreover, there is a cause-effect relationship between these two facts: once the aunt - cause - vanishes, the fact **(I am rich)** - the effect - has no justification to exist and should be automatically retracted from the knowledge base of the program. This is what happens in CLIPS if the logical dependence between the two facts above is declared in the following way:

```
(defrule easy-life
          (logical (I have a millionaire aunt X))
       => (assert (I am rich)))
```

The facts that match the patterns from the **logical** clause of a rule provide logical support to the facts asserted by the rule. They form a support group of the asserted fact. A fact can be logically supported by more than one group of facts from different rules. If any one supporting fact is removed (and there are no other supporting groups), then the supported fact is retracted automatically. A fact created without logical support is said to have unconditional support. It has to be removed explicitly by using the **retract** action. Thinking in terms of propositional calculus, the following conventions apply:

- The logical support group G_k of a fact **F** corresponds to the formula: $G_k = F_{k,1}$ **and** $F_{k,2}$ **and ...** $F_{k,nk}$, where $F_{k,j}$ are the facts of G_k.
- If the fact **F** has several logical support groups G_k, **k=1,m**, then the logical support of **F** corresponds to the formula **support(F) =** G_1 **or** G_2 **or ...** G_m.

As far as the formula **support(F)** is true, the fact **F** is not retracted automatically from the program factual base (certainly, it can be retracted explicitly). If the formula **support(F)** is false, the fact **F** is automatically retracted.

```
(defrule easy-life1
       (logical (I have a millionaire aunt X)
                (I have a millionaire aunt Y))
      =>
      (assert (I am rich)))

(defrule easy-life2
      (logical (I have a millionaire aunt Z))
      =>
      (assert (I am rich)))

 support(I am rich) =
     ((I have a millionaire aunt X) and (I have a millionaire aunt Y)) or
     (I have a millionaire aunt Z)
```

In the example above, if the fact **(I have a millionaire aunt Z)** is retracted, still the formula **support(I am rich)** is true and the fact **(I am rich)** is not retracted. If the facts **(I have a millionaire aunt X)** and **(I have a millionaire aunt Z)** are both retracted then the formula **support(I am rich)** is false and the fact **(I am rich)** is automatically retracted.

Each time an existing fact is re-asserted, its logical support is updated, even if the fact is not effectively inserted in the factual base of the program. The logical support of facts can be used:

- to enforce the consistency of the factual knowledge base of the problem (what is meant by consistency depends on the problem);
- to perform an automatic sort of garbage collection, retracting facts that are no longer useful.

Efficiency as an implementation issue

From the efficiency point of view the declarative programming style can run into serious trouble. The example of the **k**-combinations problem is relevant. In CLIPS efficiency is not only a matter of the quality of the problem solving method but, in addition, it is strongly influenced by the pattern-matching algorithm. Pattern matching is by far the critical operation of the system. There are two implementation approaches to pattern matching:

- rule-directed pattern matching;
- data (facts) controlled pattern matching.

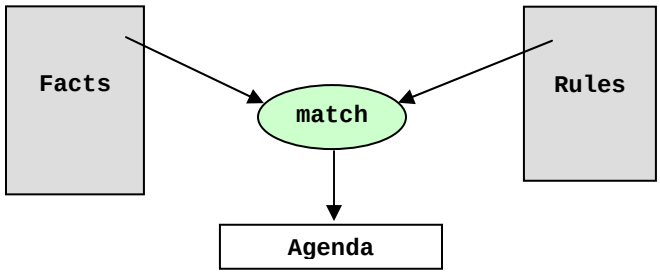

Figure 5. Rule-directed pattern match

The rule-directed pattern matching process matches the patterns of each rule in the program against all relevant combinations of facts in the factual knowledge base. The matching process is performed for each cycle of the **Agenda_algorithm**. The behavior of rule-directed pattern match is sketched in figure 5. For example, consider the rule

(defrule R $pattern_1$ **...**$pattern_n$ **=>** ***actions*****)**

and assume that there are N_k facts that can match $pattern_k$. The time necessary to construct the activation records for the rule **R** is $\Theta(\prod_1^n N_k)$ and for **M** similar rules it is $\Theta(M\prod_1^n N_k)$. If each pattern has multi-fields that can be matched in **q** different ways each, then the complexity grows to $\Theta(Mq^n\prod_1^n N_k)$. For $N_k=100$, $n=5$, $q=10$, $M=100$ the pattern match complexity is $\Theta(10^{17})$. Rule-driven pattern matching is not a viable implementation choice.

A closer look at the **Agenda_algorithm** shows that in an **Agenda** cycle only one rule is fired and the modifications of the factual base are usually minor. The data-oriented pattern match takes advantage of the locality and limitation of factual base modifications that span an **Agenda** cycle. It saves the results of the pattern matching process of the current **Agenda** cycle to the next cycle and updates these results according to the actions of the currently fired rule. The pattern match process selects the rules influenced by the current modifications of the factual knowledge base and performs the effective pattern match of these rules against the newly asserted facts or against the modified facts only, as suggested in figure 6. Significant time is saved in this way. The price to pay for data-driven pattern matching is the additional memory necessary to keep the matching information from one **Agenda** cycle to the next cycle.

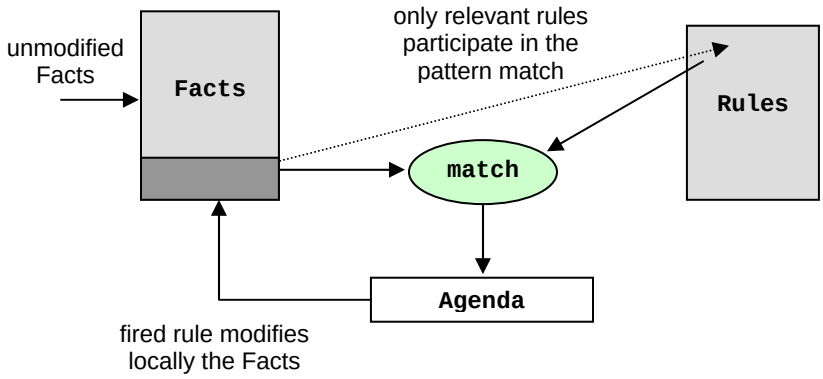

Figure 6. Data-directed pattern matching

There is an interesting algorithm, called the RETE algorithm, designed for data-driven pattern matching. It uses a data structure that can be seen as two nets:

- The template net which groups the patterns of all the rules in the program. It takes advantage of the similarities between the patterns of the program rules to save space and to increase the pattern match speed.

- The join net, which groups the activation records of rules. It speeds up the decision on what rules are applicable and for what subsets of facts.

The template net keeps the results of a partial pattern match (without bindings of the pattern variables). If the facts that govern the matching process of a specific pattern are not modified, the matching process of that pattern is not performed any longer in the current **Agenda** cycle. The information form the previous cycle is already stored in the template and join nets.

The join net contains entries for each rule, storing back-pointers to the template net and saving the current bindings of the pattern variables for each pattern of the rule. Again, no extra-processing is necessary if the facts that matched the patterns of the rule did not change. The variables of the each activation record of the rule preserve their previous **Agenda** cycle bindings. Moreover, nodes in the join net may be shared by rules with similar patterns. The algorithm (called RETE) and its elaborate data structures are discussed in:

Giarratano Joseph and Riley Gary, *Expert Systems, Principles and Programming*, PWS Publishing Company, 1994