Intel.ligència Artificial

Enginyeria en Informàtica

CLIPS - Code Snippets

Versión 0.8

Departament de Llenguatges i Sistemes Informàtics

Facultat d'Informàtica de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

CURS 2010/2011 1Q

Contents

1. Programming with rules

1.1 Introduction

To use a language based on logic like Prolog or CLIPS needs a change in the way of thinking about how to do programming. The fundamental difference is that these programming language declare what is a solution but do not explicitly tell how to obtain it, this is the reason because they are called declarative languages.

The reason to program this way is because they implement an inference engine that uses automatic proof mechanisms to find the solution that satisfies the conditions specified by the program.

This means that sometimes if we want a program to find a solution efficiently, we should think about how the inference engine works and specify the conditions of the program accordingly. Moreover, such languages usually include control structures that allow to modify the way in which the inference engine searches for the solution to obtain efficiency and to give some control to the programmer. Obviously, the way of thinking the programs may depend on how the inference engine works.

In languages with an inference engine that uses backwards reasoning, the most effective approach to program is to think that what you want to do is a decomposition of problems. The rules specify the conditions to be met to solve each subproblem. The inference engine will explore all possibilities and check whether there is a decomposition that meets the problem data.

The type of questions to be solved with this type of program is always well established and usually follows the pattern "It is this set of facts a problem of type X?"

In languages with an inference engine that uses forward reasoning the approach is different, mainly because they solve problems in which the goal is not known. The rules seek to obtain all the possible consequences until a set of facts that satisfies certain conditions appears.

The type of questions to be solved with this type of programs is far more open, since in fact it is not known a priori what is the nature of the response. We are exploring what possible responses we can get from the facts introduced waiting to see one that satisfies us.

Obviously when creating this kind of programs it is important to have more control of what is explored and how is explored. Fortunately this type of reasoning allows to do this.

1.2 The inference engine

It is important to think that unlike imperative languages the control of execution is not imposed by us, but by the inference engine. So, except for some control structures that the language could have, we do not know exactly how the program will be executed.

In fact is the conflict resolution strategy of the inference engine the one to make those decisions. We have to think that in this type of programming we do not decide the sequence of instructions that the program will follow, but it will be decided depending on the characteristics (facts) of the problem.

We could think of this type of programming as a puzzle in which we create the pieces (rules) and they are automatically assembled to create the program that solves our problem.

We must also have in mind that, being aware of the conflict resolution strategy of the inference engine can influence how we program. Basically we can decide how are represented the conditions in each rule or the order in what they appear to favor a more efficient exploration of the solution.

We should therefore put aside our ideas from imperative programming, we do not control anything (or very little).

This that in principle could seem bad, in fact allows to express more efficiently and concise programs that would be quite complex for imperative programming. We are giving up control to have more level of abstraction, to worry more about the *what* and less about the *how*.

Obviously, not all problems can be easily expressed in the declarative paradigm, lets say that each programming paradigm has its strengths and weaknesses.

1.3 Variables and rules

Something that surprises a lot when you begin using declarative programming is that variables behave completely different. This behavior differs depending on whether the method of reasoning is forwards or backwards.

First of all, all variables are local to each rule, so they are not visible from any other rule. So if we want to pass information among the rules we must use a different mechanisms to pass that information.

In the case of forward reasoning this communication is obtained primarily from using facts. Every rule you want to pass any kind of information to other rules should generate some type of facts to be collected by another rule.

In the case of backward reasoning facts can also be used to pass information between rules, but the usual mechanism is the unification of the variables in the conditions of the rules. Being each subgoal a condition, the rules able to satisfy this subgoal will unify its variables with the corresponding subgoal instance, obtaining this way their values.

Another difference is variable assignation, we can say that does not exist such thing in this paradigm. It is not necessary because all rules are independent from each other and we are using other mechanisms for communicating values among rules.

However there are some rule languages that deviate slightly from the declarative paradigm and allow variable assignation usually locally in each rule [1].

1.4 The facts and the working memory

As communication between different parts of the program is done using facts, the working memory plays a critical role in programming in such languages, especially in languages that use forward reasoning.

The facts can be used to communicate information between rules, and by creating this new facts other rules can instantiated, retrieving this way the information stored on those facts. Obviously by

[1]Some languages allow also assignment to global variables, but obviously this is a heresy independently of the programming paradigm used.

generating this chain of facts we will also get the result of the program.

Another essential use of the facts is to establish control mechanisms to alter how the inference engine explores the space of the problem. Special facts can be created to enable or disable the execution of rules and thus impose some order of execution.

2. Code snippets

2.1 Introduction

A *code snippet* is a piece of code that illustrates how to solve a specific problem. It is common to find collections of examples for many programming languages, primarily imperatives, but there are also examples for declarative languages like Prolog.

Unfortunately there are not many examples about languages that use forward reasoning. This chapter provides programming examples using forward rules and programming techniques and schemes that may be useful when developing applications that use this kind of rules.

Since the CLIPS language also includes a pseudo-functional language, there are things that can be programmed directly using an imperative style using this language and avoiding the declarative paradigm. Obviously, it is better to use it only when absolutely necessary.

The correct use of a rule language requires some effort and a change in the way of thinking about programming. But this effort is rewarded with a change in the perspective about how to program, that even helps to have better imperative programming skills.

STOP Before starting with the examples you should review how an inference engine works and what is the forward chaining reasoning strategy.

2.2 Examples

2.2.1 Computing the factorial

Calculate the factorial is fairly straightforward in an imperative language. Programming it using forward chaining rules requires thinking a little bit more. First we have to note that the rules are independent elements and we need facts in order to execute them. Therefore we need to define a fact that represents the factorial of a number.

Lets call this fact `fact` and lets think of it as a relation that links a number with its factorial. We can say that for example `(fact 3 6)` is representing the fact that the factorial of 3 is 6.

The simplest way to represent facts in CLIPS is using **unordered facts** with them we can establish relationships among elements of any type, the only restriction is that the first element of the fact has to be a **symbol**.

Since we are using forward rules our program will only be able to calculate the factorial of a number if we know the factorial of a previous number. The rule will establish the recurrence between a the factorial of a number and the next one.

```
(defrule factorial
 (fact ?x ?y)
 =>
 (assert (fact (+ ?x 1) (* ?y (+ ?x 1))))
)
```

The rule simply checks for a fact `fact` and as a result creates a new fact that relates the next value to its factorial using the recurrence. This rule by itself does nothing unless we create a fact that allows the recurrence to begin. The obvious choice is to establish the fact that defines the first factorial:

```
(deffacts facts
  (fact 1 1)
)
```

Now the rule can be executed, but if we start the inference engine every factorial that exist will be calculated and the execution will never end. What we are missing from the imperative version is to tell our program what number we want to calculate its factorial.

We can not pass parameters to a rule, so we must introduce facts that control when we have to stop. We can use for example the fact `limit`, that tells what is the number we want to calculate the factorial (for example 6):

```
(deffacts facts
 (fact 1 1)
 (limit 6)
)
```

We must also modify the rule so it stops when this limit is reached:

```
(defrule factorial
 (limit ?l)
 (fact ?x&:(< ?x: ?l): ?y)
 =>
 (assert (fact (+ ?x 1) (* ?y (+? x 1))))
)
```

Basically what we do is to obtain the fact that tells the limit and to check in the factorial rule condition that the value of the number is less than the limit.

A problem that has this rule is that it leaves all the intermediate calculations in the working memory. From an imperative perspective this makes little sense, so we could eliminate them once they have been used:

```
(defrule factorial
 (limit ?l)
 ?f <- (fact ?x&:(< ?x ?l) ?y)
 =>
 (retract ?f)
 (assert (fact (+?x 1) (* ?y (+ ?x 1))))
)
```

We could also not to eliminate the facts and use this circumstance to not to calculate the factorial always from the beginning. This would require some changes in the rule that are left to the reader.

We can see that the rule is in fact reproducing the behavior of an iterative structure in which the facts are the ones that control this behavior while performing the calculation of the factorial. The iteration behaviour is obtained from the exploration performed by the inference engine.

An improvement could be to add rules that allow the rule to ask the number that you want to calculate the factorial and to print the solution at the end. How would you program this?

2.2.2 **Bubble sort**

The Bubble sort algorithm is one of the simplest sorting algorithms of a linear structure (vector, list, ...). We can create a set of rules that sort a structure that simulates for example a vector. CLIPS does not have vectors, so we need to simulate them using facts. To do this instead of `unordered facts` as we used in the previous example we will use a `deftemplate`, just to use a different element from CLIPS, but you can obtain the same result using the former.

The `deftemplate` allows to define the structure of facts, so it lets us to define complex facts and to easily access the features that represent them. The CLIPS syntax allows us to define these characteristics quite freely. In fact CLIPS is a weakly typed language, so we can define features without specifying its type. This is an advantage sometimes, because that allows us to implicitly include genericity, but requires greater discipline during programming.

Lets define the template that we will use as vector positions:

```
(deftemplate pos
 (slot index (type INTEGER))
 (slot value (type INTEGER))
)
```

And with this we can declare a set of facts that represents a vector:

```
(deffacts vector
 (pos (index 1) (value 15))
 (pos (index 2) (value 7))
 (pos (index 3) (value 4))
 (pos (index 4) (value 2))
)
```

Obviously we could have chosen the positions to be not consecutive, but it is more clear this way. Each fact is an individual element, so actually there is no vector in the working memory is just the interpretation we make of the set of facts.

Now we only need the program that performs the sorting. The bubble sort algorithm works by swapping two consecutive positions if they are not in order, and this is precisely what the rule has to do:

```
(defrule sort-bubble
  ?f1 <- (pos (index ?p1) (value ?v1))
  ?f2 <- (pos (index ?p2&:(= ?p2 (+ ?p1 1)))  (value ?v2&:(< ?v2 ?v1)))
=>
 (modify ?f1 (value ?v2))
 (modify ?f2 (value ?v1))
)
```

The condition of the rule states that there must be two consecutive positions out of order. If it is so the values of the positions are swapped.

Here the analogy with the imperative version of the algorithm is not so obvious, apparently there is no loop that iterates through all the different positions. In fact all the work is performed by the inference engine, that will perform all the swaps that the two loops of the imperative version perform. Note that there is no control over in what order will be carried out the swaps and only will be accessed the positions that need to be swapped. In the imperative version, positions can be visited without performing any swapping.

In fact we can think of this type of programming like a parallel execution of an algorithm, where if we were not constrained by the sequential execution of the inference engine we could do several steps at a time by running multiple rules simultaneously.

To complete the program we can include a rule that prints the ordered values of the positions of our virtual vector. It seems difficult to print things in order in this paradigm, but it is all about expressing the conditions properly. The rule that does this is the following:

```
(defrule print-res
 (declare (salience -10))
 ?f1 <- (pos (index ?p1) (value ?v))
 (forall (pos (index ?p2)) (test (<= ?p1 ?p2)))
 =>
 (printout t ?v crlf)
 (retract ?f1)
)
```

The condition simply says that we want a position for which all positions are greater than or equal to it. The implementation of the rule controls the existence of facts that represent the vector. By eliminating the position from the working memory we will have the next element we want to print, until there is none left. Defining the `rule salience` we make sure that this rule will only run once and when the rule that performs the sorting has finished.

The second condition also could have been written as:

```
(not (pos (index ?p2&:(< ?p2 ?p1))))
```

In this case we are saying that there has not to be a fact that has a index lower than the instance from the first condition.

In this program we have used the conditions of the rules to say what we want to obtain. This is a good time for reviewing how conditions as expressed in CLIPS and think about examples of how you can use them to program other sorting algorithms.

CLIPS has the list structure as primitive structure. These can be indexed like vectors, so you can actually program the sorting algorithm imperatively.
A good exercise to learn how to use the functional language of CLIPS could be to implement a function to sort a list of elements.

2.2.3 Max of a set of values

Calculate the maximum may seem complicated using rules. If we think about the imperative version of the algorithm we should have the data in a structure and iterate through it, keeping the maximum value we find.

Actually, we can reproduce this behavior using rules perfectly. For the structure we only need a set of facts that store the values we want to calculate the maximum, using the same relation for each fact will allow us to have the structure we have in the imperative version. If we want to reproduce the behavior of the imperative algorithm we need an additional fact where we are going to store the maximum value during the search. The following set of rules can calculate the maximum of a set of values:

```
(deffacts maximum
  (val 12)
  (val 33)
  (val 42)
  (val 56)
  (val 64)
  (val 72)
)

(defrule init "Initializes the computation of the max"
  (not (max ?))
 =>
 (assert (max 0))
)

(defrule compute-max
  (val ?x)
  ?h <- (max ?y&:(> ?x ?y))
=>
 (retract ?h)
 (assert (max ?x))
)

(defrule print-max
 (declare (salience -10))
 (max ?x)
=>
 (printout t ?x crlf)
)
```

In this case the relation `val` is what links all the values and the fact `max` is the one that stores the maximum value. The first rule creates the fact `max` and initiates its value to the minimum value, which is what we do in the imperative algorithm and the rule `compute-max` compares the values with the current maximum value. The last rule simply prints the value after the maximum value has been calculated. We know this because because the rule `compute-max` can not be executed anymore.

Although apparently the algorithm we are using is the same that the imperative version, the behavior is quite different. First in the imperative algorithm we are the ones that decide the order in which the values are traversed, in this algorithm is the inference engine who decides, so depending on the strategy of conflict resolution, the comparisons that are actually performed may vary considerably.

If for example the first facts to instantiate the rule `compute-max` are `(val 72)` and `(max 0)` no more executions of the rule will occur because that is the maximum value. If you look at what is in the conditions of the rule, what is demanded is that there is a fact `val` and its value has to be larger than the fact `max`. When `max` has the maximum possible value the search will be finished.

That means that if we use wisely the conditions of the rules and how the inference engine explores the space of our problem, we can be more efficient than programming imperatively.

An interesting exercise would be to implement the rules that reproduce the exact behavior of imperative algorithm so that it goes sequentially through all positions to calculate the maximum. The way we implemented the bubble sort algorithm can give you a hint.

If we want to be smarter, we can ignore the imperative version of the algorithm and declare a rule that states what we are looking for, which is simply a fact that has a value that is larger or equal than all other possible values. The condition of equal comes from the fact that, obviously, the maximum value also participates of the comparison, so it is possible that in the comparisons we compare the maximum value with itself. This can be stated with a single rule:

```
(defrule compute-max
 (val ?x)
 (forall (val ?y) (test  (>= ?x ?y)))
=>
 (printout t ?x crlf)
)
```

It may seem miraculous, evidently it is not. Here the work is done by the unification mechanism of the inference engine. It must test all possible comparisons of values and keep the only one that satisfies the conditions. This way of describing how to calculate the maximum is an example of the declarative programming philosophy, we state what we want, but without worrying about how it is computed that is work for the inference engine.

This scheme allows solving algorithms that search for an item from a set that meets certain conditions. In declarative programming is not necessary to make an algorithm to do the search, we only have to query to the fact base properly and the inference engine will give us the item we are looking for. We can think of the fact base as a database and of the inference engine as the query engine of the database.

2.2.4 The traveling salesman problem

As you must know, the traveling salesman problem consists in finding a path that passs through a set of cities, it must begin and end in the same city and should not visit a city more than once.

This problem can be solved by approximation using the hill-climbing algorithm. To perform the search different operators can be used, we are only going to use the exchange of two cities in the path. Thus there will be $\frac{N*(N-1)}{2}$ possible exchanges each iteration.

We are going to use a little more complicated structure to represent the elements of the problem. In this case we will use two **deftemplate** to define the distance matrix and the solution.

```
(deftemplate solution
  (multislot path)
  (slot cost (type INTEGER))
  (slot desc)
)

(deftemplate mat-dist
 (multislot dist)
)
```

In the case of `solution` we have three slots, the solution path, the cost, and a slot that we will use to control the search, which indicates whether the fact corresponds to the current solution or their possible descendants. In `mat-dist` we store the matrix of the distances among cities. This is represented as a list corresponding to the lower (or higher, it does not matter) triangular part of the distance matrix.

It is common in rule programming to include elements in the representation that allow to control how the rules are executed. In this case in the representation of the solution the slot `desc` will be used to distinguish between the steps that make the generation of the successors and those making the selection of the best successor.

We also have two functions to access the distance matrix and to calculate the cost of a path.

```
 ;;; Distance between two cities
(deffunction calc-dist (?i ?j ?nciu ?m)
 (if (< ?i  ?j)
  then (bind ?pm ?i) (bind ?pn ?j)
  else (bind ?pm ?j) (bind ?pn ?i)
 )
 (bind ?off 0)
 (loop-for-count (?i 1 (- ?pm 1))
  do
  (bind ?off (+ ?off (- ?nciu ?i)))
 )
 (nth$ (+ ?off (- ?pn ?pm))  ?m)
)

;;; Cost of a solution
(deffunction calc-cost (?s ?m)
 (bind ?c 0)
 (loop-for-count (?i 1 (- (length$ ?s) 1))
  do
  (bind ?c (+ ?c (calc-dist (nth$ ?i ?s) (nth$ (+ ?i 1) ?s) (length$ ?s) ?m )))
 )
 (bind ?c (+ ?c (calc-dist (nth$ 1 ?s) (nth$ (length$ ?s) ?s) (length$ ?s) ?m)))
)
```

The first one transforms the indices to get the correct position in the list representing the distance matrix, the second one goes through the list representing the path accumulating the distances.

In CLIPS lists are a primitive type and have a set of operators to manipulate them. Usually you will need to use these structures in your programs, so this would be a good time to look up in the manual of CLIPS how they are used.

We will have a fact to indicate the number cities the problem has:

```
(TSP deffacts
 (num-cities 10)
)
```

To obtain the program that implements the hill climbing search, we must first analyze a bit about what is what we have to do. We will have to explore iteratively solutions beginning with an initial solution. For each iteration we have to generate all possible city interchanges and keep the best one. the algorithm will stop when from the current solution no better solution can be generated.

To compute all possible exchanges we can use the inference engine to perform calculations itself. Only an auxiliary fact is needed to control the generation of combinations. This fact will be called `pos` and we will have as many facts as positions and each one will have one of the possible values. To generate all pairs we only need a condition like `(pos ?i) (pos ?j)`. If we have the facts `pos` in the working memory the condition will be instantiated for all possible combinations of pairs. As we do not want to do more work than necessary, we add the constraint `(pos ?i) (pos ?j&:(> ?j ?i))`, so that only pairs that interest us are instantiated and no duplicated are considered.

Many programs usually require generating all combinations of a set of elements. In imperative programming we have to write the program that makes the generation. In declarative programming we can use the unification mechanism of the inference engine and write rules that perform the generation of combinations. A good exercise would be to think of other problems where it is necessary to generate combinations of elements and write the rules that generate them.

To understand better what we have to do we can structure the algorithm in the following steps:

1. Initialize the search by creating an initial solution

2. Generate all the best descendants using city swaps

3. Choose the best successor and update the solution or stop if there is no new solutions

We have to realize that every time the current solution is updated the rule that explores all possible swaps can be executed again because the fact has been changed. This means that we have not to think in terms of iterations, the inference engine does all this work for us.

Lets first implement a simple solution, just having in mind the scheme we have presented, then we will give more thought about the problem and how the inference engine performs the search to do things a little bit more *cleaner*. We will begin with the rule that initializes the search:

```
(defrule init
  (num-cities ?x)
=>
 ; Distance matrix
 (bind ?m (create$))
 (loop-for-count (?i 1 (/ (* ?x (- ?x 1)) 2))
  do
   (bind ?m (insert$ ?m ?i (+ (mod (random) 50) 1)))
 )
 (assert (mat-dist (dist ?m)))
 ; Control fact for the swaps
 (loop-for-count (?i 1 ?x)
  do
  (assert (pos ?i))
 )
 (bind ?s (create$))
 ; Initial solution (Cities in sequential order)
 (loop-for-count (?i 1 ?x)
  do
  (bind ?s (insert$ ?s ?i ?i))
 )
 (assert (solution (path ?s) (cost (calc-cost ?s ?m)) (desc n)))
 (printout t "Initial ->"  ?s ": " (calc-cost ?s ?m) crlf)
)
```

This rule will only run once at the beginning, it is responsible for creating the list representing the distance matrix for the *n* cities, the facts that control the generation of all swaps and the initial solution, that will visit the cities in sequential order.

The next rule is the one that performs the generation each iteration (so to speak) of all possible swaps.

```
(defrule HC-step1
 (pos ?i)
 (pos ?j&:(> ?j ?i))
 (solution (path $?s) (cost ?c) (desc n))
 (mat-dist (dist $?m))
 =>
 (bind ?vi (nth$ ?i ?s))
 (bind ?vj (nth$ ?j ?s))
 (bind ?sm (delete$ ?s ?i ?i))
 (bind ?sm (insert$ ?sm ?i  ?vj))
 (bind ?sm (delete$ ?sm ?j ?j))
 (bind ?sm (insert$ ?sm ?j  ?vi))
 (bind ?nc (calc-cost ?sm ?m))
 (assert (solution (path ?sm) (cost ?nc) (desc s)))
)
```

This rule is instantiated with all possible swaps and creates a solution descendant of the current. This will fill the memory of facts of this type for each time it can be instantiated. We do not need to be concerned about if the solution is better than the current one, we will use another rule to select the one with the best cost of all generated solutions. This will be done by the following rule:

```
(defrule HC-step2
 (declare (salience -10))
 (solution (path $?s) (cost ?c) (desc s))
 (forall
   (solution  (cost ?oc) (desc s))
   (test (<= ?c ?oc)))
 ?solact <- (solution (cost ?cs&:(< ?c ?cs)) (desc n))
 =>
  (modify ?solact (path ?s) (cost ?c))
  (printout t "->"  ?s ": " ?c crlf)
)
```

We simply say we want a descendant solution that has the minimum cost and that has a cost better than the current solution. We want to execute this rule once we have executed all possible instantiations of the previous rule, so we give to it a lower priority.

We will know we have finished the search when none of the previous rules can be executed. We can also include a rule that prints the solution when this happens:

```
(defrule HC-is-end
 (declare (salience -20))
 (solution (path $?s) (cost ?c) (desc n))
 =>
  (printout t "Final ->"  ?s ": " ?c crlf)
)
```

By assigning to this rule the lower priority we will know that will be executed at the end.

In this case, we have assigned priorities this way because we want the search to behave like the original Hill Climbing algorithm, but we could play a little bit with the rules or with the conflict resolution strategy of the inference engine to obtain other behaviors. If for example we assign the two first rules the same priority, each time the second rule is executed the current solution will be updated, changing the point from which the search continues. If we use a conflict resolution strategy depth-first we would have a hill climbing that every step would move to the first swap that improves the current solution, rather than choose the best of all, that is what we get with the priorities we have assigned.

Cleaning up the memory a little

The rules as they are now are doing exactly what we want, but we can see that it is a problem. We fill the working memory with all the intermediate solutions that we obtain and clearly filling the working memory increases the work for the inference engine.

In fact, the unification and the detection of the rules to be applied is quite efficient in forward reasoning engine (RETE algorithm), but for obtaining this efficiency some data structures that allow to update the unifications must be built and this increases the memory needed.

A simple first change in the current solution is not to generate solutions that are worse than the current one, we can modify the rule `HC-step1` simply by changing the final assert:

```
(if (< ?nc ?c)
  then (assert (solution (path ?sm) (cost ?nc) (desc s))))
```

Another strategy for not filling the working memory is to clean all successors once we have selected the best solution. To do this we must divide our algorithm in two differentiated phases, the rules that search for the solution and the rules that clean the working memory. To do this we use two control facts `(search)` and `(clean)` that are used to activate the rules that perform each phase. When one set of rules finish their work the control will be passed to the other by simply asserting and retracting the appropriate control facts.

This way, the rule that initializes the search will assert the control fact that activates the search rules:

```
(defrule init
  (num-cities ?x)
=>
 (bind ?m (create$))
 (loop-for-count (?i 1 (/ (* ?x (- ?x 1)) 2))
  do
   (bind ?m (insert$ ?m ?i (+ (mod (random) 50) 1)))
 )
 (assert (mat-dist (dist ?m)))
 (loop-for-count (?i 1 ?x)
  do
  (assert (pos ?i))
 )
 (bind ?s (create$))
 (loop-for-count (?i 1 ?x)
  do
  (bind ?s (insert$ ?s ?i ?i))
 )
(assert (solution (path ?s) (cost (calc-cost ?s ?m)) (desc n)))
(assert (search))
(printout t "Initial ->"  ?s ": " (calc-cost ?s ?m) crlf)
)
```

We will have three rules that perform the search that will be controlled by the fact `search`:

```
(defrule HC-step1
 (search)
 (pos ?i)
 (pos ?j&:(> ?j ?i))
 (solution (path $?s) (cost ?c) (desc n))
 (mat-dist (dist $?m))
 =>
 (bind ?vi (nth$ ?i ?s))
 (bind ?vj (nth$ ?j ?s))
 (bind ?sm (delete$ ?s ?i ?i))
 (bind ?sm (insert$ ?sm ?i  ?vj))
 (bind ?sm (delete$ ?sm ?j ?j))
 (bind ?sm (insert$ ?sm ?j  ?vi))
 (bind ?nc (calc-cost ?sm ?m))
 (if (< ?nc ?c)
   then (assert (solution (path ?sm) (cost ?nc) (desc s))))
)

(defrule HC-step2
 (declare (salience -10))
 ?h <-(search)
 ?solact <- (solution (desc n))
 (solution (path $?s) (cost ?c) (desc s))
 (forall
   (solution  (cost ?oc) (desc s))
   (test (<= ?c ?oc)))
 =>
  (modify ?solact (path ?s) (cost ?c))
  (printout t "->"  ?s ": " ?c crlf)
 (assert (clean))
 (retract ?h)
)

(defrule HC-is-end
 (declare (salience -20))
 (search)
 (solution (path $?s) (cost ?c) (desc n))
 =>
  (printout t "Final ->"  ?s ": " ?c crlf)
)
```

Note that the rule `HC-step2` is the only one able to pass control to the rules that clean the working memory. It should also be noted that the condition of this rule does not require that the cost of the best descendant to be better than the current solution.

The reason for that change is because the condition in the first set of rules is a bit more complicated than it apparently seems. If we do not require the condition this way, the rule can enter in an infinite loop due to the way it works. This rule modifies the fact that corresponds to the current solution with the best solution (the logic thing to do), and this provokes that the rule can be activated again. Because in this set of rules we are going to remove all the descendants solutions now there is no possibility of infinite loop.

For cleaning the intermediate facts we can use the following set of rules:

```
(defrule clean-HC
 (clean)
 ?h <- (solution (desc s))
=>
 (retract ?h)
)

defrule HC-reinit
 ?h <- (clean)
 (not (solution (desc s)))
=>
 (retract ?h)
 (assert (search))
)
```

The first rule simply instantiates the facts and deletes them. When there are no more intermediate facts the other rule passes control to the rules that perform the search.

This example illustrates how you can divide the different parts of an algorithm implemented by rules and pass control from one part to another. In the case that each part does relatively simple things this mechanism is quite handy.

If we are faced with something more complex, it is best to use the CLIPS module mechanism. On one side, it will be more efficient and control will be easier, on the other side, we will not need to remove from the working memory the facts generated that are private to the module, since they are automatically deleted once a module finishes its execution.

Now with objects

Just to show how you can use the CLIPS objects system, we are going to rewrite the problem, transforming the facts defined as `deftemplate` as object. The advantage is that there are primitive operations for objects that allow to query for the objects using the same kind of patterns we use in the rules, which will facilitate the cleaning of temporary facts. Also, the information can be structured better and make the rules a bit clearer, leaving all the work to the objects.

This is a good time to review how CLIPS objects can be defined, the mechanisms for using them inside the rules and the definition and invocation of methods.

We will begin by defining classes that represent the information of the problem:

```
(defclass mat-dist
  (is-a USER)
  (role concrete)
  (pattern-match reactive)
  (slot nciu)
  (multislot dist)
)

(defclass solution
  (is-a USER)
  (role concrete)
  (pattern-match reactive)
  (multislot way)
  (slot cost (type INTEGER))
  (slot desc)
)
```

There is little difference about how we have defined this in the previous versions. In this case we included the number of cities inside the distance matrix.

To deal with the different operations that we have to perform we are now going to define them as `message-handlers` saving this way some parameter passing and encapsulating the modification of the solution.

```
 (defmessage-handler mat-dist calc-dist (?i ?j)
 (if (< ?i  ?j)
  then (bind ?pm ?i) (bind ?pn ?j)
  else (bind ?pm ?j) (bind ?pn ?i)
 )
 (bind ?off 0)
 (loop-for-count (?i 1 (- ?pm 1))
  do
  (bind ?off (+ ?off (- ?self:nciu ?i)))
 )
 (nth$ (+ ?off (- ?pn ?pm))  ?self:dist)
)

(defmessage-handler solution calc-cost (?m)
 (bind ?self:cost 0)
 (loop-for-count (?i 1 (- (length$ ?self:path) 1))
  do
  (bind ?self:cost
   (+ ?self:cost (send ?m calc-dist
                       (nth$ ?i ?self:path)
                       (nth$ (+ ?i 1) ?self:path))))
 )
 (bind ?self:cost
    (+ ?self:cost (send ?m calc-dist
                       (nth$ 1 ?self:path)
                       (nth$ (length$ ?self:path) ?self:path))))
)

(defmessage-handler solution print-sol ()
 (printout t ?self:path ": " ?self:cost crlf )
)

(defmessage-handler solution swap (?i ?j ?m)
 (bind ?vi (nth$ ?i ?self:path))
 (bind ?vj (nth$ ?j ?self:path))
 (bind ?sm (delete$ ?self:path ?i ?i))
 (bind ?sm (insert$ ?sm ?i ?vj))
 (bind ?sm (delete$ ?sm ?j ?j))
 (bind ?sm (insert$ ?sm ?j ?vi))
 (bind ?self:path ?sm)
 (send ?self calc-cost ?m)
)
```

The first `message-handler` returns the distance between two cities, the second calculates the cost of a path, the third prints the information of a solution and the fourth performs the swap between two cities and updates the cost of the path.

The object-oriented language of CLIPS not only can be used to define the domain concepts of the program. Using techniques of object-oriented programming can help to develop CLIPS programs by allowing data encapsulation. Think also that being CLIPS a weakly typed language methods can be overloaded and genericity implementation techniques can be applied. The inheritance mechanism can also help to simplify programming methods.

Now the rule that initializes the search will be somewhat different given the change of representation:

```
(defrule init
  (num-cities ?x)
=>
 (bind ?m (create$))
 (loop-for-count (?i 1 (/ (* ?x (- ?x 1)) 2))
  do
   (bind ?m (insert$ ?m ?i (+ (mod (random) 50) 1)))
 )
 (bind ?md
   (make-instance (gensym) of mat-dist (dist ?m) (nciu ?x)))
 (loop-for-count (?i 1 ?x)
  do
  (assert (pos ?i))
 )
 (bind ?s (create$))
 (loop-for-count (?i 1 ?x)
  do
  (bind ?s (insert$ ?s ?i ?i))
 )
 (bind ?sol
  (make-instance (gensym) of solution
                           (path ?s) (desc n)))
 (send ?sol calc-cost ?md)
 (printout t "Initial -> ")
 (send ?sol print-sol)
)
```

Now instances are created using `make-instance`, the function `(gensym)` generates a new symbol each time it is called and can be used to generate the names of the instances.

The rules that perform the search also change slightly to accommodate the representation with objects. We changed the conditions using the constructor `object` and we modify and create the solutions using the methods that the objects have:

```
(defrule HC-step1
 (pos ?i)
 (pos ?j&:(> ?j ?i))
 ?s <- (object (is-a solution) (desc n))
 ?m <- (object (is-a mat-dist))
 =>
 (bind ?ns (duplicate-instance ?s to (gensym)))
 (send ?ns put-desc s)
 (send ?ns swap ?i ?j ?m)
 (if (< (send ?s get-cost) (send ?ns get-cost))
   then (send ?ns delete))
)
```

The descendant solutions are created by duplicating the object that has the current solution and modifying the path. In this case each time an object is createt is stored in the working memory, so we can delete it if it has higher cost. With that we have the same effect as in the previous solution.

The rule that keeps the best solution is not very different:

```
(defrule HC-step2
 (declare (salience -10))
 ?solmejor <- (object (is-a solution)  (cost ?c) (desc s))
 (forall
   (object (is-a solution) (cost ?oc) (desc s))
   (test (<= ?c ?oc)))
 ?solact <- (object (is-a solution) (cost ?cs&:(< ?c ?cs)) (desc n))
 =>
  (send ?solmejor put-desc n)
  (send ?solact delete)
  (do-for-all-instances ((?sol solution)) (eq ?sol:desc s)
   (send ?sol delete))
  (send ?solmejor print-sol)
)
```

In this case we change the value of slot `desc` to `n` and we delete the current solution. We also use one of the functions that allows to query the object database to eliminate all descendants solutions. We could have used same method as in the previous solution, but this option is more efficient.

Finally this is the rule that prints just the solution when the search ends:

```
(defrule HC-is-end
 (declare (salience -20))
 ?sol <- (object (is-a solution) (desc n))
 =>
  (printout t "Final -> ")
  (send ?sol print-sol)
```

You can still make more modifications to the program. For example, you could define two specializations of the class solution, a solution for the current one and one for the descendants, eliminating the need for slot `desc`. The distances matrix could also be included in the solution, avoiding this way to include it in the conditions of the rules.

2.3 The auto repair expert system

One of the examples that includes the distribution of CLIPS is an expert system for diagnosing problems in a car. The system is very simple and only allows for a limited number of responses, but is illustrative of how a flow of questions and hypothesis generation in a rule-based program can be controlled. You can access the program code at http://www.lsi.upc.edu/~bejar/ia/material/laboratorio/clips/a

The control is based on a set of facts that are asserted in the working memory and that allow rules to be executed in a specific order to reach the different diagnoses that can be obtained. Since the problem is quite simple using *unordered* facts is enough but for a more complex problem it would require a more structured representation.

The facts that are used are:

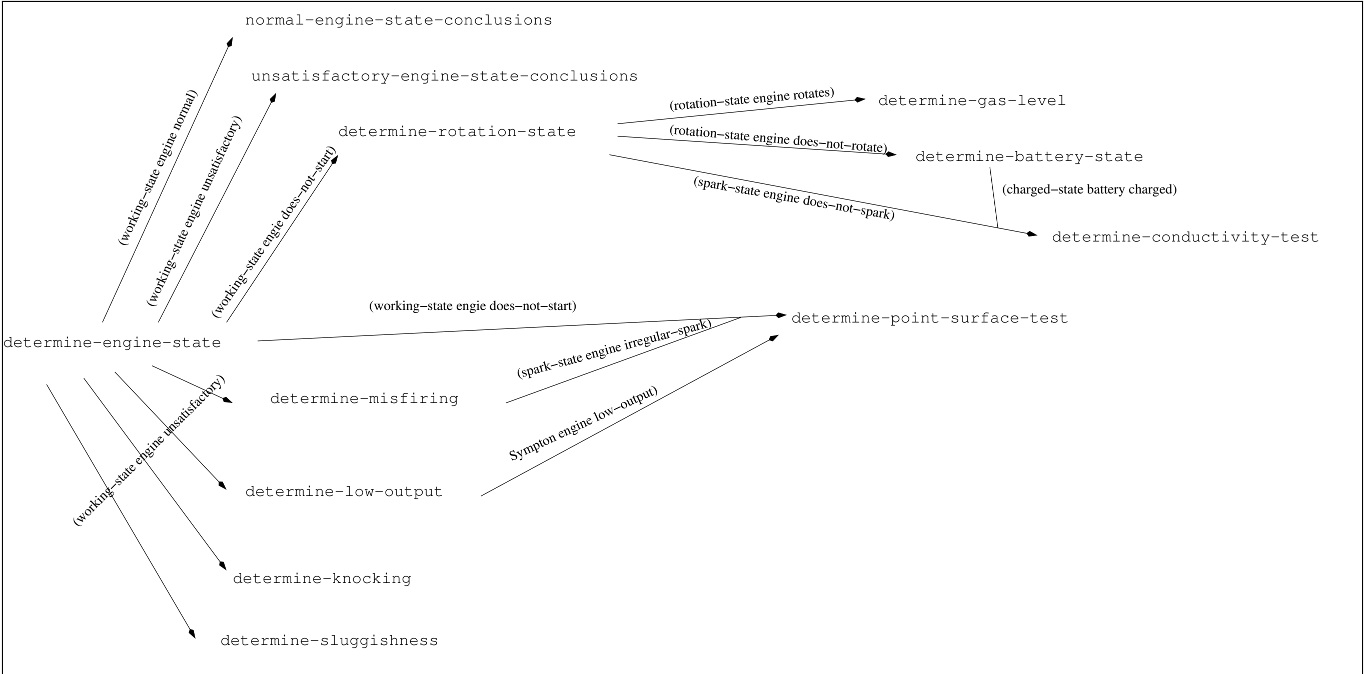
normal-engine-state-conclusions
unsatisfactory-engine-state-conclusions
determine-rotation-state
determine-gas-level
determine-battery-state
determine-conductivity-test
determine-point-surface-test
determine-engine-state
determine-misfiring
determine-low-output
determine-knocking
determine-sluggishness
(working–state engine normal)
(working–state engine unsatisfactory)
(working–state engie does–not–start)
(rotation–state engine rotates)
(rotation–state engine does–not–rotate)
(spark–state engine does–not–spark)
(charged–state battery charged)
(working–state engie does–not–start)
(spark–state engine irregular–spark)
Sympton engine low–output)
(working–state engine unsatisfactory)

Figure 2.1: Chaining of the rules

- (working-state engine normal | unsatisfactory | does-not-start)
- (rotation-state engine rotates | does-not-rotate)
- (spark-state engine irregular-spark | does-not-spark)
- (symptom engine low-output | not-low-output)
- (charge-state battery charged)

Following the structure that has been determined for the diagnosis, the program will be asking questions and facts will be asserted from the answers that lead to the execution of a sequence of rules that result in a specific diagnosis. In figure 2.1 you can see the different links among the program rules.

Before going into detail with the rules we should comment on the functions that are used in the rules to ask to the user for information.

```
(deffunction ask-question (?question $?allowed-values)
   (printout t ?question)
   (bind ?answer (read))
   (if (lexemep ?answer)
       then (bind ?answer (lowcase ?answer)))
   (while (not (member ?answer ?allowed-values)) do
      (printout t ?question)
      (bind ?answer (read))
      (if (lexemep ?answer)
          then (bind ?answer (lowcase ?answer))))
   ?answer)

(deffunction yes-or-no-p (?question)
   (bind ?response (ask-question ?question yes no y n))
```

```
  (if (or (eq ?response yes) (eq ?response y))
      then TRUE
      else FALSE))
```

The first function (`ask-question`) receives the text of a question as the first parameter and an unbounded number of parameters that correspond to the acceptable answers. The second parameter is a variable that accepts multiple values and will store as a list the extra parameters that are received.

To print the question it is used the operator `printout` that receives a channel (`t` represents the standard output) and a string. To read the input, the operator `read` is used. In CLIPS the input that is read is interpreted in order to cast the value to the corresponding data type, because of this, it is checked if the input is a string or a symbol (`lexemp`). The function will keep reading until it receives a response that is in the list that has been passed as parameter and then it will return the value read.

The function `yes-or-no-p` only admits as answers `(yes no y n)` and returns true or false depending on whether the answer is affirmative or negative.

The program is divided into three groups of rules. The rules that only deduce facts and do not ask questions to the the user, the rules that ask questions to the user and those that present the system and write the final result.

These latter rules are controlled using the `salience` and the existence in the working memory of the fact that indicates that it has been obtained a repair.

```
;;;****************************
;;; * STARTUP AND REPAIR RULES *
;;;****************************

(defrule system-banner ""
  (declare (salience 10))
  =>
  (printout t crlf crlf)
  (printout t "The Engine Diagnosis Expert System")
  (printout t crlf crlf))

(defrule print-repair ""
  (declare (salience 10))
  (repair ?item)
  =>
  (printout t crlf crlf)
  (printout t "Suggested Repair:")
  (printout t crlf crlf)
  (format t " %s%n%n%n" ?item))

```

For the rest of the rules the presence or absence of facts in the working memory is used to activate them. For example, all the rules that diagnose have the condition `(not (repair?))` in order to disable them when one of the rules that gives a diagnose asserts the final solution.

It is the algorithm of the diagnosis process the one that will tell us when to activate the rules and what conditions to put in each rule. For example, the first rule of the diagnostic process is:

```
 (defrule determine-engine-state ""
   (not (working-state engine ?))
   (not (repair ?))
   =>
   (if (yes-or-no-p "Does the engine start (yes/no)? ")
       then
       (if (yes-or-no-p "Does the engine run normally (yes/no)? ")
           then (assert (working-state engine normal))
           else (assert (working-state engine unsatisfactory)))
       else
       (assert (working-state engine does-not-start))))
```

All other rules depend directly or indirectly from the fact `(working-state engine ?)`, this rule will be the first one to run and ask about the status of the engine.

Salience is also used as a control mechanism, to be sure that certain rules are executed first, for instance:

```
(defrule unsatisfactory-engine-state-conclusions ""
   (declare (salience 10))
   (working-state engine unsatisfactory)
   =>
   (assert (charge-state battery charged))
   (assert (rotation-state engine rotates)))
```

will be executed before:

```
(defrule determine-sluggishness ""
   (working-state engine unsatisfactory)
   (not (repair ?))
   =>
   (if (yes-or-no-p "Is the engine sluggish (yes/no)? ")
       then (assert (repair "Clean the fuel line."))))
```

or also that certain rules are executed when no other can:

```
(defrule no-repairs ""
  (declare (salience -10))
  (not (repair ?))
  =>
  (assert (repair "Take your car to a mechanic.")))
```

As control mechanisms we can play with the presence or absence of facts, to make that the execution of the rules fill the information that we need for the problem or to make that the rules that deduce facts that can be derived from the ones we already have are executed, for example:

```
 (defrule determine-battery-state ""
   (rotation-state engine does-not-rotate)
   (not (charge-state battery ?))
   (not (repair ?))
   =>
   (if (yes-or-no-p "Is the battery charged (yes/no)? ")
       then
```

```
        (assert (charge-state battery charged))
        else
        (assert (repair "Charge the battery."))
        (assert (charge-state battery dead))))

(defrule determine-conductivity-test ""
   (working-state engine does-not-start)
   (spark-state engine does-not-spark)
   (charge-state battery charged)
   (not (repair ?))
   =>
   (if (yes-or-no-p "Is the conductivity test for the ignition coil
                                                        positive (yes/no)? ")
        then
        (assert (repair "Repair the distributor lead wire."))
        else
        (assert (repair "Replace the ignition coil."))))
```

You can execute the program to find out what sets of answers lead to each one of the diagnoses. Remember that with the rules we are defining the graph that connects the different facts of the problem and the solutions. This graph can be very simple or very complex, we have only to determine which paths we want to represent with the rules.

For example, suppose we make the following interaction with the program:

```
The Engine Diagnosis Expert System

Does the engine start (yes/no)?  no
Does the engine rotate (yes/no)? yes
What is the surface state of the points (normal/burned/contaminated)? normal
Does the tank have any gas in it (yes/no)? no


Suggested Repair:

 Add gas.
```

If we follow the program, the first rule to be executed is `system-banner` since it has the greatest salience and has no conditions. After this the rule `determine-engine-state` will be executed, that is the one that initiates the process to obtain the diagnosis. If you look at the agenda you will see that we will always can execute the rule `no-repairs` that is waiting for the moment when no other rule of higher priority can run.

When we answer `no` to its question the fact `(working-state engine does-not-start)` is added to the working memory. This fact allows the rule `determine-rotation-state` to be executed, asking the question about the status of the motor rotation. This rule adds two facts if we answer its question affirmatively, `(rotation-state engine rotate)` and `(spark-state engine irregular-spark)`. This leads to the circumstance that there are two rules with the same priority that can be executed at this time `determine-point-surface-state` and `determine-gas-level`.

In this case the order of execution depends on the conflict resolution strategy that is using the inference engine. If you look at the agenda during program execution and change the strategy, you will see that the order of the rules in this changed in the agenda.

Assuming that the conflict resolution strategy selects the rule `determine-point-surface-state` and we answer that the state of the surface of the plugs are normal, we will have new facts on the fact base, and the rule `determine-gas-level` will be executed. Responding negatively to the question of this rule adds to the fact base the fact `(repair "Add gas.")` that activates the rule that writes the solution, ending the program.

Obviously, with a different set of answers we would follow other rule chaining and we would get another result.

These lecture notes introduce to rule programming using the CLIPS expert systems environment by solving simple examples.

A problem with programming using rules is the necessity of changing from the imperative way of programming to the declarative one. From well known examples such as the computing of the factorial, finding the maximum of an array, sorting an array or solving the travel salesman problem, different schemas of programming using rules and the declarative programming approach are presented.

With the examples the basic elements of the CLIPS language are also introduced such as declaring facts, declaring the conditions of a rule, programming using the functional language from CLIPS, declaring and using objects and the programming of message handlers.

Javier Béjar teaches the course Artificial Intelligence at the Barcelona Computing School of the Technical University of Catalonia (bejar@lsi.upc.edu)

FIB *Facultat d'Informàtica de Barcelona*
Departament de Llenguatges i Sistemes informàtics
Universitat Politècnica de Catalunya