CSCI 4202 – Introduction to Artificial Intelligence
Spring 2017 – Dr. Williams
Programming Assignment 3 – CLIPS Bagger Production System

In this programming assignment you will implement the Bagger Production System from class in CLIPS. Since most of you have not had any exposure to rule-based programming, this document will guide you through getting started.

CLIPS

Download CLIPS for your platform from http://clipsrules.sourceforge.net/.

Read the Lecture-14-CLIPS.pdf file in the Lecture 14 – CLIPS folder on the course's Canvas web site. This gives you an introduction to CLIPS.

Bagger Production System

Read the Bagger-Expert-System.pdf file in the Lecture 16 – Forward-Chaining Expert Systems.

CLIPS Implementation

> The original text from the Bagger Production System pdf will appear as in that document and comments on implementing it in CLIPS will be indented like this. I have tried to keep to the structure of the structured English version, but there are places where I deviated.

Suppose that Robbie has been hired to bag groceries in a grocery store. Because he knows little about bagging groceries, he approaches his new job by creating Bagger, a rule-based production system that decides where each item should go.

After a little study, Robbie decides that Bagger should be designed to take four steps:

1. The check-order step: Bagger analyzes what the customer has selected, looking over the groceries to see whether any items are missing.
2. The bag-large-items step: Bagger bags the large items, taking care to put the big bottles in first.
3. The bag-medium-items step: Bagger bags the medium items, taking care to put frozen ones in freezer bags.
4. The bag-small-items step: Bagger bags the small items.

> In CLIPS, we can use a simple ordered fact, like `(step check-order)`, to store the current step. There are four (4) step changing rules in Bagger: B2, B6, B10, and B13. The last rule, B13, changes the step to done, which does not match any rules. The system will effective halt at that point.

We can represent these rules in CLIPS as follows:

```
(deffacts initial-facts
  (step check-order))

(defrule B2
  ?step-fact <- (step check-order)
=>
  (retract ?step-fact)
  (assert (step bag-large-items)))

(defrule B6
  ?step-fact <- (step bag-large-items)
=>
  (retract ?step-fact)
  (assert (step bag-medium-items)))

(defrule B10
  ?step-fact <- (step bag-medium-items)
=>
  (retract ?step-fact)
  (assert (step bag-small-items)))

(defrule B13
  ?step-fact <- (step bag-small-items)
=>
  (retract ?step-fact)
  (assert (step done)))
```

The deffacts If you type these rules into a file, like bagger-step.clp, you can load it into the CLIPS interpreter and run it. Here is a listing of such a run.

```
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (watch rules)
CLIPS> (load "C:/Users/Doug/Desktop/bagger-step.clp")
Defining deffacts: initial-facts
Defining defrule: B2 +j+j
Defining defrule: B6 +j+j
Defining defrule: B10 +j+j
Defining defrule: B13 +j+j
TRUE
CLIPS> (reset)
<== f-0     (initial-fact)
==> f-0     (initial-fact)
==> f-1     (step check-order)
==> Activation 0      B2: f-1
CLIPS> (run)
FIRE    1 B2: f-1
<== f-1     (step check-order)
==> f-2     (step bag-large-items)
==> Activation 0      B6: f-2
FIRE    2 B6: f-2
<== f-2     (step bag-large-items)
==> f-3     (step bag-medium-items)
==> Activation 0      B10: f-3
FIRE    3 B10: f-3
<== f-3     (step bag-medium-items)
==> f-4     (step bag-small-items)
==> Activation 0      B13: f-4
FIRE    4 B13: f-4
```

```
<== f-4     (step bag-small-items)
==> f-5     (step done)
CLIPS>
```

I used the watch command to trace the facts (assertions and retractions) and rules (activations and firings). You could also use the facts and agenda windows and step command (control-T) to see the same thing in the GUI.

Now let us see how this knowledge can be captured in a rule-based production system. First, Bagger needs working memory. The working memory must contain assertions that capture information about the items to be bagged. Suppose that those items are the items in the following table:

Item	**Container type**	**Size**	**Frozen?**
Bread	plastic bag	medium	no
Glop	jar	small	no
Granola	cardboard box	large	no
Ice cream	cardboard carton	medium	yes
Potato chips	plastic bag	medium	no
Pepsi	bottle	large	no

Next, Bagger needs to know which step is the current step, which bag is the current bag, and which items already have been places in bags. In the following example, the first assertion identifies the current step as the check-order step, the second identifies the bag as Bag1, and the remainder indicate what items are yet to be bagged.

Initial Working Memory:

Step is check-order
Bag1 is a bag
Bread is to be bagged
Glop is to be bagged
Granola is to be bagged
Ice cream is to be bagged
Potato chips are to be bagged
Pepsi is to be bagged

Now we need to represent the other elements that will be stored in working memory. Here is how I approached it:

```
;;; An item represents something to be bagged.
(deftemplate item
  (slot name
        (type SYMBOL))
  (slot container-type
        (type SYMBOL))
  (slot size
        (allowed-values small medium large))
```

```
  (slot frozen
        (allowed-values no yes)
               (default no)))

;;; New bag names are generated by keeping a counter, which is initializes to
;;; zero, and when we need a new bag we increment the counter and create a
;;; symbol by appending to the "bag". This is a purely procedural approach.

(defglobal ?*prev-bag-number* = 0)

(deffunction new-bag-name ()
  (bind ?*prev-bag-number* (+ ?*prev-bag-number* 1))
  (sym-cat bag ?*prev-bag-number*))

;;; The current bag has a name as well as the number and size of its contents.
;;; For bagger, bags only have items of a particular size. Note that current-bag
;;; self initializes - name defaults to the next bag name, item-count defaults
;;; to zero, and item-size initializes to nil, which won't match any known size.
(deftemplate current-bag
  (slot name
        (type SYMBOL)
               (default-dynamic (new-bag-name)))
  (slot item-count
        (type INTEGER)
               (default 0))
  (slot item-size
        (type SYMBOL)))

;;; Bagged facts represent items that have been bagged.
(deftemplate bagged
  (slot item-name
        (type SYMBOL))
  (slot bag-name
        (type SYMBOL)))

;;; The initial facts for Bagger. Note that the pepsi is commented out. This
;;; allows you to test rule B1.
(deffacts initial
  (current-bag)
  (item (name bread)        (container-type plastic-bag)      (size medium))
  (item (name glop)         (container-type jar)              (size small))
  (item (name granola)      (container-type cardboard-box)    (size large))
  (item (name ice-cream)    (container-type cardboard-carton) (size medium) (frozen yes))
  (item (name potato-chips) (container-type plastic-bag)      (size medium))
  ;(item (name pepsi)        (container-type bottle)           (size large))
)
```

The item deftemplate and initial facts are a straightforward representation of the table above. The current-bag and new-bag-name function represent the bag that is currently being filled. In keeping with the Bagger system, and keeping it simple, a bag can only hold one size (small, medium, or large) of items and we also maintain a count of the items currently in the bag. Finally, the bagged fact relates items to bags they are in.

There is also the matter of the freezer bag, which I just used a simple ordered fact in-freezer-bag with the name of the item.

You may, of course, choose to use different facts for your program.

CLIPS doesn't support rule ordering. For this production system, the complexity strategy is best. The specificity of a rule is determined by the number of comparisons that must be performed on the preconditions of the rule. With the complexity strategy, among rules

with the same salience, newly activated rules are placed above all activations of rules with equal or lower specificity. That is, the most complex rule instances are selected first.

I added a rule B0 to set the control strategy to complexity. It also seems a better place to initialize the step to check-order.

```
(defrule B0
=>
  (printout t "Bagger Production System" crlf)
  (set-strategy complexity)
  (assert (step check-order)))
```

What remains is encoding the rules themselves. Here are examples of encoding some of the rules.

```
(defrule B1
  (step check-order)
  (item (name potato-chips))
  (not (item (name pepsi)))
=>
  (printout t "Do you want a bottle of Pepsi? ")
  (if (= (str-compare (upcase (readline t)) "YES") 0)
   then (assert (item (name pepsi) (container-type bottle) (size large)))))
```

Rule B1 is kind of a pain because CLIPS I/O and string manipulation is rather primitive. So, I've provided that rule.

```
(defrule B3
  (step bag-large-items)
  ?item-fact <- (item (name ?item-name) (container-type bottle) (size large))
  ?current-bag-fact <- (current-bag (name ?bag-name) (item-count ?item-count&:
(< ?item-count 6)))
=>
  (retract ?item-fact)
  (assert (bagged (item-name ?item-name) (bag-name ?bag-name)))
  (modify ?current-bag-fact (item-count (+ ?item-count 1)) (item-size large)))
```

Rule B3 is an example of a rule that actually bags something – a large item in a bottle in this case. Note that I didn't check if the bag only has large items, because we cannot have bagged anything smaller yet. [Neither does the structured English rule B3.]

```
(defrule B8
  (step bag-medium-items)
  ?item-fact <- (item (name ?item-name) (size medium))
  ?current-bag-fact <- (current-bag (name ?bag-name) (item-count ?item-count)
(item-size ?item-size))
  (test (or (= ?item-count 0) (and (eq ?item-size medium) (< ?item-count 12))))
=>
  (retract ?item-fact)
  (assert (bagged (item-name ?item-name) (bag-name ?bag-name)))
  (modify ?current-bag-fact (item-count (+ ?item-count 1)) (item-size medium)))
```

Rule B8 is similar to rule B3, but for medium items. Here we have to check both size of the items in the bag as well as the count. This is most easily done with a separate test precondition.

```
(defrule B5
  (step bag-large-items)
  (item (name ?item-name) (size large))
  ?current-bag-fact <- (current-bag)
=>
  (retract ?current-bag-fact)
  (assert (current-bag)))
```

Rule B5 is a rule that changes the current bag. Note that we don't say what size items will be in the bag until we actually put something in it.

All of the other rules are similar in structure to these ones.

Finally, I added some logging rules that print out when something is put into a bag or a freezer bag. This keeps the original rules clean and we don't have to duplicate the code to print these out.

```
(defrule log-bagged
  (declare (salience 10))
  (bagged (item-name ?item-name) (bag-name ?bag-name))
=>
  (printout t ?item-name " in " ?bag-name crlf))

(defrule log-in-freezer-bag
  (declare (salience 10))
  (in-freezer-bag ?item-name)
=>
  (printout t ?item-name " in freezer bag" crlf))
```

Here is a trace my implementation of Bagger in CLIPS.

```
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (watch rules)
CLIPS> (load "C:/Users/Doug/Desktop/bagger.CLP")
Defining deftemplate: item
Defining defglobal: prev-bag-number
Defining deffunction: new-bag-name
Defining deftemplate: current-bag
Defining deftemplate: bagged
Defining deffacts: initial
Defining defrule: B0 +j+j
==> Activation 0      B0: *
Defining defrule: B1 +j+j+j+j
Defining defrule: B2 =j+j
Defining defrule: B3 +j+j+j+j
Defining defrule: B4 =j+j+j+j
Defining defrule: B5 =j=j+j+j
Defining defrule: B6 =j+j
Defining defrule: B7 +j+j+j+j
Defining defrule: B8 =j+j+j+j
Defining defrule: B9 =j=j+j+j
Defining defrule: B10 =j+j
Defining defrule: B11 +j+j+j+j
Defining defrule: B12 =j=j+j+j
Defining defrule: B13 =j+j
Defining defrule: log-bagged +j+j
Defining defrule: log-in-freezer-bag +j+j
TRUE
```

```
CLIPS> (reset)
<== Activation 0      B0: *
<== f-0     (initial-fact)
==> Activation 0      B0: *
==> f-0     (initial-fact)
==> f-1     (current-bag (name bag1) (item-count 0) (item-size nil))
==> f-2     (item (name bread) (container-type plastic-bag) (size medium)
(frozen no))
==> f-3     (item (name glop) (container-type jar) (size small) (frozen no))
==> f-4     (item (name granola) (container-type cardboard-box) (size large)
(frozen no))
==> f-5     (item (name ice-cream) (container-type cardboard-carton) (size
medium) (frozen yes))
==> f-6     (item (name potato-chips) (container-type plastic-bag) (size
medium) (frozen no))
CLIPS> (run)
FIRE    1 B0: *
Bagger Production System
==> f-7     (step check-order)
==> Activation 0      B2: f-7
==> Activation 0      B1: f-7,f-6,*
FIRE    2 B1: f-7,f-6,*
Do you want a bottle of Pepsi? yes
==> f-8     (item (name pepsi) (container-type bottle) (size large) (frozen
no))
FIRE    3 B2: f-7
<== f-7     (step check-order)
==> f-9     (step bag-large-items)
==> Activation 0      B6: f-9
==> Activation 0      B5: f-9,f-4,f-1
==> Activation 0      B4: f-9,f-4,f-1
==> Activation 0      B5: f-9,f-8,f-1
==> Activation 0      B4: f-9,f-8,f-1
==> Activation 0      B3: f-9,f-8,f-1
FIRE    4 B3: f-9,f-8,f-1
<== f-8     (item (name pepsi) (container-type bottle) (size large) (frozen
no))
<== Activation 0      B4: f-9,f-8,f-1
<== Activation 0      B5: f-9,f-8,f-1
==> f-10    (bagged (item-name pepsi) (bag-name bag1))
==> Activation 10     log-bagged: f-10
<== f-1     (current-bag (name bag1) (item-count 0) (item-size nil))
<== Activation 0      B4: f-9,f-4,f-1
<== Activation 0      B5: f-9,f-4,f-1
==> f-11    (current-bag (name bag1) (item-count 1) (item-size large))
==> Activation 0      B5: f-9,f-4,f-11
==> Activation 0      B4: f-9,f-4,f-11
FIRE    5 log-bagged: f-10
pepsi in bag1
FIRE    6 B4: f-9,f-4,f-11
<== f-4     (item (name granola) (container-type cardboard-box) (size large)
(frozen no))
<== Activation 0      B5: f-9,f-4,f-11
==> f-12    (bagged (item-name granola) (bag-name bag1))
==> Activation 10     log-bagged: f-12
<== f-11    (current-bag (name bag1) (item-count 1) (item-size large))
==> f-13    (current-bag (name bag1) (item-count 2) (item-size large))
FIRE    7 log-bagged: f-12
granola in bag1
FIRE    8 B6: f-9
<== f-9     (step bag-large-items)
==> f-14    (step bag-medium-items)
==> Activation 0      B10: f-14
```

```
==> Activation 0      B9: f-14,f-2,f-13
==> Activation 0      B9: f-14,f-5,f-13
==> Activation 0      B9: f-14,f-6,f-13
==> Activation 0      B7: f-14,f-5,*
FIRE    9 B7: f-14,f-5,*
==> f-15    (in-freezer-bag ice-cream)
==> Activation 10     log-in-freezer-bag: f-15
FIRE   10 log-in-freezer-bag: f-15
ice-cream in freezer bag
FIRE   11 B9: f-14,f-2,f-13
<== f-13    (current-bag (name bag1) (item-count 2) (item-size large))
<== Activation 0      B9: f-14,f-6,f-13
<== Activation 0      B9: f-14,f-5,f-13
==> f-16    (current-bag (name bag2) (item-count 0) (item-size nil))
==> Activation 0      B9: f-14,f-6,f-16
==> Activation 0      B9: f-14,f-5,f-16
==> Activation 0      B9: f-14,f-2,f-16
==> Activation 0      B8: f-14,f-6,f-16
==> Activation 0      B8: f-14,f-5,f-16
==> Activation 0      B8: f-14,f-2,f-16
FIRE   12 B8: f-14,f-6,f-16
<== f-6     (item (name potato-chips) (container-type plastic-bag) (size
medium) (frozen no))
<== Activation 0      B9: f-14,f-6,f-16
==> f-17    (bagged (item-name potato-chips) (bag-name bag2))
==> Activation 10     log-bagged: f-17
<== f-16    (current-bag (name bag2) (item-count 0) (item-size nil))
<== Activation 0      B8: f-14,f-2,f-16
<== Activation 0      B8: f-14,f-5,f-16
<== Activation 0      B9: f-14,f-2,f-16
<== Activation 0      B9: f-14,f-5,f-16
==> f-18    (current-bag (name bag2) (item-count 1) (item-size medium))
==> Activation 0      B9: f-14,f-5,f-18
==> Activation 0      B9: f-14,f-2,f-18
==> Activation 0      B8: f-14,f-5,f-18
==> Activation 0      B8: f-14,f-2,f-18
FIRE   13 log-bagged: f-17
potato-chips in bag2
FIRE   14 B8: f-14,f-5,f-18
<== f-5     (item (name ice-cream) (container-type cardboard-carton) (size
medium) (frozen yes))
<== Activation 0      B9: f-14,f-5,f-18
==> f-19    (bagged (item-name ice-cream) (bag-name bag2))
==> Activation 10     log-bagged: f-19
<== f-18    (current-bag (name bag2) (item-count 1) (item-size medium))
<== Activation 0      B8: f-14,f-2,f-18
<== Activation 0      B9: f-14,f-2,f-18
==> f-20    (current-bag (name bag2) (item-count 2) (item-size medium))
==> Activation 0      B9: f-14,f-2,f-20
==> Activation 0      B8: f-14,f-2,f-20
FIRE   15 log-bagged: f-19
ice-cream in bag2
FIRE   16 B8: f-14,f-2,f-20
<== f-2     (item (name bread) (container-type plastic-bag) (size medium)
(frozen no))
<== Activation 0      B9: f-14,f-2,f-20
==> f-21    (bagged (item-name bread) (bag-name bag2))
==> Activation 10     log-bagged: f-21
<== f-20    (current-bag (name bag2) (item-count 2) (item-size medium))
==> f-22    (current-bag (name bag2) (item-count 3) (item-size medium))
FIRE   17 log-bagged: f-21
bread in bag2
FIRE   18 B10: f-14
```

```
<== f-14    (step bag-medium-items)
==> f-23    (step bag-small-items)
==> Activation 0      B13: f-23
==> Activation 0      B12: f-23,f-3,f-22
FIRE   19 B12: f-23,f-3,f-22
<== f-22    (current-bag (name bag2) (item-count 3) (item-size medium))
==> f-24    (current-bag (name bag3) (item-count 0) (item-size nil))
==> Activation 0      B12: f-23,f-3,f-24
==> Activation 0      B11: f-23,f-3,f-24
FIRE   20 B11: f-23,f-3,f-24
<== f-3     (item (name glop) (container-type jar) (size small) (frozen no))
<== Activation 0      B12: f-23,f-3,f-24
==> f-25    (bagged (item-name glop) (bag-name bag3))
==> Activation 10     log-bagged: f-25
<== f-24    (current-bag (name bag3) (item-count 0) (item-size nil))
==> f-26    (current-bag (name bag3) (item-count 1) (item-size small))
FIRE   21 log-bagged: f-25
glop in bag3
FIRE   22 B13: f-23
<== f-23    (step bag-small-items)
==> f-27    (step done)
CLIPS>
```

Without the trace you get the following:

```
CLIPS> (load "C:/Users/Doug/Desktop/bagger.CLP")
Defining deftemplate: item
Defining defglobal: prev-bag-number
Defining deffunction: new-bag-name
Defining deftemplate: current-bag
Defining deftemplate: bagged
Defining deffacts: initial
Defining defrule: B0 +j+j
Defining defrule: B1 +j+j+j+j
Defining defrule: B2 =j+j
Defining defrule: B3 +j+j+j+j
Defining defrule: B4 =j+j+j+j
Defining defrule: B5 =j=j+j+j
Defining defrule: B6 =j+j
Defining defrule: B7 +j+j+j+j
Defining defrule: B8 =j+j+j+j
Defining defrule: B9 =j=j+j+j
Defining defrule: B10 =j+j
Defining defrule: B11 +j+j+j+j
Defining defrule: B12 =j=j+j+j
Defining defrule: B13 =j+j
Defining defrule: log-bagged +j+j
Defining defrule: log-in-freezer-bag +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Bagger Production System
Do you want a bottle of Pepsi? yes
pepsi in bag1
granola in bag1
ice-cream in freezer bag
potato-chips in bag2
ice-cream in bag2
bread in bag2
glop in bag3
CLIPS>
```