

USING CLIPS

1. INTRODUCTION:

In this presentation, we will describe a great deal of CLIPS structure and commands, and show the details of building an expert system solution. Using CLIPS version 6.2, we will take the Coins example discussed before to demonstrate the use of CLIPS and its commands. By the end of this presentation, we would be in a position to start solving main problems. The following flow chart shows the organization of this presentation.

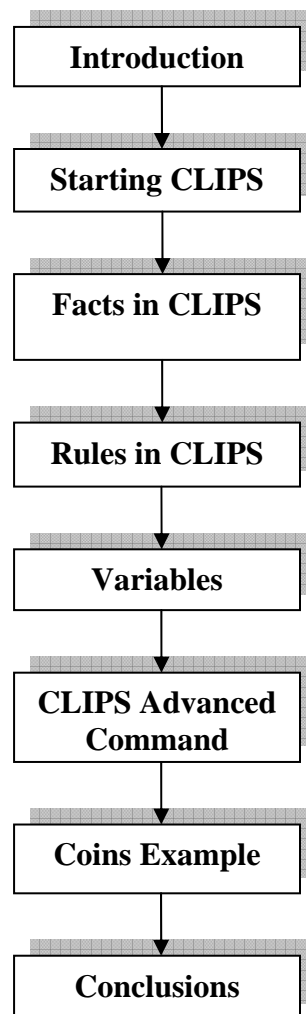


Figure 1: Presentation Flow Chart

2. STARTING CLIPS:

If we are using the DOS environment, to begin CLIPS, you have to enter the full path of the directory where the executable file resides. You should see the CLIPS prompt:

```
CLIPS>
```

Now, you can start entering commands directly into CLIPS. But if you are using CLIPS windows version, you just have to click on the executable file icon. Then you will be able to see CLIPS Graphical User Interface (GUI) as shown in Figure 2.

Once the GUI starts, you can directly enter commands. In the GUI version you can choose most of the commands from complete window by typing the first letter of the command and then pressing ctrl+j (see Figure 3). The command completion window will appear with several commands that you can choose between them by mouse.

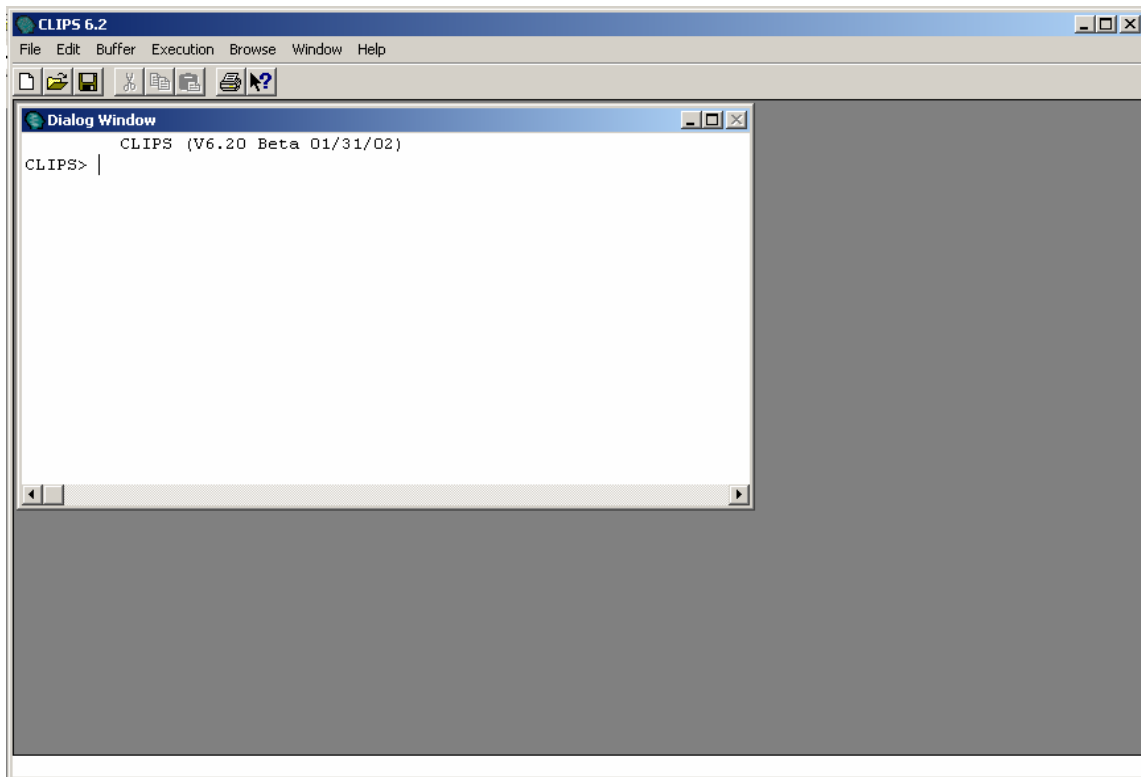


Figure 2: CLIPS Graphical User Interface.

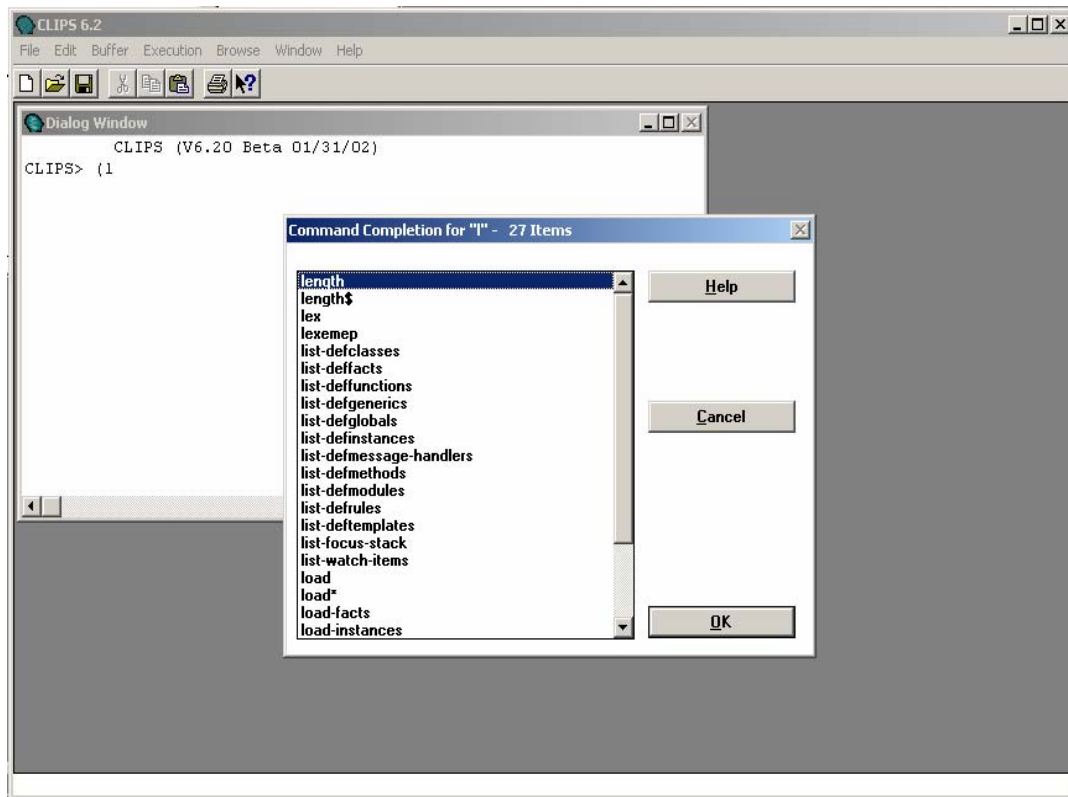


Figure 3: Command Completion Window.

All CLIPS commands should be between two parentheses. Finally, to end the CLIPS program just click on the close button in window version you can also enter **exit** command in both DOS version and windows version as following:

```
CLIPS> (exit)
```

This command will end CLIPS. You have to be careful if you are using the DOS version under Microsoft Windows operating system since in this case you have to use **exit**. If you do not use **exit**, you will be closing the running DOS application without ending it. This can cause an error in your windows operating system.

3. FACTS IN CLIPS:

To enter a fact into fact-list in CLIPS use **assert** command. This command enters the fact and let CLIPS give it an identifier which is unique. An example on **assert**, by entering the following statement right after CLIPS prompt as shown:

```
CLIPS> (assert (coin))
```

After this, you have to press **enter** key for executing the command. Once the command is executed you will see the response

```
<Fact-0>
```

This means that the fact has been entered and has been given an identifier 0. CLIPS will give all facts which will be entered after this one an identifier in an increasing way.

Now, if you want to see your fact-list. If you are using the Windows version 6.2, you can see them in fact window. Also you can see the fact-list in all versions by using **facts** command as shown:

```
CLIPS> (facts)
f-0 (coin)
For a total of 1 fact.
CLIPS>
```

For deleting the fact-list there are several commands. The **clear** command is common in this situation but this command clears the fact-list from all facts which makes it a dangerous command in some cases. In other words, it restores CLIPS to its startup state. There is also **reset** command which does the same like **clear** command but the difference is that **reset** command inserts **initial-fact** into the list after deleting all facts and gives it the first identifier number 0. An example of both commands is listed as follow:

```
CLIPS> (reset)
CLIPS> (facts)
f-0 (initial-fact)
For a total of 1 fact.
CLIPS> (assert (coin))
<Fact-1>
CLIPS> (facts)
```

```
f-0 (initial-fact)
f-1 (coin)
For a total of 2 facts.
CLIPS> (clear)
CLIPS> (facts)
```

Now, let us try to enter two facts with the same name into the fact-list to see what is going to happen:

```
CLIPS> (assert (coin))
<Fact-0>
CLIPS> (assert (coin))
FALSE
CLIPS>
```

There is a FALSE message returned from CLIPS which means that there is no way to enter two facts with the same name. This situation is the default for CLIPS but if you insist on entering two facts with the same name, you can use **set-fact-duplication** command that allows duplicate fact.

There is a possibility to see the fact you interested in. In other words you can see one fact in the fact-list without seeing all other facts. That can be done by entering the **fact-index** number after **facts** command as follow:

```
CLIPS> (clear)
CLIPS> (assert (a) (b) (c))
<Fact-2>
CLIPS> (facts 1)
f-1 (b)
f-2 (c)
for a total of 2 facts.
CLIPS> (facts 1 1)
f-1 (b)
for total of 1 fact.
CLIPS>
```

As we have seen in the example above, we can enter more than one fact in the same **assert** command. But you have to keep in mind that there should be parentheses between each fact to separate facts from each other.

There are two different facts. Facts that have **single-field** which is like what we saw so far and **multi-field** facts are which like:

```
(gold coin)
```

This one can be inserted into CLIPS by writing the following:

```
CLIPS> (assert (gold coin))
<Fact-0>
CLIPS> (facts)
f-0 (gold coin)
For a total of 1 fact.
CLIPS>
```

You can use **nil** to give the fact multi-field properties but without actually entering values. So, it is like this, there is a difference between having an empty mail box and having no mail box.

There are many types of fields: **float**, **integer**, **symbol**, **string**, **external-address**, **fact-address**, **instance-name** and **instance-address**. These types indicate the values which are stored in it. Refer to *CLIPS user's guide* for more details.

A **symbol** is a field that starts with any ASCII character. CLIPS is case-sensitive and some symbols have a special meaning like: “ () & | < ~ ; ? \$. There is no permeation to embed facts into other in CLIPS. For example the following is an illegal statement:

```
(coin (attribute gold circle))
```

However, we can use ‘-‘ to indicate the name of the same field like:

```
(coin-attribute gold circle)
```

String in CLIPS is like other programming languages. It should be between double quotes “ ”. Anything could be inside these quotes. Some examples on **string** type are listed below:

```
"coin"
"coin/gold"
"coin 2"
```

The **integer** and **float** types are more like other languages. They are both for numeric fields. But the difference is in the number nature. **Integer** is for numbers which have no decimal point while **float** is for floating point or real numbers. You have to keep in mind that you cannot start the field with a number. An example on numeric fields is shown below:

```
CLIPS> (assert (a 1))
<Fact-0>
CLIPS> (assert (b -2))
<Fact-1>
CLIPS> (assert (c 3.5e5))
<Fact-2>
CLIPS> (facts)
f-0 (a 1)
f-1 (b -2)
f-3 (c 350000.0)
For a total of 3 facts.
CLIPS>
```

All fields of a fact should be separated by **space**, **enter**, **tabs** or **linefeeds**. But you have to be careful on using the **enter key** inside of a string.

```
CLIPS> (assert (the coin is gold))
<Fact-0>
CLIPS> (facts)
f-0 (the coin is gold)
For a total of 1 fact.
CLIPS> (clear)
CLIPS> (assert ( the coin is gold))
<Fact-0>
CLIPS> (facts)
f-0 (the coin is gold)
For a total of 1 fact.
```

Also, CLIPS is case-sensitive. This means that (coin) is a different fact from (Coin). It is a good way of writing the facts with **relation**. **Relation** means that you put the first field name to describe the relationship between all fields. The case-sensitivity, **relation** and the enter key are good ways to improve the readability of the fact-list.

```

CLIPS> (clear)
CLIPS> (assert (coin-attribute
                Gold
                Circle
                2.5cm))
<Fact-0>
CLIPS> (facts)
f-0 (coin-attribute Gold Circle 2.5cm)
For a total of 1 fact.
CLIPS>

```

If you want to use the double quotes in a field, you have to put them after **backslash** “\” as the following:

```

CLIPS> (assert (the coin is "\"gold\""))
<Fact-0>
CLIPS> (facts)
f-0 (the coin is "\"gold\"")
For a total of 1 fact.

```

We learned so far how to enter the facts. But how to remove them from the fact-list? One command known in CLIPS for this action is **retract** command. This command takes the fact index as an argument for retraction. For example make the following fact-list:

```

CLIPS> (assert (the coin is gold))
<Fact-0>
CLIPS> (assert (the coin is circle))
<Fact-1>
CLIPS> (assert (the coin size is 2.1cm))
<Fact-2>
CLIPS> (facts)
f-0 (the coin is gold)
f-1 (the coin is circle)
f-2 (the coin size is 2.1cm)
For a total of 3 facts.
CLIPS>

```

Now if you want to remove the second fact just enter the **retract** command and the fact index which is f-1 as an argument as follows:

```
CLIPS> (retract 1)
CLIPS> (facts)
f-0 (the coin is gold)
f-2 (the coin size is 2.1cm)
For a total 2 facts.
CLIPS>
```

You can also retract more than one fact at the same time by typing the facts indices. Be careful to give non-existent fact index because this will give an error message. For retracting all facts at once (**retract ***) should be used. An example is shown below:

```
CLIPS> (retract 0 2)
CLIPS> (facts)
CLIPS> (assert (the coin is gold))
<Fact-0>
CLIPS> (assert (the coin is circle))
<Fact-1>
CLIPS> (assert (the coin size is 2.1cm))
<Fact-2>
CLIPS> (facts)
f-0 (the coin is gold)
f-1 (the coin is circle)
f-2 (the coin size is 2.1cm)
For a total of 3 facts.
CLIPS> (retract *)
CLIPS> (facts)
CLIPS>
```

There are several commands in CLIPS which allow you to debug the program more easily. One of these commands is **watch facts** command which allows you to see all asserted and retracted facts. This is better way than writing (facts) command over and over again.

It is possible to stop this command from working by typing the **unwatch facts** command. The following is a solved example:

```
CLIPS> (watch facts)
CLIPS> (assert (coin is gold))
==> f-0      (coin is gold)
<Fact-0>
```

```

CLIPS> (retract 0)
<== f-0      (coin is gold)
CLIPS> (unwatch facts)
CLIPS> (assert (coin is gold))
<Fact-1>
CLIPS>

```

As seen in the previous example, the **right double arrow** `==>` means that a fact has been asserted while the **left double arrow** `<==` means that the fact has left the facts-list. There are a number of **watch** commands for watching several things. The **semicolon** “;” is being used for comments in CLIPS. And for stop the **watch** command you have to enter the **unwatch** command. Here are some **watch** commands:

```

(watch facts)
(watch instance)           ;used with objects
(watch slots)             ;used with objects
(watch rules)
(watch message)           ;used with objects
(watch message-handlers) ;used with objects
(watch all)

```

CLIPS has its own on-line help file. To see the help just type (**help**) and press the enter key. If an error message appears telling you that CLIPS can not find the clips.hlp file use the (**help-path**) command to discover where CLIPS expected the file to be. More details on this can be found in *CLIPS user guide*.

4. RULES IN CLIPS:

There must be rules in all expert systems. Actually, the most important difference between the expert systems and procedural languages are the rules. Rules in expert systems are similar to IF-THEN statement in procedural languages like C, Ada and Basic. An example of IF-THEN statement is:

```

IF coin is bronzy THEN  the coin is 1P

```

To write a rule in CLIPS you have to use **defrule** command. You can write the entire rule in the same line but it is more likely to write every part of the rule in different line. The following is an example on coins. To make the example without errors you have to enter a fact first as follow:

```
CLIPS> (assert (coin is bronzy))
<fact-0>
CLIPS> (defrule coin
        (coin is bronzy)
=>
        (assert (coin is 1p))
)
CLIPS>
```

If you do not see any error messages that mean you made the example right. After running the example you will see the second fact assert into the fact list as follow:

```
CLIPS> (run)
CLIPS> (facts)
f-0 (coin is bronzy)
f-1 (coin is 1p)
for a total of 2 facts.
CLIPS>
```

Be careful, if you try to run the program again nothing will happen. That is because no new pattern entity was asserted, i.e. to run the program again you have to reassert the fact again. This is a principle in CLIPS.

So, from the previous example we see how the rules can be entered in CLIPS. The following is the general shape of the rule statement:

```
(defrule rule-name "optional comment"
  (Pattern-1)          ; Left-Hand Side (LHS)
  (Pattern-2)
  .
  .
  (Pattern-n)
=>
  (Action-1)          ; Right-Hand Side (RHS)
```

```
(Action-2)
.
.
(Action-m)
)
```

Zero or more pattern can be in the rule at the LHS. The pattern and the actions in the rule do not have to be equal. The “=>” in the rule is the start of THEN section in the rule which can be called **arrow** or **implies**. The LHS is the part before the arrow and the RHS is the part after. To activate the rule all patterns in the RHS should be matched with the facts in the fact-list. When this happens, the rules go into the **agenda**. The **agenda** is the place where all rules which have been matched are being ordered by priority or **salience**. To see what is in the agenda just type (**agenda**) as follow:

```
CLIPS> (agenda)
0    coin: f-0
for a total of 1 activation.
CLIPS>
```

The number “0” is the salience of the activation. “f-0” is the fact-identifier of the fact that matches the rule. The default number for salience is 0 while the range for it is between -10,000 to +10,000.

To save your work you have to enter the **save** command. This command saves the rules we have entered in CLIPS window. An example follows:

```
CLIPS> (save coin-test.clp)
TRUE
CLIPS>
```

The saved file will be in the same folder with CLIPS executable file. If you want to change the place of the file, just enter the full path for it as follow:

```
CLIPS> (save c:\coin-test.clp)
TRUE
CLIPS>
```

The “.clp” extension is for reminding us that this is a CLIPS source code file. You can use any text editor to write your code but remember to save it as “.clp” file or “.txt” file.

You can watch the activations like watching facts. That is done by entering the **watch activations** command into CLIPS. The right arrow is to indicate entering of fact or activation while the left arrow is to indicate leaving of the fact or activation as in the next example:

```
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (defrule coin
        (coin-is bronzy)
=>
        (printout t "coin is 1p" crlf)
)
CLIPS> (assert (coin-is bronzy))
==> f-0      (coin-is bronzy)
==> Activation 0      coin: f-0
<Fact-0>
CLIPS> (agenda)
0      coin: f-0
For a total of 1 activation.
CLIPS> (run)
coin is 1p
CLIPS> (facts)
f-0      (coin-is bronzy)
For a total of 1 fact.
CLIPS> (run)
CLIPS>
```

You can make the rule fire again if you retract the fact and assert it again. If you want to see the rule when you are in CLIPS, you have to use **ppdefrule** command. This command takes the rule name as an argument. Also you can see the name of the rules by typing **rules** command in CLIPS. The following example shows who these commands works:

```
CLIPS> (ppdefrule coin)
(defrule MAIN::coin
  (coin-is bronzy)
```

```
=>
  (printout t "coin is 1p" crlf))
CLIPS> (rules)
coin
For a total of 1 defrule.
CLIPS>
```

Actions are not the only thing you can write in the RHS. You can call functions in this part of the rule too. One example of functions is the **printout** command. This command allows you to print statements in CLIPS window. There is some argument in this command like “t” which means the output of the command should be in the standard output device (screen most of time). And the “crlf” word means that it leaves empty line after it. For sure you can use more than one command in the same rule. The output text should be in double quotes. You may not include the “crlf” in the command if you do not want to put an empty line.

After saving all of your work you can load it again using the **load** command. There is also a save command that allows you to save in binary which is **bsave**. You have to load it using the **blogd** command and this command makes it faster to save and load because of binary nature of data. To learn more details on this topic you can see the *CLIPS Reference manual* and *CLIPS user guide*.

5. VARIABLES:

Like most of other programming languages, CLIPS has variables for saving data in it. To define a variable, you should write a question mark before the variable name as follows:

```
?<variable name>
```

The variable should have a meaning to increase the readability. Here are some examples:

```
?x          ?coin-color      ?sensor  ?value
```

In CLIPS all variables should have a value before use. This condition is different in global variables. The following example shows an error message for using a variable before assignment:

```
CLIPS> (defrule coin
        (coin-is gold)
=>
        (printout t ?x crlf)
)
```

```
[PRCCODE3] Undefined variable x referenced in RHS
of defrule.
```

```
ERROR:
(defrule MAIN::coin
  (coin-is gold)
=>
  (printout t ?x crlf))
```

```
CLIPS>
```

The common use of variable is to match facts in LHS then assert some facts depends on the match at the RHS. For example:

```
CLIPS> (defrule coin
        (coin-is ?color)
=>
        (assert (color-is ?color))
)
CLIPS> (assert (coin-is gold))
<Fact-0>
CLIPS> (run)
CLIPS> (facts)
f-0      (coin-is gold)
f-1      (color-is gold)
For a total of 2 facts.
CLIPS>
```

Now something important we have mention here. It is about facts, as we can see from previous examples we did use dash “-“ in the fact name. This is allowed but there is difference between the facts which have dash in the name and the facts without it. The difference is that dash makes the name one field but without it the fact contains two fields. Let us try to enter two facts with same name with and without the dash:

```
CLIPS> (clear)
CLIPS> (assert (coin-is gold))
<Fact-0>
CLIPS> (assert (coin is gold))
<Fact-1>
CLIPS>
```

So, no error message appears because the first fact has only two fields while the second one has three fields which is “coin”, ”is” and “gold”. In other words, the fact can be **single** field or **multi-field**.

Let us get back to variables, they can be used in functions too like **printout**. You can use them before or after the text. It is possible to print only variables in CLIPS by writing only variable name in printout command. All of this can be done as follow:

```
CLIPS> (clear)
CLIPS> (defrule coin
      (coin-is ?gold)
=>
      (printout t "coin color is " ?gold crlf)
)
CLIPS> (assert (coin-is gold))
<Fact-0>
CLIPS> (run)
coin color is gold
CLIPS> (clear)
CLIPS> (defrule coin
      (coin-is ?color)
=>
      (printout t ?color crlf)
)
CLIPS> (assert (coin-is bronzy))
```

```
<Fact-0>
CLIPS> (run)
bronzy
CLIPS>
```

You can for sure use more than one variable in the same rule in more complex cases:

```
CLIPS> (clear)
CLIPS> (defrule coin
        (coin-is ?color ?shape)
=>
        (printout t "the coin color is " ?color "
and the coin shape is " ?shape crlf)
)
CLIPS> (assert(coin-is gold circle))
<Fact-0>
CLIPS> (run)
the coin color is gold and the coin shape is circle
CLIPS> (assert (coin-is hex silver))
<Fact-1>
CLIPS> (run)
the coin color is hex and the coin shape is silver
CLIPS>
```

Notice that order is very important in facts. As you see from the previous example, there is difference in the second assertion than the first one.

In CLIPS the retraction is very important. You can use the variable to retract some fact. That can be done by giving the retract command the fact-address as an argument. In this case the variable will not store what is in the field as the pervious example but it will store only the fact-address without even knowing what the fact is. To assign the fact-address into a variable you have to use “<-“ sing by typing “<” first then “-“. Example is below:

```
CLIPS> (clear)
CLIPS> (assert (gold coin))
<Fact-0>
CLIPS> (defrule coin
        ?coin <- (gold coin)
```

```

=>
    (printout t "the coin is gold" crlf)
    (retract ?coin)
)
CLIPS> (run)
the coin is gold
CLIPS> (facts)
CLIPS>

```

So, when you use the facts command, it shows you that there are no facts in the facts-list. For sure you can use more than one variable in the same rule. In the following example the **deffact** command is used to define more than one fact at the same time:

```

CLIPS> (clear)
CLIPS> (defrule coin
    ?coin <- (coin-is ?color)
=>
    (printout t "coin color is " ?color crlf)
    (retract ?coin)
)
CLIPS> (deffacts coin-attrib
    (coin-is gold)
    (coin-is silver)
    (coin-is bronzy))
CLIPS> (reset)
CLIPS> (run)
coin color is bronzy
coin color is silver
coin color is gold
CLIPS>

```

The rule fired with all matching facts. The **reset** command does a lot of functions like: clearing the facts-list, asserting all facts in **deffacts**, defining all rules in the program and some more functions that are not included in our study.

The question mark can be used as **wildcard** in CLIPS too. An example is, if you know the first field of the fact but you are not sure about the others. So you just type the first field name and put "?" after it. Remember that you

can put more than one in the same rule depending on the number of the fields you do not know. Also the place of the question mark makes a difference. The following shows the idea:

```
CLIPS> (defrule searching-coin
        (coin shape ?)
=>
        (printout t "the right coin" crlf)
)
CLIPS> (deffacts coins
        (coin shape is hex)
        (coin shape hex)
        (coin circle)
        (coin is circle)
)
CLIPS> (reset)
CLIPS> (run)
the right coin
CLIPS>
```

As you see from the above example, the only fired fact was the second one because of the all matching fields. Even when the last field was not specified. Here you can choose the place of the non-specified fields and the number of them. Just type like the follow:

```
(coin ? gold)
(coin ? ? circle)
(coin ? is ?)
```

And so on. But what if you want to find a fact without knowing the number of the fields in it? One way to solve this is to write more than one rule each time you add one more question mark. It seems that you will have many matching possibilities. The easiest way to do this is by writing the dollar-sign "\$" before the question mark. This will make all facts that match the known fields in the rule fire. Example is listed below:

```
CLIPS> (clear)
CLIPS> (defrule coins
        (coin shape $?)
=>
```

```

        (printout t "The right coin" crlf)
    )
CLIPS> (deffacts coins
        (coin shape is hex)
        (coin shape hex)
        (coin circle)
        (coin is circle)
    )
CLIPS> (reset)
CLIPS> (run)
The right coin
The right coin
CLIPS> (facts)
f-0      (initial-fact)
f-1      (coin shape is hex)
f-2      (coin shape hex)
f-3      (coin circle)
f-4      (coin is circle)
For a total of 5 facts.
CLIPS>

```

So, the first and the second facts fire because the rule matches. Even if the number of the field is not specified in the rule. The question mark and the dollar-sign are very useful in CLIPS for a lot of purposes.

But, what if you want to see the name of the field that matches the rule? In CLIPS you can use the wildcards for this situation too. The wildcard can be assigned to any symbolic field such as ?x, \$?x, ?name or \$?shape. Using the "\$" makes multi-fields assigned to the variable. An example follows below:

```

CLIPS> (defrule coin
        (coin shape $?shape)
=>
        (printout t "The right coin " ?shape crlf)
    )
CLIPS> (run)
The right coin (hex)
The right coin (is hex)
CLIPS>

```

Now if you want all facts that have “is” some where in it to fire. It is not hard to do specially after knowing how to use the “?”. It can be solved as follows:

```
CLIPS> (defrule coins
        (coin $?before is $?after)
=>
        (printout t "The right coin" ?before
?after crlf)
)
CLIPS> (deffacts coins
        (coin shape is hex)
        (coin shape hex)
        (coin circle)
        (coin is circle)
)
CLIPS> (reset)
CLIPS> (run)
The right coin()(circle)
The right coin(shape)(hex)
CLIPS>
```

Notice that you can use the same name of the rule and the fact because they are two different commands. Also, if there is no value, it will still print empty parentheses. The multi-fields wildcard can match zero or more fields while the single field wildcard should match one field only. The single and multi-fields wildcards can come together in the same rule like:

```
(coin ? is $?)
```

The use of multi-fields wildcard is only for unknown length of the fields. Using this wildcard in a wrong way may cause much inefficiency because of increased memory requirements.

6. CLIPS ADVANCED COMMANDS:

So far, we have learned the very basics of CLIPS. Now we will discuss some of CLIPS advanced commands that help us in our problems. Let us start with the “**and**”, “**or**” and “**not**” logic gates in CLIPS. In CLIPS like many other languages there are logical operators to handle logic operations. The

tilde “~” in CLIPS indicate the not logic. It will not allow the value that comes after it. For example:

```
CLIPS> (clear)
CLIPS> CLIPS> (defrule not-gold
              (coin-is ~gold)
=>
              (printout t "The coin is not gold" crlf)
)
CLIPS> (deffacts coins
        (coin-is silver)
        (coin-is gold)
        (coin-is bronzy)
)
CLIPS> (reset)
CLIPS> (agenda)
0      not-gold: f-3
0      not-gold: f-1
For a total of 2 activations.
CLIPS> (run)
The coin is not gold
The coin is not gold
CLIPS>
```

As you see, the rule fired two times only when the fact was not gold. By using this tilde you can do work of many other rules.

There is also the “or” operator in CLIPS. The bar “|” gives the meaning of “or”. With this operator you can make the program fire a rule if one value only or group of values that matches it. In the next example, we are trying to make the program fire the rule if the color is gold or the shape is hexagon:

```
CLIPS> (clear)
CLIPS> (CLIPS> (defrule coin
              (coin-is gold|hex)
=>
              (printout t "The right coin" crlf)
)
CLIPS> (deffacts coins
        (coin-is gold)
```

```

        (coin-is silver)
        (coin-is circle)
        (coin-is hex)
    )
CLIPS> (reset)
CLIPS> (agenda)
0      coin: f-4
0      coin: f-1
For a total of 2 activations.
CLIPS> (run)
The right coin
The right coin
CLIPS>

```

The rule fired two times with the matching facts. As you see at the agenda, the first and last facts were the only two matching facts. The **ampersand** “&” in CLIPS indicate the “**and**” operator. The “&” is usually used in conjunction with other operators. Suppose you want to know the color of the coin that matches the rule, you may write the following:

```

CLIPS> (clear)
CLIPS> CLIPS> (defrule coin
              (coin-is ?color & gold|hex)
=>
              (printout t "The coin is " ?color crlf)
)
CLIPS> (deffacts coin
        (coin-is gold)
        (coin-is silver)
        (coin-is hex)
        (coin-is circle)
)
CLIPS> (reset)
CLIPS> (agenda)
0      coin: f-3
0      coin: f-1
For a total of 2 activations.
CLIPS> (run)
The coin is hex
The coin is gold
CLIPS>

```

It is time to start reading data given by the user after. The **read** command is used for this. We put this command in **assert** command as follows:

```
CLIPS> (defrule coin-color
        (initial-fact)
=>
        (printout t "What is the coin color?")
        (assert (color (read)))
)
CLIPS> (defrule read-color
        ?color-fact <- (color ?color &
gold|sivler|bronzy)
=>
        (retract ?color-fact)
        (printout t "correct color" crlf)
)
CLIPS> (reset)
CLIPS> (run)
What is the coin color?gold ;input from user
correct color
CLIPS> (reset)
CLIPS> (run)
What is the coin color?red
CLIPS> ; wrong answer
```

This is one way of reading data in CLIPS. There is another advanced command, namely the **switch** command. This command can make the rule choose between several values inside it. The following example shows its use:

```
CLIPS> (defrule coins
        (initial-fact)
=>
        (printout t "What is the color of the
coin? " )
        (assert (color (read)))
)
CLIPS> (defrule coin-color
        (color ?val)
=>
```

```

        (switch (lowercase ?val)
            (case gold then
                (printout t "the coin is gold"
crlf))
            (case silver then
                (printout t "the coin is
silver" crlf))
            (case bronzy then
                (printout t "the coin is
bronzy" crlf))
            (default none)
        )
    )
CLIPS> (reset)
CLIPS> (run)
What is the color of the coin? gold
the coin is gold
CLIPS> (reset)
CLIPS> (run)
What is the color of the coin? bronzy
the coin is bronzy
CLIPS> (reset)
CLIPS> (run)
What is the color of the coin? silver
the coin is silver
CLIPS> (reset)
CLIPS> (run)
What is the color of the coin? red
CLIPS> ;wrong enter

```

Finally, we have to talk about the **lowercase** function. This function makes all entry in lower case because CLIPS is case sensitive. Now, we are ready to write the whole coin example and solve it which is the topic of the next section.

7. COINS RECOGNITION EXAMPLE:

After introducing CLIPS basic commands and structure we will solve the coins example. The purpose of the coin example is to make the expert system to identify the coins by asking about its properties. After it gets all answers it needs, it tells us what the coin is. The example explains the CLIPS structure and commands used. First we have to put the property table to be able to solve the problem. (See Table 1 below).

Coins	Color	Shape	Size
1 Dinar	Gold	Circle	2.4 cm
1 Dinar	Gold	Hexagon	3.1 cm
½ Dinar	Gold	Hexagon	2.8 cm
½ Dinar	Silver & gold	Hexagon	2.8 cm
¼ Dinar	Gold	Hexagon	2.6 cm
10 Piastres	Silver	Circle	2.7 cm
5 Piastres	Silver	Circle	2.5 cm
2.5 Piastres	Silver	Circle	2.1 cm
1 Piastres	Bronzy	Circle	2.4 cm

Table 1: Coin Attributes

Now we have to put the decision tree for the problem which we will use to design our program see Figure 4 below.

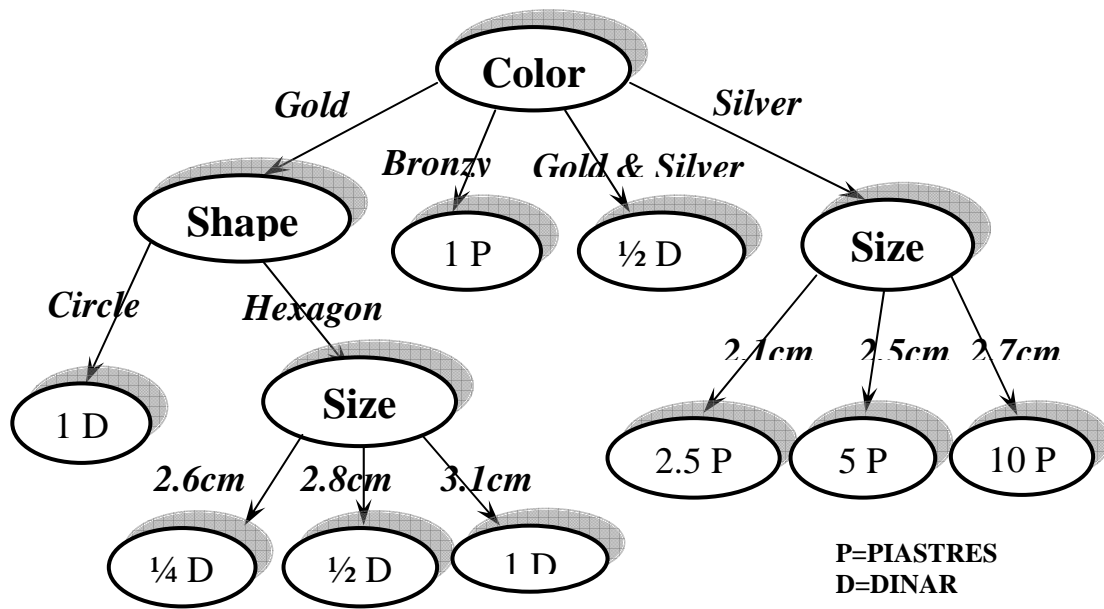


Figure 4: Decision Tree of Coin Example.

After studying the attribute table and the decision tree of the coins we can start writing the program with the basic knowledge we have. So let us start the program as follows:

```
(defrule start-sys
  (initial-fact)
=>
  (printout t "          Coins Recognition Expert
System" crlf crlf)
  (printout t "          **PLEASE ENTER THE NUMBER
OF THE CHOICE ONLY**" crlf crlf)
  (printout t "What color is the coin?(a) Gold /
(b) Silver / (c) Both /(d) bronzy)>")
  (assert (ans-color (read)))
)

(defrule silver-color
  ?ans-color-ret <- (ans-color b|B)
=>
  (printout t "          What size is the
coin?(a)2.1cm /(b)2.5cm /(c)2.7cm >")
  (assert (ans-size (read)))
  (retract ?ans-color-ret)
)

(defrule silver-size
  (ans-size ?val)
=>
  (switch (lowercase ?val)
    (case a then (printout t "          *The
coin is 2.5 Piastres*"crlf))
    (case b then (printout t "          *The
coin is 5 Piastres*" crlf))
    (case c then (printout t "          *The
coin is 10 Piastres*" crlf))
    (default none)
  )
)

(defrule silver-gold-color
  ?ans-color-ret <-(ans-color c|C)
=>
```

```

    (printout t "          *The Coin is 1/2 Dinar*"
  crlf)
    (retract ?ans-color-ret)
  )
  (defrule bronzy
    ?ans-color-ret <-(ans-color d|D)
=>
    (printout t "          *The coin is 1 Piastres*"
  crlf)
    (retract ?ans-color-ret)
  )
  (defrule gold-color
    ?ans-color-ret <-(ans-color a|A)
=>
    (printout t "          What shape is the
  coin?(a)circle /(b)hexagon=>")
    (assert(ans-shape (read)))
    (retract ?ans-color-ret)
  )
  (defrule gold-shape-circle
    (ans-shape a|A)
=>
    (printout t "          *The Coin is 1 Dinar*"
  crlf)
  )
  (defrule gold-shape-hex
    ?ans-shape-ret <- (ans-shape b|B)
=>
    (printout t "          What size is the
  coin?(a)2.6cm /(b)2.8cm /(c)3.1>" )
    (assert (ans-gold-size (read)))
    (retract ?ans-shape-ret)
  )
  (defrule gold-size
    ?ans-gold-size <-(ans-gold-size ?val)
=>
    (switch (lowercase ?val)
      (case a then (printout t "*The coin is 1/4
  Dinar*" crlf))

```

```
        (case b then (printout t "*the coin is 1/2
Dinar*" crlf))
        (case c then (printout t "*the coin is 1
Dinar*" crlf))
    )
    (retract ?ans-gold-size)
)
```

When this code is executed, the program will start to ask you about the coin color as the first attribute in the decision tree. Then based on your answer, the program will tell you the coin name or will ask you again to get more needed data. As seen from the decision tree the biggest number of questions is in the gold-hex side which means when you choose the gold color and hexagon shape. Some functions are used to make the program more powerful, faster and easier.