Man, Marriage and Machine – Adventures in Artificial Advice

Asheesh Goja

Table of Contents

Introduction 5

Background 6

What is an Expert System? 6

Programming an Expert system 7

- Writing “Hello world” in CLIPS 8
- Heuristic knowledge as Rules 8
- Rules with antecedents 8

Anatomy of an expert system 10

Execution of an expert system 11

Knowledge Engineering 12

Knowledge Representation 12

- Epistemology 13
- Productions 13
- Logic 14
- Formal Logic 14
- Methods of inference or reasoning 14
- Chaining 15

Knowledge representation in CLIPS 15

- Ordered Facts 16
- Unordered facts 16
- Rules 16
- Literal constraints 17
- Variable constraints 18
- And conditional element (CE) 19
- Or conditional Element 20

Limitations of Expert Systems 21

Socrates—the artificial advisor 22

- Domain Expertise 22
- Knowledge engineering “Socrates” 22

The backward chaining algorithm 23
Implementing the algorithm 24
Algorithm step 1-2 implementation 24
Algorithm step 3, 4, 5, 6 implementation 24
Algorithm step 7,8 implementation 26
Algorithm step 9 implementation (combining certainty) 27
The complete algorithm 28
Making the system interactive 29
Define the question template 29
Inferring the “age-difference” answer 29
Question with pre conditions 30
Implementing the question dependency rules 31
The complete algorithm 34
Making the system interactive with validation 34
Expressing user input constraints 34
Validating user input based on constraints 35
The complete algorithm 37
Making “Socrates” wiser 38
Extending the backward chaining algorithm 38
Adding Salience 39
Adding more questions 39
Running the wiser “Socrates” 40
Printing and formatting the results 40
The complete algorithm 41
Converting the knowledge base into a ”CLIPS” file 42
Organizing “a priori” questions 42
Organizing “a priori” domain rules 43
Creating rules to infer additional facts 44
Creating a rule of mark questions with pre conditions 44
Adding the rules to accept and validate user input 45
Implementing the backward changing algorithm for domain-rule 46
Implementing rules to resolve question dependencies 46

Combining confidence factors 47
Formatting and printing the final conclusion 47
Running the CLIPS file 47
Integrating CLIPS with C++ 48
Sample Code 48
Integrating CLIPS with C# using CLIPSNet 48
Sample Code 49
Integrating CLIPS with Java using JESS 49
Sample Code 49
The complete Socrates Expert System 50
CLIPS I/O router 50
Adding the router to CLIPS runtime 51
Embedding the rules file as a resource 52
Sample Code 52
Conclusion 52

Introduction

Athens—Greece year 460 BC; a young man contemplating wedlock, asks Socrates for his advice. The wise one replies: "*By all means, marry, if you get a good wife, you'll become happy; if you get a bad one, you'll become a philosopher*".

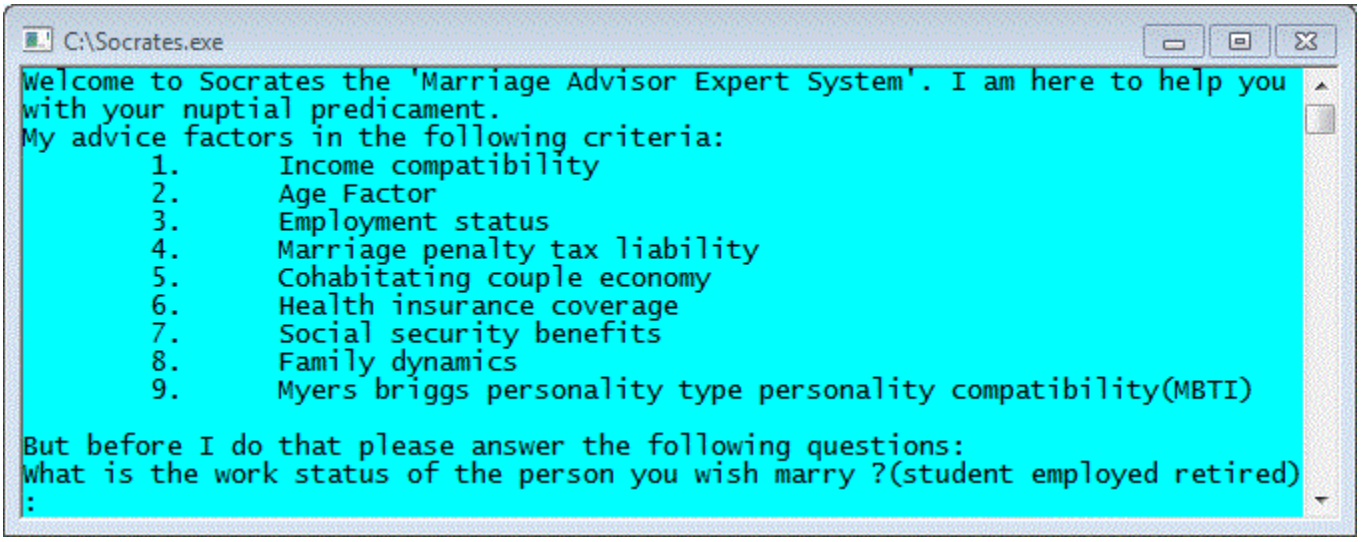
NY—NY year 2011; a young man asks "Socrates" the same question. The wise one replies: "*Based on income compatibility, age, employment status, marriage penalty tax liability and cohabitating couple economy; you have a low chance of a successful marriage.* Well "Socrates" is not communicating via a paranormal medium—it is an "Expert System".

While Ray Kurzweil's age of spiritual machines is still a future possibility, the age of **artificial advisors** is already here. These expert systems based advisors operate in diverse domains like:

- Reviewing mortgage loans
- Providing regional retail recommendations
- Evaluate carcinogenic potential of chemicals
- Assessing tax returns
- Assessing environmental impact
- Business rules engine

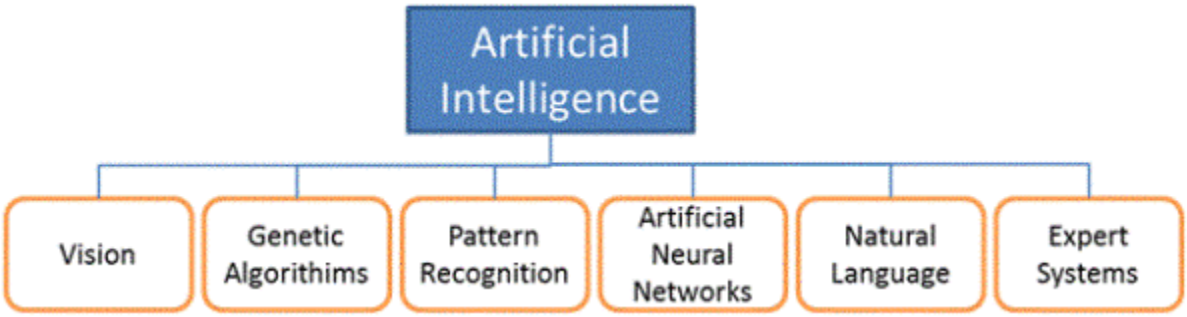
So I set out to write my own artificial advisor. Its goal is to provide an objective assessment of one's nuptial success, based on factors like MBTI, income compatibility, age, employment status, marriage penalty tax liability etc. To honor its wisdom I named it "Socrates".

"Download Socrates" and try it out. If you find it amusing then read on–I will show you how to build it.



Background

Year 1997, **Deep Blue** defeats chess grandmaster Gary Kasparov and this year **Watson** defeats jeopardy champions Ken Jennings and Brad Rutter. Deep Blue demonstrated solving extremely complex but logically well bounded problems like chess. Watson on the other hand solved the problem of meaning behind natural languages, which inherently is unbounded. Both Deep Blue and Watson are indicative of the great strides made in the field of Artificial Intelligence. AI-complete fields like natural language processing and open domain question answering are still in the realm of computer science research. It may take a while for such fields to become useful for mainstream commercial applications. One AI discipline that has matured and found applicability in diverse domains like internal medicine, accounting, finance, organic chemistry etc. is **Expert Systems**.



What is an Expert System?

An expert system is a computer system that emulates, or acts in all respects, with the decision-making capabilities of a human expert. An expert system is functionally equivalent

to a human expert in a specific problem domain of reasonable complexity. The equivalence in qualified in terms of its capability to:

- Reason over representations of human knowledge
- Solve the problem by heuristic or approximation techniques
- Explain and justify the solution based on known facts

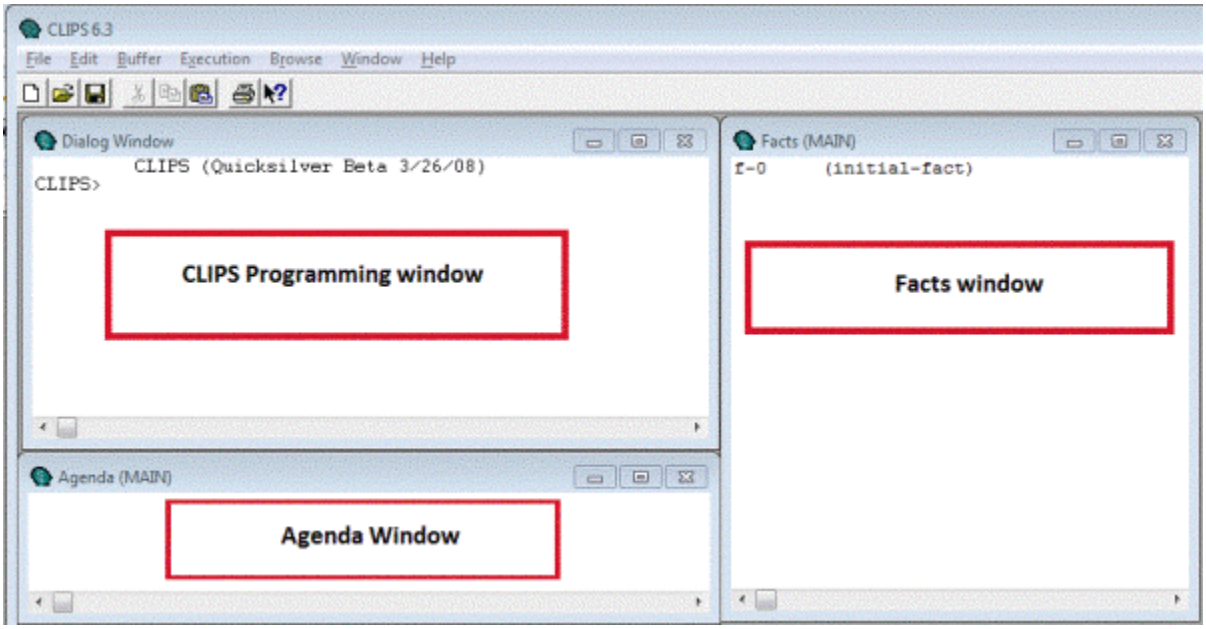
So why do we need to create an Expert System if a human expert can solve the same problem? Well here are a few reasons :

- Low cost of expertise
- Response without cognitive bias or logical fallacies
- 24/7 availability with high reliability
- Clone the expert
- Use in hazardous environments
- Ability to explain the reasoning without attitude.

I will delve into theoretical concepts later, for now let's get started with programming an expert.

Programming an Expert system

Download and install CLIPS. (CLIPS is an open source framework for programming expert systems, it was developed by NASA-Johnson Space Center).
Open CLIPSWin.exe. From the Window menu open the facts and agenda windows.



You are now all set to program an expert system.

Writing "Hello world" in CLIPS

In the CLIPS programming window write the following code (or just copy/paste it) and hit enter.

```
(defrule start-up
=>
(printout t "Hello indecisive homo sapien. The 'wise one' is here to help!" crlf ))
```

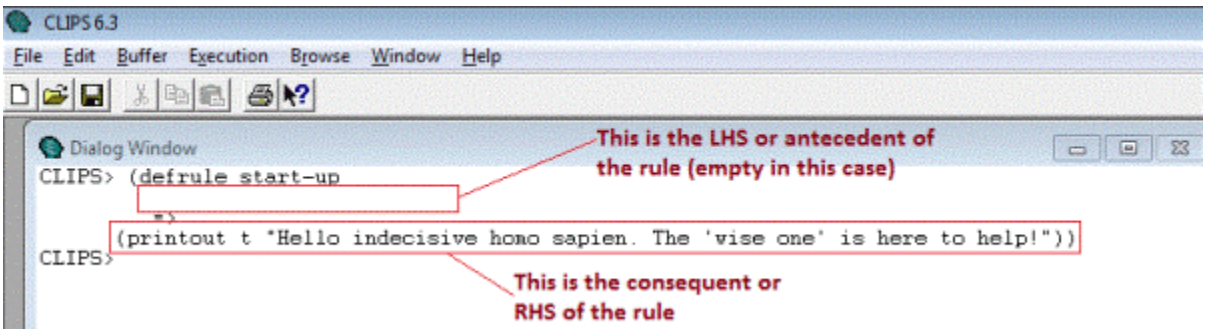
Then type `(run)` and hit enter.

A "*Hello indecisive…"* message will show up on the window. Let us take a closer look at what just happened.

Heuristic knowledge as Rules

A rule captures a set of actions to perform in response to certain stimuli. The stimuli come from the information in the working memory of an expert system. This information is called facts. Coming back to the "Hello world" app. The first thing you did was define a rule using the "*defrule*" construct and named it "*start-up*". This instructs the expert system to create a rule. A rule consists of two parts

1. Left hand side (LHS) or antecedent
2. Right hand side (RHS) or consequent



The antecedents of a rule are a collection of "conditional elements" (CEs). When all CEs are satisfied (evaluate to true) actions in the RHS of the rule are executed. Since our "`start-up`" rule has no conditions it executes (or fires) immediately after you issue the `(run)` command. The "run" command instructs the expert system engine to fire all rules whose LHS or antecedents are true (or have no antecedents).

Rules with antecedents

Now let us modify the rule and add a condition (antecedent or LHS). Say you want to print "Hello indecisive…" only when there exists a fact with a specific name in it. In order to do that

we modify the rule and add an antecedent or LHS to it. The antecedent is a fact represented by symbols "name asheesh goja" . I will explain facts and symbols in more detail later, but for now modify the rule by typing the following text and hit enter

```
(defrule start-up
(name asheesh goja)
=>
(printout t "Hello indecisive homo sapien. The 'wise one' is here to help!" crlf))
```

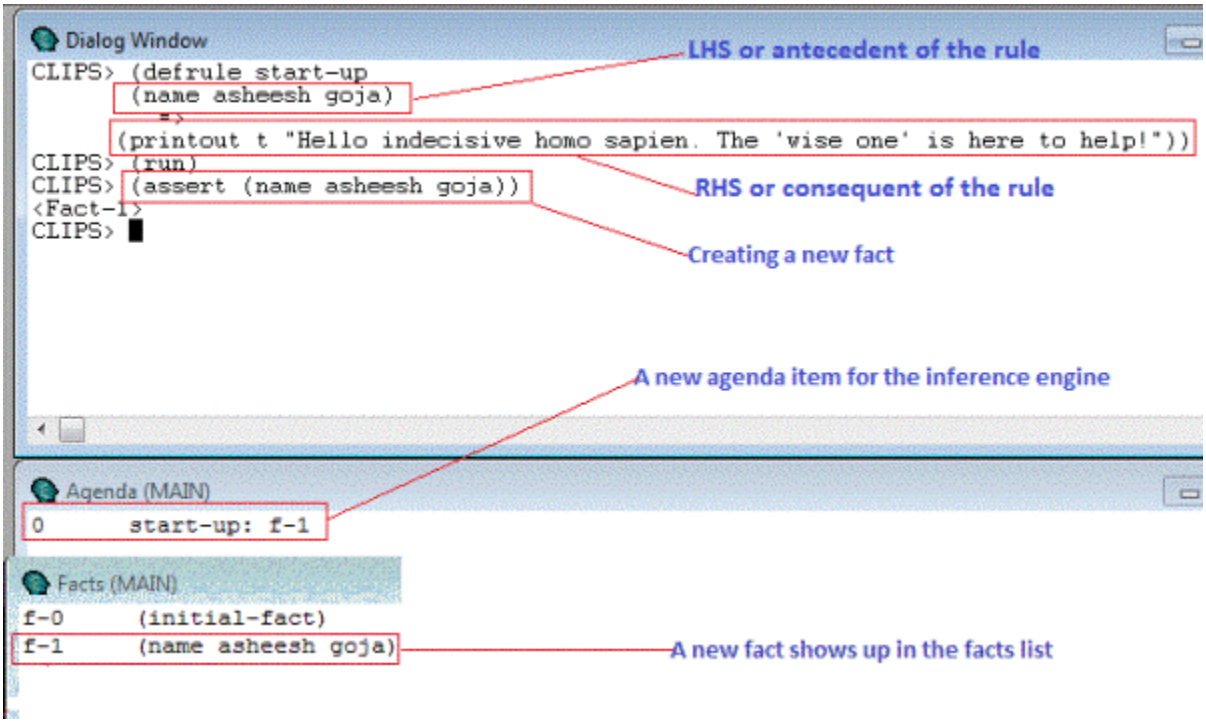
Type (run) and hit enter. This time there is no response from the system. It is because this rule will fire only when a fact "(name asheesh goja)" exists in its working memory or facts list.

To add this fact type the following code and hit enter

```
(assert (name asheesh goja))
```

Notice two things happen after you issue this command

1. An entry shows up in the agenda window: "start-up: f-1"
2. And another one in the facts windows: "f-1 (name asheesh goja)".
   (Note: You will also see a fact "f0 (initial-fact)", this 'a priori' fact is created by the CLIPS engine)



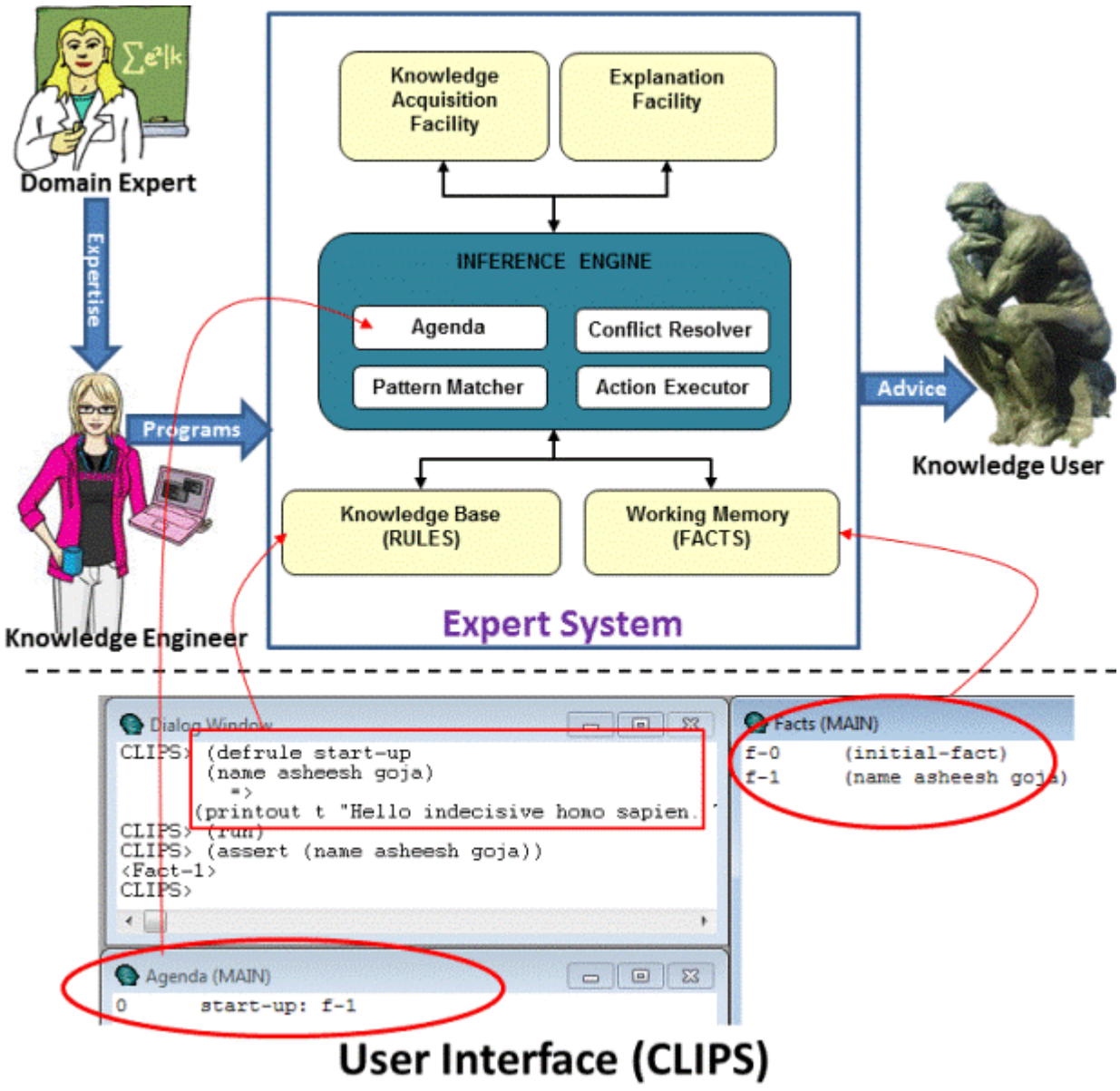
Creating (or "asserting" in experts system parlance) a fact triggers the "start-up" rule, as the rule's antecedents are now true. So when you run the engine it fires the rule. Type `(run)` and hit enter to see the "Hello indecisive…" shows up.

Ok now let us step back to take a peek under the hood.

Anatomy of an expert system

An expert system consists of the following components:

1. **Explanation facility** – Provides the reasoning behind a certain conclusion.
2. **Knowledge acquisition facility** – Provides a means for capturing and storing knowledge elicited from a human expert(s) into a knowledge base.
3. **Knowledge base**– Stores this knowledge in form of rules.
4. **Working memory**–Database of facts used by the rules.
5. **Inference engine**– Decides which rule to fire and in what priority.
6. **Agenda** – A prioritized list of rules whose conditions are satisfied by facts.
7. **Pattern matcher**– Compares rules and facts.



Execution of an expert system

The core facility in an expert system is its **inference engine**. The inference engine follows this cycle

1. **Pattern matcher** uses an algorithm like **Rete** to create a list of rules whose antecedents (or LHS) match (or are satisfied) with the facts in the **working memory**.
2. The **agenda** determines the order in which the rule fire from this list.
3. The **conflict resolver** selects the rule with the highest priority from the agenda.

4. The **action executor** executes the consequent (or RHS) of the selected rule and also removes this rule from the agenda.
5. The **pattern matcher** kicks in again and updates the agenda with rules whose antecedents match and also removes the rules whose don't.

This cycle continues until there are no rules left in the agenda. A particular rule fires only once it's LHS is satisfied. This important feature of expert systems is called **Refraction**. It ensures that the expert system doesn't get caught in trivial loops.

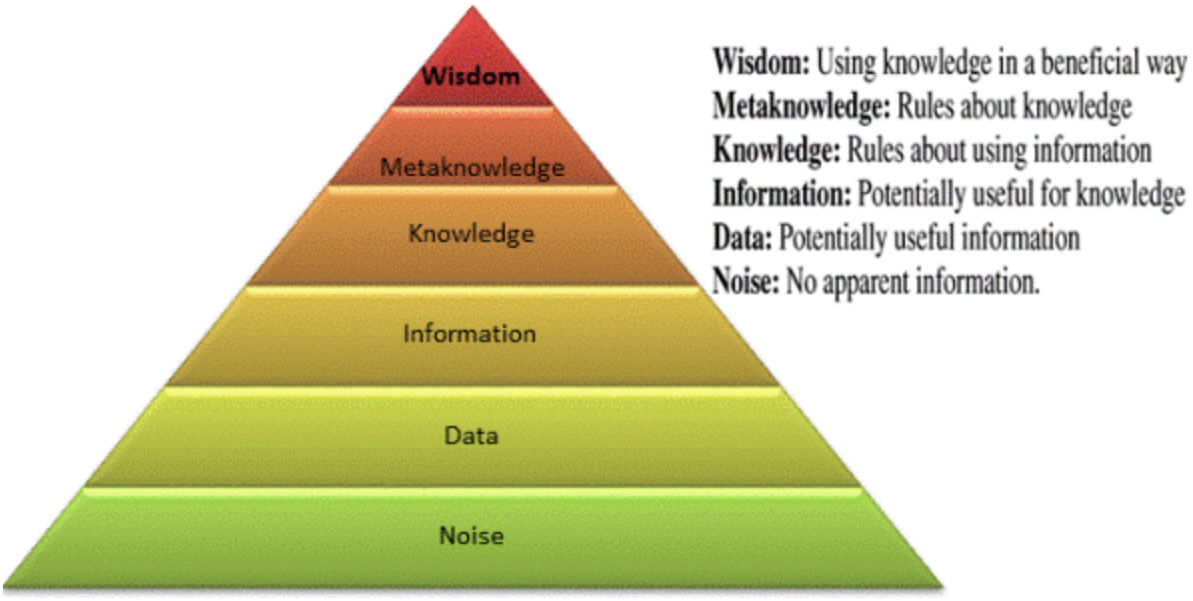
Now that we know how an expert system works; the next thing is to find out how human expertise is converted into a format for expert systems to reason over. This is called Knowledge Engineering.

Knowledge Engineering

Knowledge engineering is the process of eliciting the domain knowledge from a human expert and then structuring, formalizing and integrating it into the knowledge base of an expert system. Domain knowledge is the logic or heuristic used by a human to arrive at a particular solution. Unlike an algorithm where you are guaranteed to get an exact solution in a finite amount of time, expert systems only guarantee the most reasonable solution in a finite time. Depending on the level of a human expert's understanding of the causal connections of a system, a knowledge engineer can program an expert system using heuristic/empirical or structural/behavioral knowledge. The former is called **shallow knowledge** and latter **deep knowledge**. This knowledge is expressed as a collection of rules called the knowledge base. Now let us see how domain knowledge is represented as rules.

Knowledge Representation

Computer programs consist of **data structures** and **algorithms** while expert systems consist of **knowledge** and **inference**. Knowledge is part of hierarchy called the knowledge pyramid.



Representing knowledge so that valid inferences can be made from it is called knowledge representation. Before we represent knowledge we first need to define:

- What is knowledge?
- How is knowledge acquired?
- How do we know what we know?

A study of such questions is called epistemology.

Epistemology

Epistemology is the study of the nature, structure and origins of knowledge. Categories of epistemology are:

- A priori - This is the universally true knowledge.
- A posteriori – This is knowledge acquired from senses and can be invalidated on basis of new knowledge
- Procedural - This is the knowledge of how to do something.
- Declarative – This is the knowledge that something is true or false.
- Tacit – This is unconscious knowledge.

Epistemic categories are represented in expert systems as **productions.**

Productions

Productions consists of two parts: A precondition (an IF) and a post condition (a THEN). If the production's precondition is verified as true based on the current state of the system,

then the production is triggered. You saw one such production called **rules** in the "*Heuristic knowledge as Rules*" section. Additional techniques used to define productions are:

- Semantic nets
- Frames
- Logic

Logic

Logic is the study of making valid inferences. It has two primary branches:

- **Informal Logic** – Here the meaning of statement is used to make valid inferences. Semantics is of the essence.
- **Formal Logic** – Here the form of statement is used to make valid inferences. Syntax is of the essence.

Formal Logic

Expert systems need to make inferences without knowing the meaning of words, hence they use formal logic. The most fundamental formal logic is based on syllogism. Consider the following statements

1. all programmers are smart.
2. asheesh is a programmer.
3. therefore asheesh is smart.

Using the law of syllogism , the 3rd statement was inferred from the hypothesis of the 1st statement and the conclusion of 2nd. This syllogism is represented in an expert system as a rule that takes the form

IF all programmers are smart **AND**
asheesh is as programmer
THEN asheesh is smart

Methods of inference or reasoning

Key to knowledge representation is to separate the actual meaning of words from the reasoning process itself. What may appear as reasoning may be knowledge. Once that is accomplished then **inferences** can be represented as mathematical rules that change one set of symbols to another. Various methods of making inferences are:

1. Deduction – a premise, a conclusion and a rule that the former implies the latter
2. Induction – inference is from the specific case to the general
3. Intuition – no proven theory
4. Heuristics – rules of thumb based on experience

5. Generate and test – trial and error
6. Abduction – reasoning back from a true condition to the premises that may have caused the condition
7. Default – absence of specific knowledge
8. Autoepistemic – self-knowledge
9. Nonmonotonic – previous knowledge
10. Analogy – inferring conclusions based on similarities with other situations

The process of grouping multiple inferences that connects a problem with a solution is called chaining.

Chaining

A chain that is traversed from a problem to its solution is called a forward chain. A chain traversed from a hypothesis back to the facts that support the hypothesis is a backward chain. Consider the following rules chain

- IF A then B
- IF B then C
- IF C then D

If fact A is asserted then the inference engine matches the facts against the antecedents resulting in assertion of intermediate facts B and C. This in-turn leads to conclusion D. This is **forward chaining**.

In **backward chaining** if fact (or hypothesis) D is asserted then the inference engine matches the facts against the consequents. Thus C and B now become sub-goals or intermediate hypotheses that must be satisfied to satisfy the hypotheses D. The evidence as fact A ends the backward chain of sub-goals hence validating D. If no A is found then hypothesis D is unsupported. The chain now becomes

- IF D then C
- IF C then B
- IF B then A

We will use this technique to create a **backward chaining algorithm** in part-2 of this article. For now let see how these concepts apply to our "Hello world" program.

Knowledge representation in CLIPS

In the "Hello world" app the "start-up" rule has an antecedent (or LHS) expressed as a set of three **symbols** "*name asheesh goja*". This rule creates the causal connection between the precondition (LHS) and the conclusion (RHS/antecedent). The CLIPS engine doesn't

know the meaning of these symbols. All it does is to see if a **fact** (a set of **symbols** in the working memory) matches the LHS of the rule. So when you entered symbols "name asheesh goja" in the facts list, it resulted into a complete pattern match (between LHS and fact), hence activation of "start-up" rule. This in-turn triggers the rule's consequent, hence executing the command in its RHS viz. *"(printout t "Hello indecisive homo sapien. The 'wise one' is here to help!"))".* Additional constructs used by a knowledge engineer to represent domain expertise are:

1. Ordered Facts
2. Unordered Facts
3. Rules

Let me explain them in more detail.

Ordered Facts

The fact "(name asheesh goja)" is example of an ordered fact. The first field of an ordered fact specifies a "relation" that applied to the remaining fields in the ordered fact. Ordered facts don't have an associated fact template. The ones that have are called unordered facts.

Unordered facts

These are structured fact representations created using the "**deftemplate**" construct. We can use this construct to create a new fact called "homo-sapien" and add *name* and *age* attributes/slots to it. Type or copy/paste the following code blocks in CLIPS (hit enter after you type them)

```
(clear)

(deftemplate homo-sapien
    (multislot name (type SYMBOL) (default none) )
    (slot age (type INTEGER) (default 0) ))
```

I issued a `(clear)` command to remove any existing facts and rules. You can constraint attributes of an unordered fact with following restrictions:

1. type
2. default
3. cardinality
4. range
5. allowed-[symbols, strings, lexemes, floats, numbers, instance-names, values].

Rules

Rules in CLIPS are created using the "**defrule**" construct. Let us rewrite the "start-up" rule to use the unordered fact created above. Type or copy/paste the following code in CLIPS and press enter

```
(defrule start-up
    (homo-sapien (age 37) (name asheesh goja))
     =>
    (printout t "Hello indecisive homo sapien. The 'wise one' is here to help!"))
```

Notice that in the LHS of this rule the order of the fact's slots doesn't matter. I specified the age before name. This is the reason why such facts are called unordered facts.

Now that you know the fundamental constructs required to create the knowledge base and working memory, let us see how to create a non-trivial program using some more advanced concepts. CLIPS allows you to control the pattern matching (matching facts with LHS) using

1. Literal constraints
2. Variable constraints
3. Conditional Elements (CE)

Literal constraints

The "start-up" rule's LHS is constrained by the literals 37 and "asheesh goja". This means that this rule will fire only when facts match these constraints. To see that add the following facts (one at a time, followed by pressing the enter key)

```
(assert (homo-sapien (name huan goja) (age 38)))

(assert (homo-sapien (name thomas goja) (age 41)))
```

Notice that no activations show up in the agenda window. This means that none of these facts match the LHS of the "start-up" rule; reason being the literal constrain. Now add a fact that matches the constraints

```
(assert (homo-sapien (name asheesh goja) (age 37)))
```

You will now see three facts in the facts window and the following activation in the agenda window:

"0 start-up: f-3".

This means that the rule "start-up" LHS matches the fact "f-3". The fact "f-3" corresponds to "(name (value asheesh goja) (age 37))"

Variable constraints

What if you want to constraint age attribute within a range of values. Say you want the rule to fire the rule only when age of a person is more than 30 but less than 40. Additionally you want to print out the name of the person(s) that fall in this age category. For that you need variable constraints.

Modify the "start-up" rule's LHS and add the following variable constraints. Type this code and press enter

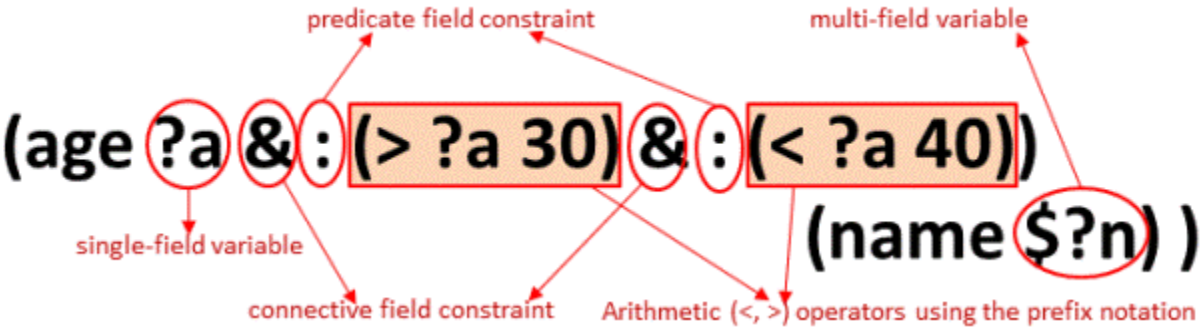
```
(defrule start-up
    (homo-sapien (age ?a&:(> ?a 30)&:(< ?a 40)) (name $?n) )
     =>
    (printout t "Hello indecisive homo sapien named " ?n ". The 'wise one' is here to help!" crlf))
```

Notice two activations show up in the agenda window:

1. "0 start-up: f-3".
2. "0 start-up: f-1".

The syntax may take some time getting used to, but it perfectly complements the functional programming style of CLIPS. CLIPS uses the **postfix** notation as opposed to the **infix** which is common in procedural languages like C++, .net, Java etc. Read this article for more information on such notations.

Now let us take a closer look at LHS to see it individual elements along the variable constraints



This is how you read it in plain English:

*Any homo sapien (**homo-sapien**) whose (**age**) stored in variable named 'a' **(?a**) and (**&**) constrained by (**:**) a value greater that 30 **(> ?a 30**) and (**&**) constrained by (**:**) a value less that 40 **(< ?a 40**) and (**name**) stored in multi-field variable named 'n' **($?n**)*

In additional to the '&' connective constraint CLIPS supports:

1. ~ (not connective)
2. | (or connective)

And conditional element (CE)

We looked at controlling the pattern matching using variable constrains within a fact, but what if we want to express relations between groups of facts. Say we want to trigger "start-up" rule with a constrained "homo -sapien" fact and only if he/she is a millionaire. To do that we add another condition in the LHS of the rule and constraint it with the value stored in variable ?n. Type this rule in CLIPS

```
(defrule start-up
    (and (homo-sapien (age ?a&:(> ?a 30)&:(< ?a 40)) (name $?n) )
    (millionaire $?n))
     =>
    (printout t "Hello indecisive millionaire homo sapien named " ?n ". The 'wise one'
is here to help!" crlf))
```

The rule will fire when following facts exist:
Any homo-sapien whose age is within 30-40 **and** *whose name stored in variable ?n matches any millionaire whose name is the name stored in variable ?n.*

Notice after you write this modified rule the agenda window becomes empty. Now enter a "millionaire" fact

```
(assert (millionaire asheesh goja))
```

The agenda now shows:
"0 start-up: f-3,f-4".

The conditional element used in this rule was "and". Using it is optional as it is assumed by default. So you can rewrite the rule and omit "**and**" CE.

```
(defrule start-up
     (homo-sapien (age ?a&:(> ?a 30)&:(< ?a 40)) (name $?n) )
     (millionaire $?n)
   =>
```

```
     (printout t "Hello indecisive millionaire homo sapien named " ?n ". The 'wise
one' is here to help!" crlf))
```

Or conditional Element

Say you want to trigger this rule with either of the conditions: homo-sapien or millionaire. This is done using the "or" CE

```
(defrule start-up
   (or (homo-sapien (age ?a&:(> ?a 30)&:(< ?a 40)) (name $?n) )
   (millionaire $?n))
=>
   (printout t "Hello indecisive millionaire homo sapien named " ?n ". The 'wise one'
is here to help!" crlf))
```

The rule will now fire only when the following facts exist:
Any homo-sapien whose age is within 30-40 and whose name stored in variable $?n, **OR** *any millionaire whose name is stored in variable $?n.*

Notice three activations show up in the agenda window:

- "0 start-up: f-4".
- "0 start-up: f-3".
- "0 start-up: f-1".

This means that this rule will fire three times.

Enter (run) and to see the following:

- *Hello indecisive millionaire homo sapien named (asheesh goja). The 'wise one' is here to help!*
- *Hello indecisive millionaire homo sapien named (asheesh goja). The 'wise one' is here to help!*
- *Hello indecisive millionaire homo sapien named (huan goja). The 'wise one' is here to help!*

Additional CEs supported in CLIPS are

1. not
2. exists
3. test
4. forall
5. logical.

Limitations of Expert Systems

While expert systems provide consistent answers for repetitive decisions and can process large quantities of information; it is important to know its limitations though. Here are a few:

- Expert systems work best in a narrow domain of reasonable complexity. Generalizing them for a broader domain is usually counterproductive.
- An expert system cannot completely replace human common sense.
- It is not always possible for a human expert to explain how he/she reasons. This can make developing an expert system completely unfeasible.
- Expert systems don't guarantee the quality of its rules. Garbage in = Garbage out.
- Given the ease of adding new rules, a novice knowledge engineer can easy add rules that conflict with existing rules, potentially compromising the accuracy/reliability of the expert system.

Now that we learnt the fundamental concepts of Expert Systems—It is time to use these concepts to develop an expert system based artificial advisor.

Socrates—the artificial advisor

The goal of this expert system is to provides an objective assessment of one's nuptial success, based on the following factors:

1. Income compatibility
2. Age Factor
3. Employment status
4. Marriage penalty tax liability
5. Cohabitating couple economy
6. Health insurance coverage
7. Social security benefits
8. Family dynamics
9. Myers briggs personality type personality compatibility (MBTI)

Domain Expertise

The first and foremost thing to write an expert system is to find a **domain expert**. But for the purpose of this article I will use the following website as the SMEs. (For a serious expert system you will need a human domain expert).

- http://articles.moneycentral.msn.com/CollegeAndFamily/SuddenlySingle/WhenItPaysToStaySingle.aspx?page=2
- http://www.bankrate.com/brm/news/pf/20051017a1.asp
- http://money.usnews.com/money/personal-finance/articles/2008/07/02/marriages-financial-pros-and-cons
- http://marriagecalculator.acf.hhs.gov/marriage/assumptions.php
- http://www.personalitydesk.com/story/compatibility-and-your-myers-briggs-personality-type

This domain expertise needs to be converted into a format that the inference engine can reason over.

Knowledge engineering "Socrates"

As explained earlier the two most frequently employed strategies in expert systems reasoning are: "forward chaining" and "backward chaining". The former is also called data-driven and the latter goal-driven. In backward chaining the inference engine selects a goal and then attempts to find facts to affirm or contradict this goal. In this process new goals

can be established called sub-goals. Since CLIPS doesn't directly support backward chaining, I will use the following backward chaining algorithm.

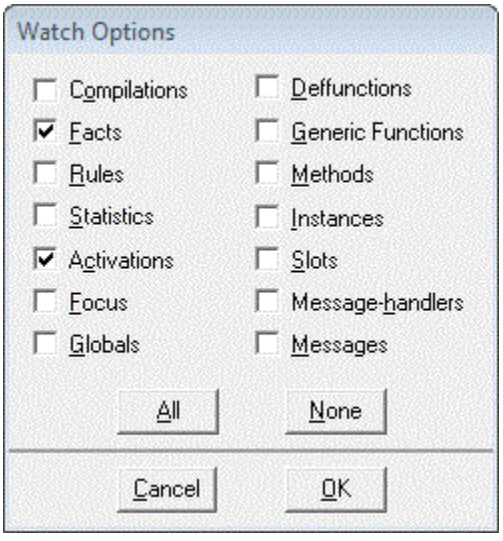
The backward chaining algorithm

1. Express associations between a goal (consequent) and the conditions it depends on (antecedents) as facts, and name this fact **domain-rule**.
2. The consequents of a domain-rule are the goals for be resolved.
3. If a domain-rule has an antecedent then each antecedent becomes a sub-goal.
4. Each sub goal is resolved by affirming or negated it based on the facts called **answer**.
5. If a sub-goal is affirmed then this antecedent is removed from the domain-rule's antecedents.
6. Repeat step 3 and 4 until there are no antecedents left.
7. When a domain-rule has no antecedent then fire the rule to create of a new fact called **conclusion**.
8. The conclusion consists of the goal along with certainty rating called "confidence-factor" .
9. If two goals resolve to same conclusion then combine their certainties (confidence-factors) using the function **((100* (cf1 + cf2))-(cf1 * cf2))/100**

Let us see how this is implemented in CLIPS. But first enable activations and facts tracing by pressing "CTRL W" or typing the commands

```
(watch facts)

(watch activations)
```



And then clear the CLIPS environment by typing **(clear)**

Implementing the algorithm

Algorithm step 1-2 implementation

Define a fact template that captures the link between the goal and its conditions. Type or copy/pastes this code in the CLIPS window
(**Note :** After each type or copy/paste operation in the CLIPS window press enter)

```
(deftemplate domain-rule
    (multislot if (default none))
    (multislot then (default none)))
```

This fact template can now express the following statement:
"if the age difference between two people is more than 30, then the experts system's confidence-factor based on "age-factor" of getting married is 20%" As represented by this 'domain-rule'. Type it in CLIPS

```
(assert
    (domain-rule
        (if age-difference is-more-than 30 )
        (then based-on age-factor the-expert-system-favours-getting-married-with-
certainty 20.0 %)))
```

You will now see a fact "*f-1 (domain-rule (if age-difference...*" in the facts window.

Algorithm step 3, 4, 5, 6 implementation

We now have to affirm or negate the goal "Age-factor with certainty 20 %" with the antecedents "age difference" of more than 30.To capture such fact that can affirm or negate the antecedents we define a fact-template called "answer". Type this in CLIPS

```
(deftemplate answer
    (slot known-factor (default none))
    (slot value (default none)))
```

Next we define a rule that fires when an antecedent of a domain-rule matches an "answer" fact. Type this rule in CLIPS

```
(defrule remove-ask-if-in-domain-rules-with-more-than
    ?r <- (domain-rule (if ?first-ask-if is-more-than ?min $?rest-of-ifs-true))
    (answer (known-factor ?f&:(eq ?f ?first-ask-if)) (value ?a&:(> ?a ?min)) )
 =>
    (if (eq (nth$ 1 ?rest-of-ifs-true) and)
```

```
    then (modify ?r (if (rest$ ?rest-of-ifs-true)))
    else (modify ?r (if ?rest-of-ifs-true))))
```

This is how you read the LHS (antecedent) of this rule

- Assign the "domain-rule" fact to a variable `?r`.
- Assign the first symbol in the "if" slot to a variable `?first-ask-if`.
- The next symbol should match "`is-more-than`"
- Assign the symbol following "`is-more-than`" to a variable `?min`
- Assign the rest of the symbols to a multi-field variable `?rest-of-ifs-true`
- **and**
- The answer fact's "known-factor" slot (assigned to variable ?f) must exactly match the variable `?first-ask-if` and the "value" slot (assigned to variable ?a) must be greater than `?min`.

This is how you read the RHS (consequent) of this rule

- **If** the first symbol in the `?rest-of-ifs-true` is "and" **then**
- Modify the "if" slot of the rule ?r and set its value to rest of the symbols in `?rest-of-ifs-true` after skipping the first symbol.
  (we need this to handle domain-rules with multiple antecedents)
- **Else** if first symbol in the `?rest-of-ifs-true` is not "and" then
- Modify the "if" slot of the rule ?r and set its value to `?rest-of-ifs-true`.

To see this rule triggered let us create a fact that affirms the antecedent (*age-difference > 30*)
We will see later how this fact can be automatically inferred by asking the user a series of questions.

```
(assert
    (answer (known-factor age-difference) (value 31)))
```

Understanding the trace

Notice the following trace

- *==> f-2 (answer (known-factor age-difference) (value 31))*
- *==> Activation 0 remove-ask-if-in-domain-rules-with-more-than: f-1,f-2*

What this means that the fact f-2 in now in working memory and the rule "remove-ask-if-in-domain-rules-with-more-than" is now on the agenda, as its LHS matches facts f-1 and f-2.

Next press "CTRL T" or type (run 1) and the trace now shows

- *<== f-1 (domain-rule (if age-difference is-more-than 30) (then based-on age-factor the-expert-system-favours-getting-married-with- certainty 20.0 %))*
- *==> f-3 (domain-rule (if) (then based-on age-factor the-expert-system-favours-getting-married-with- certainty 20.0 %))*

This means that the domain-rule f-1 has been modified and its antecedent "age-difference is-more-than 30" removed from the "if" slot.

Algorithm step 7,8 implementation

This modified fact should now trigger another rule that resolves the goal by concluding that all facts in the antecedent ("if" slot) are affirmed by the facts in the working memory. Create a fact-template that represents a conclusion

```
(deftemplate conclusion
    (slot name (default none))
    (slot confidence-factor (type FLOAT) (default 0.0)))
```

Then define this rule in CLIPS

```
(defrule fire-domain-rule
    ?r <- (domain-rule (if $?a&:(=(length$ ?a) 0))
    (then based-on ?factor&:(> (str-length ?factor) 0) the-expert-system-favours-
getting-married-with-certainty ?cf % $?rest-of-factors))
  =>
    (if (eq (nth$ 1 ?rest-of-factors) and)
    then (modify ?r (then (rest$ ?rest-of-factors))))
    (assert (conclusion (name ?factor) (confidence-factor ?cf))) )
```

This is how you read the LHS(antecedent) of this rule.

- Assign the domain-rule fact to a variable `?r`.
- Assign all symbols in the "if" slot to a variable ?a and make sure it is empty (by checking its length).
- Assign the symbol in the "then" slot that appears between the "`based-on`" and "`the-expert-system-favours-getting-married-with-certainty`" symbols to a variable ?`factor` and makes sure its length is non-zero.
- Assign the symbols that appears after "`the-expert-system-favours-getting-married-with-certainty`" but before the "`%`" symbol to a variable ?cf.
- Assign the rest of the symbols to a multi-field variable `$?rest-of-factors`

This is how you read the RHS (consequent) of this rule.

- If the first symbol in variable `?rest-of-factors` is "and" then
- Modify the "then" slot of the rule ?r and set its value to rest of the symbols in variable `?rest-of-factors` after skipping the first symbol.
  (we need this to handle domain-rules with multiple consequents)
- Else if first symbol in the `?rest-of-factors` is not "and" then
- Create a new fact in the working memory called "conclusion" and set its "name" slot to the variable ?`factor` and the "confidence-factor" slot to variable ?cf.

Understanding the trace

You now see this trace

- ==> Activation 0 fire-domain-rule: f-3

Press CTRL T to fire this rule. This trace show up

- ==> f-4 (conclusion (name age-factor) (confidence-factor 20.0) (evaluated no))

What it means is that the goal "age-factor" has been resolved and a conclusion asserted with a "confidence factor" of 20.

Algorithm step 9 implementation (combining certainty)

What if two goals resolve to the same conclusion. Say if a person aged more than 40 also resolves to conclusion "age-factor" with "confidence-factor" of 45%. To express it write the following "domain-rule"

```
(assert
    (domain-rule
    (if your-age is-more-than 40 )
    (then based-on age-factor the-expert-system-favours-getting-married-with-certainty
45.0 %)))
```

Then enter this "answer" fact to trigger the "fire-domain-rule"

```
(assert
    (answer (known-factor your-age) (value 47)))
```

Now type (run) or press CTRL R

This will create another "conclusion" facts in working memory with name "age-factor" and value 45.

Now let us write a rule to combine conclusions with same name

```
(defrule combine-confidence-factors
    ?rem1 <- (conclusion (name ?n) (confidence-factor ?f1))
    ?rem2 <- (conclusion (name ?n) (confidence-factor ?f2))
    (test (neq ?rem1 ?rem2))
  =>
    (retract ?rem1)
    (modify ?rem2 (confidence-factor (/ (- (* 100 (+ ?f1 ?f2)) (* ?f1 ?f2)) 100))))
```

This is how you read the LHS(antecedent) of this rule.

- Assign the a conclusion fact to a variable `?rem1`.
- Assign the value of "name" slot to a variable "n and of "confidence-factor" to ?f1
- Assign the another conclusion fact to a variable ?rem2 and make sure its "name" slot matches the variable ?n and assigin its "confidence-factor" to ?f2.
- Make sure ?rem1 and ?rem2 don't refer to the same fact.

This is how you read the RHS (consequent) of this rule.

- Remove fact ?rem1 from working memory
- Modify the "`confidence-factor`" slot of ?rem2 to the value returned by this function((100* (?f1 + ?f2))-(?f1 * ?f2))/100

Understanding the trace

After you type the rule the following trace shows up

- *==> Activation 0 combine-confidence-factors: f-4,f-8*
- *==> Activation 0 combine-confidence-factors: f-8,f-4*

This means that the rule "combine-confidence-factors" LHS matches facts f4 and f-8.

Press CTRL T and the final conclusion shows up:

(conclusion (name age-factor) (confidence-factor 56.0)) based on this function
((100 (20 + 45))-(20 * 45))/100* = **56.0**

The complete algorithm

To see this algorithm in a single file and execute it follow these steps

- Unzip Download Backward_chaining_algorithm.zip .
- Load "*Backward chaining algorithm.bat*" in CLIPS using keys "ALT F B".
- Type (run) or press CTRL R.

Making the system interactive

Now that we have the basic backward chaining algorithm in place, let see how we can makes our system interactive and ask user questions instead of hardcoding the answers. We will also see how additional answers can be inferred from answers to basic questions.

Define the question template

We will first define an **unordered fact** called "question". Type the below (**Note**: Make sure you first load and run the "*Backward chaining algorithm.bat")*

```
(deftemplate question
    (slot factor (default none))
    (slot question-to-ask (default none))
    (slot has-pre-condition (type SYMBOL) (default no)))
```

The "question" fact has following attributes/slots/fields:

- **factor** – A unique human readable identifier for the fact, constrained by not allowing nulls.
- **question-to-ask** – The text of the question, constrained by not null.
- **has-pre-condition** – Indicates if this question should be asked only if another question has been answered first. Defaults to no.

Inferring the "age-difference" answer

To do that we first want to know is the age of the user. We do that by creating/asserting the following fact in CLIPS

```
(assert
    (question
    (factor your-age)
    (question-to-ask "What is your age?") ))
```

Then we also want to know the age of the person the user wishes to marry

```
(assert
    (question
    (factor your-partner-age)
    (question-to-ask "What is the age of the person you wish to marry?") ))
```

Next we create a rule that infers the "age-difference" from these facts

```
(defrule calculate-age-difference
    (answer (known-factor your-age) ( value ?your-age))
    (answer (known-factor your-partner-age) ( value ?your-part-age))
  =>
    (assert (answer (known-factor age-difference) (value (abs (- ?your-age ?your-part-
age)) ))))
```

Now we need to convert the questions to an interactive dialog with the user. For that we define the following rule

```
(defrule ask-question
    ?q <- (question (question-to-ask ?question)
    (factor ?factor)
    (has-pre-condition no))
    (not (answer (known-factor ?factor)))
  =>
    (printout t ?question crlf)
    (assert (answer (known-factor ?factor) (value (read)))))
```

What this rule means that if there is a question in the working memory that is not answered yet and has no pre-condition, then prompt the user with a question. The consequent of this rule (RHS) prompts the user with the "?question" and then creates (asserts) an "answer" fact with its "known-factor" slot set to "?factor" and "value" to the user response (return value of the *'(read)'* I/O function).

Understanding the trace

After typing the rule in CLIPS the following trace shows up

- *==> Activation 0 ask-question: f-11,**

Notice question fact f-10 is doesn't activate this rule as it has a matching answer fact in memory (f-4), meaning it is already answered.
Now fire the rule by typing (run).
You will be prompted with a question, to which you answer 68.
This in-turn fires the "calculate-age-difference" rule creating a new answer fact the working memory.

- *==> f-12 (answer (known-factor your-partner-age) (value 68))*
- *==> Activation 0 calculate-age-difference: f-4,f-12*
- *==> f-13 (answer (known-factor age-difference) (**value 21**))*

Question with pre conditions

Now that we have rules to trigger user prompts, let us now see how question dependencies are implemented.
Say we have a question that should be asked only if you have a certain answer in the working memory (or facts database). Like this one: Find the user's annual income only if his/her work status is 'employed'. In order to that we need to first define a question with its pre-condition slot set to 'yes'. Type this code in CLIPS

```
(assert
    (question
    (factor your-annual-income)
    (question-to-ask "What is your annual income in USD?")
    (has-pre-condition yes)))
```

This will create a new fact "f-14" in the facts database, but won't trigger the "ask-question" rule as its pre-condition slot is set to *yes*, as see in this trace

- *==> f-14 (question (factor your-annual-income) (question-to-ask "What is your annual income in USD?")(has-pre-condition yes))*

The pre-condition is that the question should only be asked if the 'employment status' of the person is 'employed'. To express that we define another fact template called "question-rule".

```
(deftemplate question-rule
    (multislot if (default none))
    (slot then-ask-question (default none)))
```

and use this fact-template to state the question dependency in a natural language like syntax

```
(assert
    (question-rule (if your-work-status is employed) (then-ask-question your-annual-
income)))
```

This will create a new "question-rule" fact "f-15" in the facts database a seen in this trace

- *==> f-15 (question-rule (if your-work-status is employed) (then-ask-question your-annual-income))*

Implementing the question dependency rules

Here, we again use the **backward chaining algorithm.** Instead of using the "domain-rule" we use the "question-rule" and instead of "answer" facts we use "question" facts. Let us see how it is implemented by the defining the following rule

```
(defrule remove-ask-if-in-question-rules
    ?r <- (question-rule (if ?first-ask-if is ?val $?rest-of-ifs-true))
    (answer (value ?val) (known-factor ?f&:(eq ?f ?first-ask-if)))
  =>
    (if (eq (nth$ 1 ?rest-of-ifs-true) and)
    then (modify ?r (if (rest$ ?rest-of-ifs-true)))
    else (modify ?r (if ?rest-of-ifs-true))))
```

When all sub-goals (antecedents) are resolved then the following rule will set the "has-pre-condition" slot of the question fact to "no".

```
(defrule set-pre-condition-when-no-antecedents
    ?r <- (question-rule (if $?a&:(=(length$ ?a) 0)) (then-ask-question ?f))
    ?q <- (question (factor ?f)(has-pre-condition yes) )
    (not (answer (known-factor ?f)))
  =>
    (modify ?q (has-pre-condition no)))
```

To see this rule in action enter the following question fact

```
(assert
    (question
    (factor your-work-status)
    (question-to-ask "What is your work status?") ))
```

Understanding the trace

This will activate the "ask-question" rule as seen in this trace

- *==> f-16 (question (factor your-work-status) (question-to-ask "What is your work status?") (has-pre-condition no))*
- *==> Activation 0 ask-question: f-16,**

Now let us run this program step-by-step. Press "CRTL T" or type "(run 1)".
This executes the code in the antecedent of the ask-question rule, resulting in a question prompt: "What is your work status?" Type *employed* in response and hit enter.
This triggers the following events:

1. A new answer fact is asserted (f-17) that contains the response "employed".

2. The *remove-ask-if-in-question-rules* is activated as its antecedents match facts f-15 and f-17

As seen in this trace

- *=> f-17 (answer (known-factor your-work-status) (value employed))*
- *==> Activation 0 remove-ask-if-in-question-rules: f-15,f-17*

Notice "0 remove-ask-if-in-question-rules: f-15,f-17" is on the top of the agenda now.
Press "CTRL T" again. This triggers the following events

1. The fact (f-15 question-rule) is retracted
2. A new fact (f-18 question-rule) a clone of f-15 is asserted with the "if" attribute set to an empty list.
3. The rule "set-pre-condition-when-no-antecedents" is activated as its LHS matches facts 18 and 14

As see in this trace

- *<== f-15 (question-rule (if your-work-status is employed) (then-ask-question your-annual-income))*
- *==> f-18 (question-rule (if) (then-ask-question your-annual-income))*
- *==> Activation 0 set-pre-condition-when-no-antecedents: f-18,f-14,**

Notice "set-pre-condition-when-no-antecedents: f-18,f-14,*" is on the top of the agenda now.
Press "CTRL T" again. This triggers the following events

1. The fact (f-14 question) is retracted
2. A new fact (f-19 question) a clone of f-14 is asserted with the "has-pre-condition" attribute set to "no".
3. The rule "ask-question" is activated as its LHS matches fact 19 and others.

As seen in this trace

- *<== f-14 (question (factor your-annual-income) (question-to-ask "What is your annual income in USD?") (has-pre-condition yes))*
- *==> f-19 (question (factor your-annual-income) (question-to-ask "What is your annual income in USD?") (has-pre-condition no))*
- *==> Activation 0 ask-question: f-19,**

Notice "0 ask-question: f-19,*" is on the top of the agenda now.
Press "CTRL T" again. This executes the code in the antecedent of the ask-question rule, resulting in a question prompt: "*What is you annual income in USD*?". Type *20000* and hit enter.
This triggers the following events

1. A new answer fact is asserted (f-25) that contains your annual income.

As seen in this trace

- *=> f-20 (answer (known-factor your-annual-income) (value 20000))*

The complete algorithm

To see this algorithm in a single file and execute it follow these steps

- Unzip Download Making_the_system_interactive.zip .
- Load "*Making the system interactive.bat*" in CLIPS using keys "ALT F B".
- Type (run) or press CTRL R.

Making the system interactive with validation

Expressing user input constraints

What if you want to constraint the response of the questions to a predefined set of options or a range. For example you want the response to the "your-work-status" to be limited of the options: student, employed or retired.

Further you want to constrain the "your-annual-income" to a range: 20 to 100K. In order to do that let us modify the question template and add additional multi-slots.
Type (clear) in CLIPS and then this fact-template

```
(deftemplate question
    (slot factor (default none))
    (slot question-to-ask (default none))
    (slot has-pre-condition (type SYMBOL) (default no))
    (multislot choices (default yes no))
    (multislot range (type INTEGER)))
```

The "question" fact now has two additional slots:

- **choices** – List of possible answers , it default to yes and no.
- **range** – Range of answers that require an integer response, constrained by integer type.

This is how you can now express constraints in the question facts. Enter the following facts in clips

```
(assert
    (question
```

```
    (factor your-age)
    (question-to-ask "What is your age?")
    (range 18 120)))

(assert
    (question
    (factor your-work-status)
    (question-to-ask "What is your work status?")
    (choices student employed retired) ))
```

You will now have two facts in the facts database.

Validating user input based on constraints

Now that we captured the input constraints; the next thing to do is to write a few CLIPS function that enforces these constraints.
First we write this function that checks ranges. It accepts the range and user response as its arguments a return 1 if the response is within the range and a 0 otherwise. This is how you write it in CLIPS

```
(deffunction check-range ( ?min ?max ?answer )
    (if (not (numberp ?answer)) then (return 0) )
    (if ( and (>= ?answer ?min) (<= ?answer ?max) )
    then (return 1)
    else (return 0)) )
```

Notice this method also validates a numeric response using the CLIPS predicate function "*numberp*". Try this method out by executing it from the CLIPS window.

```
(check-range 20 40 1)

(check-range 20 40 two)
(check-range 20 40 21)
```

The first and second call will return 0 and the third call 1.

Next we write a function that keeps prompting the user with the same question until the answer is within the constraints. Type or copy/paste this function in CLIPS

```
(deffunction ask
    (?question ?choices ?range)
    (if (eq (length$ ?range) 0)
    then (printout t ?question ?choices ":")
    else (printout t ?question "range-" $?range ":")
```

```
    )
    (bind ?answer (read) )
    (if (eq (length$ ?range) 0)
     then (while (not (member$ ?answer ?choices)) do
        (printout t "Invalid option! Please specify one of these options" ?choices ":"
)
        (bind ?answer (read))
        (if (lexemep ?answer) then (bind ?answer (lowcase ?answer))))
    else (while (eq (check-range (nth$ 1 ?range ) (nth$ 2 ?range ) ?answer) 0 ) do
        (printout t "Invalid input! Please specify a value within the range" $?range
":")
        (bind ?answer (read))
        (if (lexemep ?answer) then (bind ?answer (lowcase ?answer))))
        )
    (printout t crlf)
    ?answer
)
```

This is how the function works

1. It accepts the question and its answer constraints as its arguments.
2. It then checks if the constraint is a "range" or a set of "options" by checking the length of the multi-slot "?range" and the prompts the user with an appropriate question along with the constraints.
3. It the assigns the user response (return value of the (*read*) I/O function) to the "?answer" variable.
4. If the constraint is a set of "options" it then checks to see if the "?answer" matches any one of the elements of the multi-field variable "?choices" using the CLIPS function "`member$`".
5. If it doesn't match that it re-prompts the user and keeps doing it until the user reopens is matches any one of the elements in the "?choices" multi-field.
6. If the constraint is not a set of "options" meaning it is a "range", then it checks to see if the answer is within the range using the method "check-range".
7. It uses the CLIPS multi-field function "$nth" to access the first and second element in the ?range variable and then passes them to the "check-range" method.
8. If the user response "?answer" is not within the range it re-prompts the user and keeps doing it until the user response is within the first and the second element of the multi-field "?range".

And try it out by calling if from CLIPS (try some invalid responses first)

```
(ask "What is your employment status?" (create$ retired employed student) (create$))
```

Then try this

```
(ask "What is your age?" (create$) (create$ 18 120))
```

The "create$" functions appends any number of fields together to create a multi-field value.
Next we modify the "ask-question" rule to use the "ask" function instead of a direct request/response.
(Note: Since we cleared the CLIPS environment, re-define the "answer" fact-template first)

```
(deftemplate answer
    (slot known-factor (default none))
    (slot value (default none)))
```

Then type this function in CLIPS

```
(defrule ask-question
    ?q <- (question (question-to-ask ?question)
    (factor ?factor)
    (range $?range)
    (choices $?choices)
    (has-pre-condition no))
    (not (answer (known-factor ?factor)))
  =>
    (assert (answer (known-factor ?factor)
    (value (ask ?question ?choices ?range)))))
```

What this rule means is that if there is a question in the working memory that is not answered yet and has no pre-condition, then create (assert) an "answer" fact with its "known-factor" slot set to "?factor" and "value" to the return value of the "ask" function.
As a result this rule will now be activated as it has two matching facts, as seen in this trace

- *==> Activation 0 ask-question: f-1,**
- *==> Activation 0 ask-question: f-2,**

Type (*run*) to fire the rule.
Once prompted respond with invalid answers to see the constrain enforcement in action.
Once you enter valid responsed will see following answer facts:

- *f-3 (answer (known-factor your-work-status) (value student))*
- *f-4 (answer (known-factor your-age) (value 44))*

The complete algorithm

To see this algorithm in a single file and execute it follow these steps

- Unzip Download Making_the_system_interactive-with_validation.zip .
- Load "*Making the system interactive-with validation.bat*" in CLIPS using keys "ALT F B".
- Type (run) or press CTRL R.

Making "Socrates" wiser

So far the rules let "socrates" factor in "age". Now we will add rules to include "income-compatibility" and "marriage-penalty-tax-liability".
Load the "*Making the system interactive-with validation.bat*" into CLIPS again (**don't run it yet**).
Then add the following rules in CLIPS

```
(assert
    (domain-rule (if income-difference is-more-than 100000 )
    (then based-on income-compatibility the-expert-system-favours-getting-married-
with-certainty 15.0 %)) )


(assert
    (domain-rule (if income-difference is-more-than 1000 but-less-than 10000 )
    (then based-on income-compatibility the-expert-system-favours-getting-married-
with-certainty 55.0 %
    and based-on marriage-penalty-tax-liability the-expert-system-favours-getting-
married-with-certainty 25.0 %)))
```

You should now see 16 facts in the facts window.

Extending the backward chaining algorithm

Notice the condition in this rule requires us to check a range expressed as "*income-difference is-more-than 1000 but-less-than 10000*". Our "remove-ask-if-in-domain-rules-with-more-than" cannot process this. So we need an additional rule to handle this condition.

```
(defrule remove-ask-if-in-domain-rules-with-more-than-but-less-than
    ?r <- (domain-rule (if ?first-ask-if is-more-than ?min but-less-than ?max $?rest-
of-ifs-true))
    (answer (known-factor ?f&:(eq ?f ?first-ask-if)) (value ?a&:(and (> ?a ?min) (< ?a
?max)(numberp ?a))) )
  =>
    (if (eq (nth$ 1 ?rest-of-ifs-true) and)
    then (modify ?r (if (rest$ ?rest-of-ifs-true)))
```

```
    else (modify ?r (if ?rest-of-ifs-true))))
```

It will lead to a small problem with the **conflict resolver** though— This rule and the "*remove-ask-if-in-domain-rules-with-more-than*" rule can potentially match the same facts.

Adding Salience

To ensure that this rule takes precedence we modify the "remove-ask-if-in-domain-rules-with-more-than" rule and change its salience to -100. Salience allows you to assign priority (or weight) to a rule.

```
(defrule remove-ask-if-in-domain-rules-with-more-than
    (declare (salience -100))
    ?r <- (domain-rule (if ?first-ask-if is-more-than ?min $?rest-of-ifs-true))
    (answer (known-factor ?f&:(eq ?f ?first-ask-if)) (value ?a&:(> ?a ?min)) )
  =>
    (if (eq (nth$ 1 ?rest-of-ifs-true) and)
    then (modify ?r (if (rest$ ?rest-of-ifs-true)))
    else (modify ?r (if ?rest-of-ifs-true))))
```

Adding more questions

Now let us add a "question" fact to get the annual income of the person the user wishes to marry

```
(assert
    (question (factor your-partner-annual-income)
    (question-to-ask "What is your annual income in USD of the person you wish
marry?")
    (range 20000 1000000)
    (has-pre-condition yes)))
```

But we want to ask this question only if he/she is employed, hence this question and a dependency rule

```
(assert
    (question (factor your-partner-work-status)
    (question-to-ask "What is the work status of the person you wish marry ?")
    (choices student employed retired) ))

(assert
    (question-rule
    (if your-partner-work-status is employed)
    (then-ask-question your-partner-annual-income)))
```

Now you will see three items in the agenda

- *==> Activation 0 ask-question: f-18,**
- *==> Activation 0 ask-question: f-14,**
- *==> Activation 0 ask-question: f-11,**

To infer the income difference we will add following rule

```
(defrule calculate-income-difference
    (answer (known-factor your-annual-income) ( value ?your-inc))
    (answer (known-factor your-partner-annual-income) ( value ?your-part-inc))
  =>
    (assert (answer (known-factor income-difference) (value (abs (- ?your-inc ?your-
part-inc)) ))) )
```

Running the wiser "Socrates"

Type (run) or press CRTL R.
Respond to both work status questions as "employed".
Enter your income as 80000 and the person you wish to marry as 82000.
Enter the age of person you wish to marry as 68

To see see final conclusion facts type this query in clips

```
(find-all-facts ((?c conclusion)) TRUE)
```

You will see the conclusions: (<Fact-9> <Fact-31> <Fact-32>). Type (ppfact) for each one of them

```
(ppfact 9)
(ppfact 31)
(ppfact 32)
```

To see the following conclusions:

- *(conclusion (name age-factor) (confidence-factor 56.0))*
- *(conclusion (name income-compatibility) (confidence-factor 55.0))*
- *(conclusion (name marriage-penalty-tax-liability) (confidence-factor 25.0))*

Printing and formatting the results

To display a formatted conclusion write this rule

```
(defrule print-conclusions
    (declare (salience -5000))
    ?c<- (conclusion (confidence-factor ?cf) (name ?n))
 =>
    (printout t "Factor " (upcase ?n) ", confidence rating:" ?cf " %" crlf))
```

Note the following activations

- *==> Activation -5000 print-conclusions: f-9*
- *==> Activation -5000 print-conclusions: f-31*
- *==> Activation -5000 print-conclusions: f-32*

Type (run) to see the formatted conclusions

- *Factor MARRIAGE-PENALTY-TAX-LIABILITY, confidence rating:25.0 %*
- *Factor INCOME-COMPATIBILITY, confidence rating:55.0 %*
- *Factor AGE-FACTOR, confidence rating:56.0 %*

The complete algorithm

To see this algorithm in a single file and execute it follow these steps

- Unzip Download WiserSocrates.zip .
- Load "*WiserSocrates.bat*" in the CLIPS using keys "ALT F B".
- Type (run) or press CTRL R.

Now that we learnt programming an expert system in CLIPS, it is time to see how to integrate it into a C++, C# and Java application.

Converting the knowledge base into a "CLIPS" file

A CLIPS file is simple text file. To create one in CLIPS press CTRL N. Then and save it as *'SocratesKnowledgeBase.clp'*

You can use following editors :

1. Notepad
2. notepad++ (with its language set to LISP)
3. Jess Developer's Environment

Organizing "a priori" questions

We will first group all of our questions using the "**deffacts**" construct. This construct allows a set of ***'a priori'*** or initial knowledge to be specified as a collection of facts. When the CLIPS environment is reset the facts in this construct are added to the facts list.

Now let us add the following sections to "*SocratesKnowledgeBase.clp*" text file.
Create a "question" template first

```
(deftemplate question
    (slot factor (default none))
    (slot question-to-ask (default none))
    (multislot choices (default yes no))
    (multislot range (type INTEGER))
    (slot has-pre-condition (type SYMBOL) (default no)))
```

Then add the following "a priori" question facts

```
(deffacts questions
    (question (factor your-age) (question-to-ask "What is your age?") (range 18 120) )
    (question (factor your-partner-age) (question-to-ask "What is the age of the
person you wish to marry?") (range 18 120) )
    (question (factor your-work-status) (question-to-ask "What is your work status?")
(choices student employed retired) )
    (question (factor your-partner-work-status) (question-to-ask "What is the work
status of the person you wish marry ?") (choices student employed retired) )
    (question (factor your-annual-income) (question-to-ask "What is your annual income
in USD?") (range 20000 1000000) )
    (question (factor your-partner-annual-income) (question-to-ask "What is your
annual income in USD of the person you wish marry?") (range 20000 1000000) ))
```

Next create a "question-rule" template

```
(deftemplate question-rule
    (multislot if (default none))
    (slot then-ask-question (default none)))
```

Then add the following “a priori” question-rules facts.

```
(deffacts question-rules
    (question-rule (if your-work-status is employed) (then-ask-question your-annual-
income))
    (question-rule (if your-partner-work-status is employed) (then-ask-question your-
partner-annual-income)))
```

Organizing “a priori” domain rules

Create a domain-rule template

```
(deftemplate domain-rule
    (multislot if (default none))
    (multislot then (default none)))
```

Then add the following “a priori” domain-rules facts to be used by the backward chaining algorithm

```
(deffacts domain-rules
    (domain-rule (if age-difference is-more-than 30 )
        (then based-on age-factor the-expert-system-favours-getting-married-with-
certainty 20.0 %))

    (domain-rule (if income-difference is-more-than 100000 )
        (then based-on income-compatibility the-expert-system-favours-getting-married-
with-certainty 15.0 %))

    (domain-rule (if income-difference is-more-than 1000 but-less-than 10000 )
    (then based-on income-compatibility the-expert-system-favours-getting-married-
with-certainty 55.0 % and
                based-on marriage-penalty-tax-liability the-expert-system-favours-
getting-married-with-certainty 25.0 %))

    (domain-rule (if your-annual-income is-more-than 100000 and
                    your-partner-annual-income is-more-than 100000)
    (then based-on income-tax the-expert-system-favours-getting-married-with-certainty
60.0 %))

    (domain-rule (if your-annual-income is-less-than 100000 and
                your-partner-annual-income is-less-than 100000)
```

```
    (then based-on income-tax the-expert-system-favours-getting-married-with-certainty
80.0 %)))
```

Creating rules to infer additional facts

Define an answer template first

```
(deftemplate answer
    (slot known-factor (default none))
    (slot value (default none)))
```

Then add rules to infer age and income difference

```
(defrule calculate-age-difference
    (answer (known-factor your-age) ( value ?your-age))
    (answer (known-factor your-partner-age) ( value ?your-part-age))
    =>
    (assert (answer (known-factor age-difference) (value (abs (- ?your-age ?your-part-
age)) )))
)

(defrule calculate-income-difference
    (answer (known-factor your-annual-income) ( value ?your-inc))
    (answer (known-factor your-partner-annual-income) ( value ?your-part-inc))
     =>
    (assert (answer (known-factor income-difference) (value (abs (- ?your-inc ?your-
part-inc)) ))))
```

Next add a fact template for conclusion

```
(deftemplate conclusion
    (slot name (default none))
    (slot confidence-factor (type FLOAT) (default 0.0))
    (slot evaluated (default no)))
```

Creating a rule of mark questions with pre conditions

In the previous article we manually marked a question with a pre-condition. Now let us see how make the expert system do it automatically using this rule

```
(defrule mark-questions-with-pre-conditions
```

```
    ?q <-(question (factor ?f) (has-pre-condition no))
    (question-rule (then-ask-question ?f) (if $?i&:(> (length$ ?i) 0)) )
   =>
    (modify ?q (has-pre-condition yes)) )
```

Adding the rules to accept and validate user input

We now add rules to prompt the user with a question, validate it and then capture the response in an answer facts

```
(deffunction check-range ( ?min ?max ?answer )
    (if (not (numberp ?answer)) then (return 0) )
    (if ( and (>= ?answer ?min) (<= ?answer ?max) )
    then (return 1)
    else (return 0)))

(deffunction ask
    (?question ?choices ?range)
    (if (eq (length$ ?range) 0) then (printout t ?question ?choices ":") else
(printout t ?question "range-" $?range ":"))
    (bind ?answer (read) )
    (if (eq (length$ ?range) 0)
    then (while (not (member$ ?answer ?choices)) do
        (printout t "Invalid option! Please specify one of these options" ?choices ":"
)
        (bind ?answer (read))
        (if (lexemep ?answer) then (bind ?answer (lowcase ?answer))))
    else (while (eq (check-range (nth$ 1 ?range ) (nth$ 2 ?range ) ?answer) 0 ) do
        (printout t "Invalid input! Please specify a value within the range" $?range
":")
        (bind ?answer (read))
        (if (lexemep ?answer) then (bind ?answer (lowcase ?answer)))))
    (printout t crlf)
    ?answer)

(defrule ask-question
    ?q <- (question (question-to-ask ?question)
    (factor ?factor)
    (range $?range)
    (choices $?choices)
    (has-pre-condition no))
    (not (answer (known-factor ?factor)))
    =>
```

```
    (assert (answer (known-factor ?factor)
    (value (ask ?question ?choices ?range)))))
```

Implementing the backward changing algorithm for domain-rule

Now add rules that implements our backward chaining algorithm

```
(defrule remove-ask-if-in-domain-rules-with-more-than
(    declare (salience -100))
    ?r <- (domain-rule (if ?first-ask-if is-more-than ?min $?rest-of-ifs-true))
    (answer (known-factor ?f&:(eq ?f ?first-ask-if)) (value ?a&:(> ?a ?min)) )
   =>
    (if (eq (nth$ 1 ?rest-of-ifs-true) and)
    then (modify ?r (if (rest$ ?rest-of-ifs-true)))
    else (modify ?r (if ?rest-of-ifs-true))))

(defrule remove-ask-if-in-domain-rules-with-more-than-but-less-than
    ?r <- (domain-rule (if ?first-ask-if is-more-than ?min but-less-than ?max $?rest-
of-ifs-true))
    (answer (known-factor ?f&:(eq ?f ?first-ask-if)) (value ?a&:(and (> ?a ?min) (< ?a
?max)(numberp ?a))) )
  =>
    (if (eq (nth$ 1 ?rest-of-ifs-true) and)
    then (modify ?r (if (rest$ ?rest-of-ifs-true)))
    else (modify ?r (if ?rest-of-ifs-true))))

(defrule fire-domain-rule
    ?r <- (domain-rule (if $?a&:(=(length$ ?a) 0))
    (then based-on ?factor&:(> (str-length ?factor) 0) the-expert-system-favours-
getting-married-with-certainty ?cf % $?rest-of-factors))
   =>
    (if (eq (nth$ 1 ?rest-of-factors) and)
    then (modify ?r (then (rest$ ?rest-of-factors))))
    (assert (conclusion (name ?factor) (confidence-factor ?cf))) )
```

Implementing rules to resolve question dependencies

```
(defrule remove-ask-if-in-question-rules
    ?r <- (question-rule (if ?first-ask-if is ?val $?rest-of-ifs-true))
    (answer (value ?val) (known-factor ?f&:(eq ?f ?first-ask-if)))
 =>
    (if (eq (nth$ 1 ?rest-of-ifs-true) and)
    then (modify ?r (if (rest$ ?rest-of-ifs-true)))
```

```
    else (modify ?r (if ?rest-of-ifs-true))))
(defrule set-pre-condition-when-no-antecedents
    ?r <- (question-rule (if $?a&:(=(length$ ?a) 0)) (then-ask-question ?f))
    ?q <- (question (factor ?f) (has-pre-condition yes) )
    (not (answer (known-factor ?f)))
  =>
    (modify ?q (has-pre-condition no)))
```

Combining confidence factors

```
(defrule combine-confidence-factors
    ?rem1 <- (conclusion (name ?n) (confidence-factor ?f1))
    ?rem2 <- (conclusion (name ?n) (confidence-factor ?f2))
    (test (neq ?rem1 ?rem2))
   =>
    (retract ?rem1)
    (modify ?rem2 (confidence-factor (/ (- (* 100 (+ ?f1 ?f2)) (* ?f1 ?f2)) 100))))
```

Formatting and printing the final conclusion

```
(defrule print-conclusions
    (declare (salience -5000))
    ?c<- (conclusion (confidence-factor ?cf) (name ?n))
  =>
    (printout t "Based on [ " (upcase ?n) " ] expert systems confidence favouring
getting married is " ?cf " %" crlf))
```

Save "*SocratesKnowledgeBase.clp".* This file should look like SocratesKnowledgeBase.zip.

Running the CLIPS file

Follow these steps to run this file in CLIPS

1. Disable tracing by pressing CTRL W and then click the 'None' button
2. Clear the CLIPS environment by typing (clear).
3. Clear the CLIPS window by typing (clear-window).
4. Load "*SocratesKnowledgeBase.clp*" by pressing CTRL L.
5. Reset the CLIPS environment (to assert the initial facts) by typing (reset)
6. Run this file by typing (run).

Integrating CLIPS with C++

Install CLIPS source code using this link.

Create a new C++ console application using MS VS 2008 and name it "*Socrates_AsCppConsoleApp*". Add a cpp file to this project and name it "Socrates_AsCppConsoleApp.cpp". Add the following code to your this file. Make sure you change the file paths of "clipscpp.h" and "CLIPSCPP.lib" to the CLIPS source code folder.

```
#include "C:\Program Files\CLIPS\Projects\Source\Integration\clipscpp.h"
#pragma comment(lib, "C:\\Program
Files\\CLIPS\\Projects\\Libraries\\Microsoft\\CLIPSCPP.lib")
int main()
{
    CLIPS::CLIPSCPPEnv theEnv;
    theEnv.Load("..//SocratesKnowledgeBase.clp");
    theEnv.Reset();
    theEnv.Run(-1);
    return 0;
}
```

Sample Code

- Download Socrates-step-by-step-sample-src-code.zip .
- Open "*Article-Part-3-Samples/Socrates_AsCppConsoleApp/Socrates_AsCppConsoleApp.vcproj*".
- Build the solution and then run it.

Integrating CLIPS with C# using CLIPSNet

Unzip CLIPSNet_libs.zip and extract "**CLIPSNet**" libraries. (To download the latest version of CLIPSNet using this link.)

Create a .net console application and name it "*Socrates_AsCSharpApp*". Add a reference to CLIPSNet.dll and make sure you copy the "CLIPSLib.dll" to the bin/debug folder. Then in the program.cs file add the following code

```
using System;
namespace Socrates_AsCSharpApp
{
   class Program
   {
    static void Main(string[] args)
```

```
    {
        CLIPSNet.Environment theEnv = new CLIPSNet.Environment();
        theEnv.Load(@"..\..\..\SocratesKnowledgeBase.clp");
        theEnv.Reset();
        theEnv.Run(-1);
    }
   }
}
```

Sample Code

- Download Socrates-step-by-step-sample-src-code.zip .
- Open the "*Article-Part-3-Samples/Socrates_AsCSharpApp/Socrates_AsCSharpApp.csproj*".
- Build the solution and then run it.

Integrating CLIPS with Java using JESS

Install JESS using this link.

Create a new Java project using Eclipse. Add *jess.jar* to its list of external jars. Create a new class in the project and name it "*Socrates_AsJavaApp*". Add the following code to this class

```
import jess.JessException;
import jess.Rete;
public class Socrates_AsJavaApp{
public static void main(String[] args) {
    try {
    Rete env = new Rete();
    env.batch("..\\SocratesKnowledgeBase.clp");
    env.reset();
    env.run();
    } catch (JessException e) {
    e.printStackTrace();
    }
  }
}
```

Sample Code

- Download Socrates-step-by-step-sample-src-code.zip .
- Import "*Article-Part-3-Samples/Socrates_AsJavaApp*" into an eclipse workspace.

- Build the project and then run it.

Congratulations!!. You made to the last section of this really long series of articles. To reward your patience I will give away the entire "Socrates" knowledge base free and also show you how to route the results to a text file.

The complete Socrates Expert System

Using MS VS 2008 create a new C++ console project and name is "Socrates". Add a header file this project and name it "CustomFileRouter.h".

CLIPS I/O router

CLIPS lets you to write generic I/O functions and the assign them logical names. These routers can then be used in any of the following I/O functions

1. Open
2. Close
3. Printout
4. Read
5. Readline
6. Format
7. Rename

The first step to write a router in C++ is to extend the "CLIPSCPPRouter" class and override its Query, Print and GetContents functions. Type the code below in CustomFileRouter.h

```
using namespace CLIPS;
class CustomFileRouter : public CLIPSCPPRouter
{
    FILE* _resultsFile;
    char* fName;
public:
    CustomFileRouter(char* fileName)
    {
        _resultsFile = NULL;
        fName = fileName;
    }
    int Query(CLIPSCPPEnv *e,char * name)
    {
        if (strcmp(name,fName) == 0)
        {
```

```
            if(_resultsFile == NULL)
            _resultsFile = fopen(fName , "w+");
            return(TRUE);
        }
        return(FALSE);
    }
    int Print(CLIPSCPPEnv *e,char * name,char *p)
    {
        fputs(p ,_resultsFile);
        return(TRUE);
    }
    int Exit(CLIPSCPPEnv *e,int)
    {
        return(TRUE);
    }
    char* GetContents()
    {
        fseek (_resultsFile , 0 , SEEK_END);
        long lSize = ftell (_resultsFile);
        rewind (_resultsFile);
        char *buffer = (char*) malloc (sizeof(char)*lSize);
        ZeroMemory(buffer,lSize);
        fread (buffer,1,lSize,_resultsFile);
        fclose(_resultsFile);
        _resultsFile = NULL;
        return buffer;
    }
};
```

Adding the router to CLIPS runtime

Add a new file to the project and name it *Socrates.cpp.* In the main function add the following code

```
CLIPSCPPEnv theEnv;
CustomFileRouter* results_fileRouter = new CustomFileRouter("results_file");
theEnv.AddRouter("results_file",10,results_fileRouter);
CustomFileRouter* questions_fileRouter = new CustomFileRouter("questions_file");
theEnv.AddRouter("questions_file",10,questions_fileRouter);
```

This will add two logical routers to CLIPS runtime namely: “results_file”, “questions_file”. You can use these routers within CLIPS in any I/O function. For example this command will print to the “results_file” instead of console.

```
(printout results_file "Hello custom router !!" crlf)
```

Embedding the rules file as a resource

To package “socrates” as a single app with the rules file embedded as a resource follow these steps.

Add the CLIPS rules files to your project.
Then in resources.h file add the following code

```
#define    CLIPS_RULE_FILE 201
#define    IDR_CLP_FILE 101
```

and in the .rc file add this

```
IDR_CLP_FILE     CLIPS_RULE_FILE      "SocratesExpertSystemRules.clp"
```

And this is how you can load the resource at run time

```
HMODULE handle = ::GetModuleHandle(NULL);
HRSRC rc = ::FindResource(handle, MAKEINTRESOURCE(IDR_CLP_FILE),
MAKEINTRESOURCE(CLIPS_RULE_FILE));
HGLOBAL rcData = ::LoadResource(handle, rc);
size = ::SizeofResource(handle, rc);
data = static_cast<const char*>(::LockResource(rcData));
```

Sample Code

- Download Socrates-complete-app-scr-code.zip and unzip it.
- Build "Socrates.sln"
- Run the app.

Conclusion

Artificial advisors are expert systems which are a type of knowledge based systems . For such systems to reason well they must know a lot about the domain in which they operate. So for ‘Socrates’ to reason accurately additional domain expertise needs to be added than what is the sample code.

It wasn't possible for me to cover all aspects of experts systems in these articles, so here is a list of websites that you can use as a reference.

- http://www.aaimedicine.org/journal-of-insurance-medicine/jim/1989/021-04-0233.pdf
- http://clipsrules.sourceforge.net/documentation/v630/bpg.htm
- http://iweb.tntech.edu/bhuguenard/ds6530/ClipsTutorial/tableOfContents.htm
- http://www.slideshare.net/mksaad/introduction-to-clips-expert-system-presentation
- https://msbfile03.usc.edu/digitalmeasures/doleary/intellcont/Brown-Oleary-es_tutor-1.htm
- http://www.wtec.org/loyola/kb/c1_s1.htm
- http://ai-depot.com/Tutorial/RuleBased-Methods.html
- http://www.javaworld.com/javaworld/jw-07-2006/jw-0717-iw-brms.html?page=2

Let the adventure in artificial advising begin!!