Handed Out: Thur March 16, 2006
Due: Thur March 23, 2006

JOSHUA and Rule-based Systems Exercises

I. Introduction

This is an exercise designed to do two things: it will allow you to explore the online behavior of a rule-based system in many ways and it will get you familiar with Joshua, a tool that many people find useful for building their term project.

The Knowledge Base (The Investment Rules Again)
Following on with the paper exercises you did, in this exercise we'll use the same rule set to decide among six of the most common categories of mutual funds. The system assumes that you have $2000 to invest and can help select among the following possibilities:

- a money-market fund
- an income fund (e.g., bonds)
- an aggressive growth fund
- a mixed growth and income fund (abbreviated "G&I")
- a conservative growth fund
- a tax-free fund
- none (i.e., don't invest)

For ease of reference, the complete set of rules in English is found at the end of this handout. As before, we have used some formatting conventions to help make their structure clear. Consider rule 12, for instance:

12] *if* Investment Goal = RETIREMENT and
Number Of Years To Retirement < 10
then Category Of Fund = CONSERVATIVE GROWTH .8

Each rule is expressed in terms of an *if* part (the premise) and a *then* part (the conclusion). In rule 12 there are two clauses in the premise and (as in all of our rules) one in the conclusion.

Each clause in a rule is expressed in terms of an attribute, object, and its value. As a formatting convention, attributes are written as phrases with their first letters capitalized (e.g., Investment Goal, Number Of Years To Retirement, Category Of Fund); values are written in all capitals (RETIREMENT, 10, CONSERVATIVE GROWTH). The first clause thus asks whether the attribute Invest Goal has the value RETIREMENT, or, in smoother English, "the goal for this investment is to fund your retirement." For ease of reading the object is often left implicit, e.g., in this case *investment goal* of *user* is *retirement*.

Putting the whole rule in somewhat better English, it says:
if the goal for this investment is to fund your retirement, and
the number of years until you retire is less than 10,
then the category of fund to select is the conservative growth funds.

In Joshua the rule looks somewhat different; we'll come to that below. Note in particular that the rule numbers in the handout at the end are for convenience. Joshua names rules rather than numbers them.

II. Getting Started

You will be using the Allegro Common Lisp interpreter on Athena and the Joshua knowledge base found on the course web page. We'll start by familiarizing you with Allegro Common Lisp on Athena and Joshua. (While it is possible to install all this on a Windows PC, we strongly recommend against doing that. You can work from a Windows PC by ssh'ing to `athena-x.dialup.mit.edu` (via SecureCRT, make sure you configure the connection to Forward X11 packets,), while running XWin-32. Both SecureCRT and XWin-32 are available for installation from http://web.mit.edu/software/win.html)

1. Log onto an Athena machine (athena-x.dialup.mit.edu works if you're not in an Athena cluster).

2. `athena% add acl_v6.2-alisp 6.871`
   This will attach the Allegro Common Lisp locker for the version we will be using, and the course locker.

3. Download from the course web page the knowledge base file `invest-josh-kb.custom.lisp`. Put this in your local directory. This will make you a local copy of the example knowledge base.

4. `athena% chmod 700 ~/invest-josh-kb.custom.lisp`
   This will just make sure that the file is readable.

5. `athena% joshua8 &`
   Depending on server load, this may take awhile. An XEmacs will start and it will start up a lisp inferior with all the Joshua code in it. Eventually, you'll see a CL-USER(1): prompt in the XEmacs window (the one with the *common-lisp* label).

6. `CL-USER(1): (clim-env:start-clim-environment)`
   A small window called Navigator will pop up.

7. In the Navigator window, select Lisp Listener from the Tools menu.

   A Lisp Listener window will be raised, and you will see a ⇒ prompt. (Yes, windows all over the place.)

8. `⇒:joshua syntax yes`
   This command is necessary to enable the square bracket predication syntax. If you ever in the future forget to do this before compiling your project code, you will get an error message concerning a comma not being inside a backquote.

9. `⇒:edit file ~/invest-josh-kb.custom.lisp`
   This will open the code in a buffer in your XEmacs window.

10. Select `buffer` from the `Compile other` submenu of the `ACLFile` menu in XEmacs

11. Watch the status bar at the bottom of XEmacs for the message Compiling... done.. At this point, return to your Lisp Listener window.

12. Now you can try asking what category of fund someone should invest in. Just type the following command:

    ⇒ `(ask [category-of-fund ben-bitdiddle ?x] #'print-answer-with-certainty)`

You are of course free to inquire about people other than Ben Bitdiddle, our canonical test subject.
See notes at the end of the handout for how to handle errors and the debugger.

III. Using Joshua: Predicates and Predications

Every fact in Joshua's database is called a predication. A predication a list enclosed by square brackets. For example:

```
[FATHER-OF JOHN MARY]
```

might mean that JOHN is the father of MARY. We define predicates (what we have called "attributes" in class) using the define-predicate macro. To define a simple predicate, all we have to do is this (as you'll see below, the prompt character for Joshua is a right-arrow):

```
⇒ (define-predicate father-of (father child))
```

The first argument to the macro is the name of the predicate, and the second is an argument list. There is an optional third argument to the macro, which is a list of types for the predicate to inherit from. This is an advanced feature you don't have to know about right now, but if you are interested, you can see by examining the code that accompanies this problem set how predicates that have certainty factors are defined, inheriting from a base class that provides certainty factor support.

To undefine a predicate, simply execute this command:

```
⇒ (undefine-predicate 'father-of)
```

IV. Using Joshua: TELL-ing things

The first and most simple thing you can do with Joshua is to tell it facts. Take, for example the following code:

```
⇒(tell [HAS-HEALTH-INSURANCE BEN-BITDIDDLE YES])
```

We can also tell things that are justified by certainty factors, like so:

```
⇒(tell [HAS-LIFE-INSURANCE BEN-BITDIDDLE YES] :justification '((user-input 0.8)))
```

Note that we can replace the 0.8 with any certainty factor we choose, and that the symbol USER-INPUT is just a notation to let us to know where the justification comes from; you can write anything there.

UNTELL-ing things is also possible, for example:

```
⇒ (untell [HAS-LIFE-INSURANCE BEN-BITDIDDLE YES])
```

> (Some people have reported that UNTELL can get stuck in an infinite loop; we're trying to track down that bug.)

V. Using Joshua: Seeing what it knows

```
⇒:show joshua database
```

True things

```
[HAS-LIFE-INSURANCE BEN-BITDIDDLE YES]
```

```
[HAS-HEALTH-INSURANCE BEN-BITDIDDLE YES]
```

False things

"None"

So we see that Joshua knows three facts right now. In the Lisp Listener, you can middle click on any one of them to get an explanation, along the lines of the HOW command in MYCIN, of how Joshua inferred that fact. You can also access this explanation by issuing the following command:

```
⇒ :explain predication [HAS-HEALTH-INSURANCE BEN-BITDIDDLE YES]
```

You'll get output that looks like this:

`[HAS-HEALTH-INSURANCE BEN-BITDIDDLE YES]` is true It is a `PREMISE`

This is a rather simple explanation, because we TELL-ed (told) this fact to Joshua at top level in the Lisp Listener. Explanations of the results of the investment advisor code will be more interesting.

VI. Using Joshua: ASK-ing things

The most simple form of ASK is illustrated below:

```
⇒ (ask [HAS-HEALTH-INSURANCE ?x ?y] #'print-query)
```

This command simply runs through the database and executes the function print-query on every predication it finds that matches the pattern. Note that ?x and ?y are logic variables that match anything and that are not necessarily equal. The output of the command in this example is:

```
[HAS-HEALTH-INSURANCE BEN-BITDIDDLE YES]
```

If I were to have told Joshua additionally that ALYSSA-P-HACKER doesn't have health insurance, we'd have seen the following output:

```
[HAS-HEALTH-INSURANCE BEN-BITDIDDLE YES]
[HAS-HEALTH-INSURANCE ALYSSA-P-HACKER NO]
```

Interestingly enough, we can have ASK call any function that we want on database predications that match the pattern. What if we were to issue the following command:

```
⇒ (ask [HAS-HEALTH-INSURANCE ?who ?value]
#'(lambda (backward-support)
(if (eq ?value 'yes) (format t " ~A has health insurance." ?who)
(format t " ~A does not have health insurance." ?who))))
```

We get the following response from the system:

`ALYSSA-P-HACKER` does not have health insurance.
`BEN-BITDIDDLE` has health insurance.

One more fun thing to do is to try this:

```
⇒ (ask [HAS-HEALTH-INSURANCE ?x ?y]#'print-answer-with-certainty)
```

Now we can see the certainty factors:

```
[HAS-HEALTH-INSURANCE ALYSSA-P-HACKER NO] 1
```

```
[HAS-HEALTH-INSURANCE BEN-BITDIDDLE YES] 0.8
```

VII. Using Joshua: Defining rules

Look at the following rule definition from the example code:

```
(defrule full-insurance-coverage (:backward :certainty 1.0 :importance 97)
  if [and [HAS-LIFE-INSURANCE ? who YES]
          [HAS-HEALTH-INSURANCE ? who YES]]
  then [ADEQUACY-OF-INSURANCE-COVERAGE ?who ADEQUATE])
```

defrule is a defining form that makes rules. Its first argument is the name of the rule and its second is a list of keywords. Possible keywords include :backward, for backward chaining rules, and :forward for forward chaining rules. You can't use both at the same time. The :certainty keyword specifies that the next element of the list is the rule's certainty factor. The :importance keyword specifies that the next element of the list is the rule's importance, which is used to artificially control which rules are tried first. A higher importance value means that the rule is tried before rules that have lower importance values. In general you can ignore the importance keyword and number; it's simply evidence that Joshua has more mechanism than needed for this problem set.

The remainder of the form is the body of the rule. if and then are fixed tokens that precede the if and then rule clauses, respectively.

Logic variables are bound for the duration of the rule, so ?who in the if clause must be the same as ?who in the then clause.

This is a backward chaining rule and so it triggers when we ASK things of Joshua. If we were to:

```
⇒ (ask [ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE ADEQUATE
   #'print-answer-with-certainty)
```

Since we know `[HAS-LIFE-INSURANCE BEN-BITDIDDLE YES]` and `[HAS-HEALTH-INSURANCE BEN-BITDIDDLE YES]`, we can conclude the following:

```
[ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE ADEQUATE] 0.8
```

Note that there would be no output if we could not conclude the fact we are asking for from what is in the database. Why is the certainty factor 0.8? Let's see what Joshua knows:

```
⇒:show joshua database
```

Joshua now knows 3 facts:

True things

```
[ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE ADEQUATE]
[HAS-LIFE-INSURANCE BEN-BITDIDDLE YES]
[HAS-HEALTH-INSURANCE BEN-BITDIDDLE YES]
```

False things

"None"

Let's have Joshua explain how it concluded that Ben's coverage was adequate:

```
⇒ :explain predication [ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE ADEQUATE]
```

Recall that middle clicking on [ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE ADEQUATE] in the Lisp Listener will yield the same output:

```
[ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE ADEQUATE] is true
  It was derived from rule FULL-INSURANCE-COVERAGE
  [HAS-HEALTH-INSURANCE BEN-BITDIDDLE YES] is true
    It is an USER-INPUT
  [HAS-LIFE-INSURANCE BEN-BITDIDDLE YES] is true
    It is a PREMISE
```

The certainty factor of 0.8 arises because 0.8 was the minimum certainty factor in the AND clause in the rule's premise. The certainty factor of the rule itself was 1.0, so the conclusion's certainty is 0.8.

VIII. Seeing what Joshua is doing

You can enable tracing in Joshua by doing the following command:

```
⇒ :enable joshua tracing all
```

Watch what happens when we ask the same question as before. Assume we have UNTELL-ed the conclusion; if we hadn't, the rule would not have been triggered at all.

```
   ⇒ (ask [ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE ADEQUATE]
      'print-answer-with-certainty)

> Asking predication [ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE ADEQUATE]
    > Trying backward rule HEALTH-INSURANCE-COVERAGE (Goal... )
      [ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE VALUE]
      > Asking predication [HAS-HEALTH-INSURANCE BEN-BITDIDDLE NO]
    > Exiting backward rule HEALTH-INSURANCE-COVERAGE
    > Trying backward rule LIFE-INSURANCE-COVERAGE (Goal... )
      [ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE VALUE]
      > Asking predication [HAS-LIFE-INSURANCE BEN-BITDIDDLE NO]
    > Exiting backward rule LIFE-INSURANCE-COVERAGE
    > Trying backward rule FULL-INSURANCE-COVERAGE (Goal... )
      [ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE VALUE]
      > Asking predication [HAS-LIFE-INSURANCE BEN-BITDIDDLE YES]
      > Succeed in Asking Predication [HAS-LIFE-INSURANCE BEN-BITDIDDLE YES]
        > Asking predication [HAS-HEALTH-INSURANCE BENDITDIDDLE YES]
          > Succeed in Asking Predication [HAS-HEALTH-INSURANCE BEN-BITDIDDLE YES]
        > Succeeding backward rule FULL-INSURANCE-COVERAGE
            > Justifying: [ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE ADEQUATE]
              <--Rule: FULL-INSURANCE-COVERAGE
        > Looking for more backward rule matches FULL-INSURANCE-COVERAGE
            (Goal... )[ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE ADEQUATE]
        > Exiting backward rule FULL-INSURANCE-COVERAGE
        > Succeed in Asking Predication
        [ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE ADEQUATE]

[ADEQUACY-OF-INSURANCE-COVERAGE BEN-BITDIDDLE ADEQUATE] 1.0
```

To turn off this feature, use:

```
⇒ :disable joshua tracing all
```

IX. Things to note while running the example code

In response to any question, you are allowed to type the following characters. Typing Control+Shift+R prints what rule is running. Typing Control+Shift+W is like asking WHY in MYCIN; it explains what the system is trying to accomplish by asking the current question and running the current rule. Typing Control+Shift+H prints a help message that reminds you of what these keys do.

Each of these key combinations will also print out a message containing the legal answers (or a description like ”a number”). Any possible answer it prints out is mouse sensitive, so try just clicking on the answer you want. The question prompt also provides completions, so try just typing enough characters for your response so be unambiguous, and then hitting the space bar.

Try executing the following command to see a graph of rule calling relationships:

```
⇒ :graph rules category-of-fund
```

If you want to write this graph to a file, try the following:

```
⇒ :graph rules category-of-fund :to file foo.ps
```

X. Using Joshua: Further Reading

For more information about Joshua, see the *Joshua Users Guide* and the *Joshua Reference Manual*, both of which are posted on the course web page.

XI. Exercise 0: Make sure you can run Joshua

The first part of the assignment is to make sure you can run Joshua using the instructions we have given you. Do it today! See the course staff immediately if there is any problem whatsoever; we will do our best to assist you. Please do not wait to try Joshua on Athena. As the due date for this assignment draws near, our ability to help you will be lessened.

XII. Exercise 1: Run the following examples using the knowledge base you loaded above.

Try the following example. The user, `CLYDE`:

is 42 years old.
has health and life insurance.
has a current savings balance of $20,000.
earns a monthly salary of $3,000.
is not covered by a pension plan, but does have an IRA.
has one child, age 12, who does not have a scholarship or a trust fund,
    is not eligible for school loans, and would like to attend a school with
    expensive tuition.
does not currently own a house, but would like to.

Recall that you begin the consultation by executing the following command at the top level of the Lisp Listener:

```
⇒ (ask [CATEGORY-OF-FUND CLYDE ?X] #'print-answer-with-certainty)
```

Prepare a transcript of the dialog, and list the answer(s). For each answer, use Joshua’s Explain Predication feature to find out HOW the answer was derived. **Save this transcript and turn it in**. The easiest way to do

this is with the command `:Copy Output History <filename>` which will copy everything that has appeared in the buffer so far to <filename>. You need only do this once, at the end of the problem set, then edit the file so it contains only the material you are to turn in. If you want to be cautious, however, do it periodically with different file names so you can be sure not to lose any work.

Also try the following example. The user, DUDLEY:
is 70 years old.
does not have health insurance.
does have life insurance.
has a current savings balance of $20,000.
earns a monthly salary of $3,000.
is covered by a pension plan.
has no children.
currently owns a home.

For DUDLEY as well, prepare a transcript of the dialog, and list the answer(s). For each answer, use Joshua's Explain Predication feature to find out HOW the answer was derived. **Print these things out and turn them in.**

XIII. Exercise 2: Practice in Knowledge Engineering

Try the following example. The user, ETHYL:

has health and life insurance.
has a current savings balance of $20,000.
earns a monthly salary of $3,000.
is covered by a pension plan.
has one child, age 23.
is age 68.
owns a home.

If you examine the dialog carefully, you'll see that the system asks about College Tuition. But with the only (and hence youngest) child aged 23, rule 46 would indicate that there are no children headed for college. So why does it ask about tuition?

See if you can determine why. Use Joshua's tracing mechanism to help you determine how the system is reasoning at that point. Try drawing a diagram that shows the tree structure produced by the backward chaining; this may help you see the logic being used and see the problem.

Then see if you can fix the problem. One good approach is to modify the rule that causes the problem. Another is to try introducing a new rule. Specify a solution along both routes. Comment on both solutions. Is one better than the another, in the sense of following the "spirit" of knowledge engineering and rule-writing?

You may want to try implementing your solutions by augmenting the program's knowledge base. You can test whether they work and test their sensitivity and robustness to various inputs. **Turn in your explanation and answers to the questions, and your solution to the problem we are seeing. If you drew a diagram or wrote some Joshua code, show that as well.**

XV. Exercise 3: Looking at Rules

Take a look at the rule `goal-invest-spare-cash`. What strikes you about it? How does the rule compare to the other rules in the knowledge base? Why is this rule a good or bad example of knowledge engineering? Support your answer with information from the lectures and the readings. What is the rule trying to conclude? Can you think of a better way to do it? Turn in your answers to the questions. Please be concise.

What To When Something Breaks

For example, suppose you find the system responding to you like this:

```
Error: Attempt to take the value of the unbound variable '?X'.
Use STORE-VALUE, USE-VALUE, ABORT, or INVOKE-RESTART to resume or abort
execution:
                 Debugger top level
                 Exit Debugger
  TRY-AGAIN:     Try evaluating ?X again.
  STORE-VALUE:   Set the symbol-value of ?X and use its value.
  USE-VALUE:     Use a value without setting ?X.
  ABORT:         Return to Lisp Listener command level 0
                 Restart CLIM Lisp listener
                 Lisp Listener top level
                 Exit Lisp Listener
  ABORT:         Abort entirely from this process.
In process #<PROCESS LISP-LISTENER[8]>.
1➔
```

When you find the Lisp listener window reacting like this, use your mouse to select the first ABORT (the one that says Return to Lisp Listener). This will put you back to the top level of the listener and you can start again. Keep in mind that Joshua keeps its database, so that questions about an individual answered in the run interrupted above will still be there. As a result the system does not re-ask those questions; it continues from whatever the next question is.

Some times you get thrown back into the Xernacs window, with a list of choices that looks something like:

```
Error: Attempt to take the value of the unbound variable '?X'.
  [condition type: UNBOUND-VARIABLE]

Restart actions (select using :continue):
  0: Try evaluating ?X again.
  1: Set the symbol-value of ?X and use its value.
  2: Use a value without setting ?X.
  3: Return to Debugger command level
  4: Restart this debugger
  5: Debugger top level
  6: Exit Debugger
  7: Return to Lisp Listener command level 0
  8: Restart CLIM Lisp listener
  9: Lisp Listener top level
 10: Exit Lisp Listener
 11: Abort entirely from this process.

[changing package from "COMMON-LISP-USER" to "JOSHUA-USER"]
[Current process: LISP-LISTENER]
```

When this happens you reply using :continue (that is, colon-continue) and one of the numeric choices. Usually the correct choice is the one that says Restart CLIM Lisp listener. In the example above, it's #8 so you would type `:continue 8`. Then you can go back to the Lisp listener window and continue working.

THE KNOWLEDGE BASE

RULES ABOUT: adequacy of Basic Insurance Coverage

1] *if* Have Health Insurance = NO
then Basic Insurance Coverage = INADEQUATE 1.0

2] *if* Have Life Insurance = NO and
Should Have Life Insurance = YES
then Basic Insurance Coverage = INADEQUATE 1.0

3] *if* Have Health Insurance = YES and
Have Life Insurance = YES
then Basic Insurance Coverage = ADEQUATE 1.0

RULES ABOUT: whether you Should Have Life Insurance

4] *if* Married = YES or
Have Children = YES
then Should Have Life Insurance = YES 1.0

RULES ABOUT: which Category Of Fund to choose

10] *if* Basic Insurance Coverage = INADEQUATE
then Category Of Fund = NONE 1.0

11] *if* Current Savings < 6 * Monthly Salary
then Category Of Fund = MONEY MARKET 1.0

12] *if* Investment Goal = RETIREMENT and
Number Of Years To Retirement < 10
then Category Of Fund = CONSERVATIVE GROWTH .8

13] *if* Investment Goal = RETIREMENT and
Number Of Years To Retirement > 10 and
Number Of Years To Retirement < 20
then Category Of Fund = G&I .8

14] *if* Investment Goal = RETIREMENT and
Number Of Years To Retirement > 20
then Category Of Fund = AGGRESSIVE .8

15] *if* Investment Goal = CHILDREN'S EDUCATION and
Age Of Oldest Child < 7
then Category Of Fund = G&I .8

16] *if* Investment Goal = CHILDREN'S EDUCATION and
Age Of Oldest Child > 7
then Category Of Fund = CONSERVATIVE GROWTH .8

17] *if* Investment Goal = HOME OWNERSHIP
then Category Of Fund = G&I .9

18] *if* Investment Goal = CURRENT INCOME
then Category Of Fund = INCOME .9

19] *if* Investment Goal = INVEST SPARE CASH and
Risk Tolerance = LOW
then Category Of Fund = CONSERVATIVE GROWTH .9

20] *if* Investment Goal = INVEST SPARE CASH and
Risk Tolerance = MEDIUM
then Category Of Fund = G&I .8

21] *if* Investment Goal = INVEST SPARE CASH and
Risk Tolerance = HIGH
then Category Of Fund = AGGRESSIVE .8

22] *if* Investment Goal = INVEST SPARE CASH and
Risk Tolerance = MEDIUM and
Tax Bracket = HIGH
then Category Of Fund = TAX-FREE .9

RULES ABOUT: what your Life Stage is

23] *if* Your Age > 65
then Life Stage = RETIRED .8

24] *if* Your Age <= 65
then Life Stage = NOT-RETIRED .8

RULES ABOUT: what Investment Goal to select

31] *if* Pension = NO and
Individual Retirement Account = NO
then Investment Goal = RETIREMENT 1.0

32] *if* Have Children = YES and
Children Headed For College = YES and
Children's Education Already Funded = NO
then Investment Goal = CHILDREN'S EDUCATION .8

33] *if* Own Home = NO and
Want Home = YES
then Investment Goal = HOME OWNERSHIP .8

34] *if* Life Stage = RETIRED
then Investment Goal = CURRENT INCOME .9

35] *if* Have Pension = YES
then Has Retirement Vehicle = YES 1.0

36] *if* Has IRA = YES
then Has Retirement Vehicle = YES 1.0

37] *if* Own Home = YES or Want Home = NO, and
Pension = YES or Individual Retirement Account = YES, and
Have Children = NO or Children's Education Already Funded = YES, and
Life Stage = NOT-RETIRED
then Investment Goal = INVEST SPARE CASH .8

RULES ABOUT: what your Risk Tolerance is

41] *if* Enjoy Gambling = YES
then Risk Tolerance = HIGH .8

42] *if* Budgeting Very Important = YES
then Risk Tolerance = LOW .8

43] *if* Worry About Money At Night = YES
then Risk Tolerance = LOW .8

44] *if* Budget But Splurge Sometimes = YES
then Risk Tolerance = MEDIUM .8

RULES ABOUT: whether you have Children Headed For College

45] *if* Have Children = YES and
Age Of Youngest Child < 16
then Children Headed For College = YES 1.0

46] *if* Have Children = YES and
Age Of Youngest Child >= 16
then Children Headed For College = NO 1.0

47] *if* Have Children = NO
then Children Headed For College = NO. 1.0

RULES ABOUT: whether Children's Education Already Funded

51] *if* Have Children = YES
College Tuition Level = CHEAP
then Children's Education Already Funded = YES 1.0

52] *if* Have Children = YES
Children Have Scholarship = YES
then Children's Education Already Funded = YES 1.0

53] *if* Have Children = YES
Children Eligible For Loans = YES
then Children's Education Already Funded = YES 1.0

54] *if* Have Children = YES
Children Have Trust Fund = YES
then Children's Education Already Funded = YES 1.0

55] *if* Have Children = YES
Children Have Scholarship = NO and
Children Eligible For Loans = NO and
Children Have Trust Fund = NO
then Children's Education Already Funded = NO. 1.0