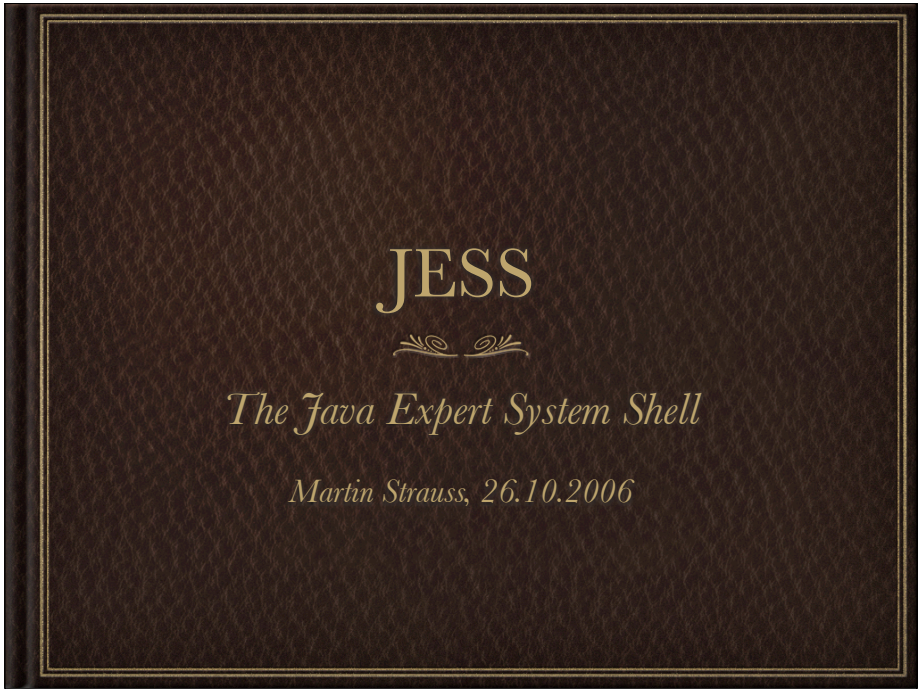


1

First of two talks on Jess.

I will introduce the language, + everything you need to write a JESS program. Then we'll write a simple program as an example.

Next week, Jens will go into more detail about how JESS works, as well as some of the advanced features of the language, with a more detailed example to illustrate these.

(terminal & vim font: Monaco 20pt)

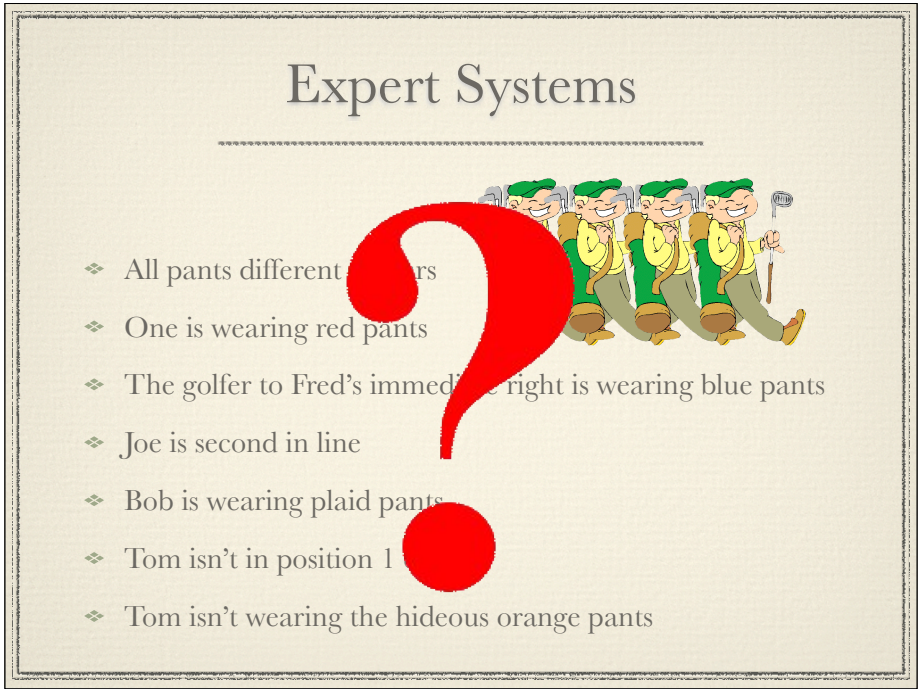
Expert Systems

- All pants different colours
- One is wearing red pants
- The golfer to Fred's immediate right is wearing blue pants
- Joe is second in line
- Bob is wearing plaid pants
- Tom isn't in position 1 or 4
- Tom isn't wearing the hideous orange pants

2

First of all... Jess stands for the "Java Expert System Shell". What is an "Expert System?"

well, let's say we have the following problem. How do we solve it?



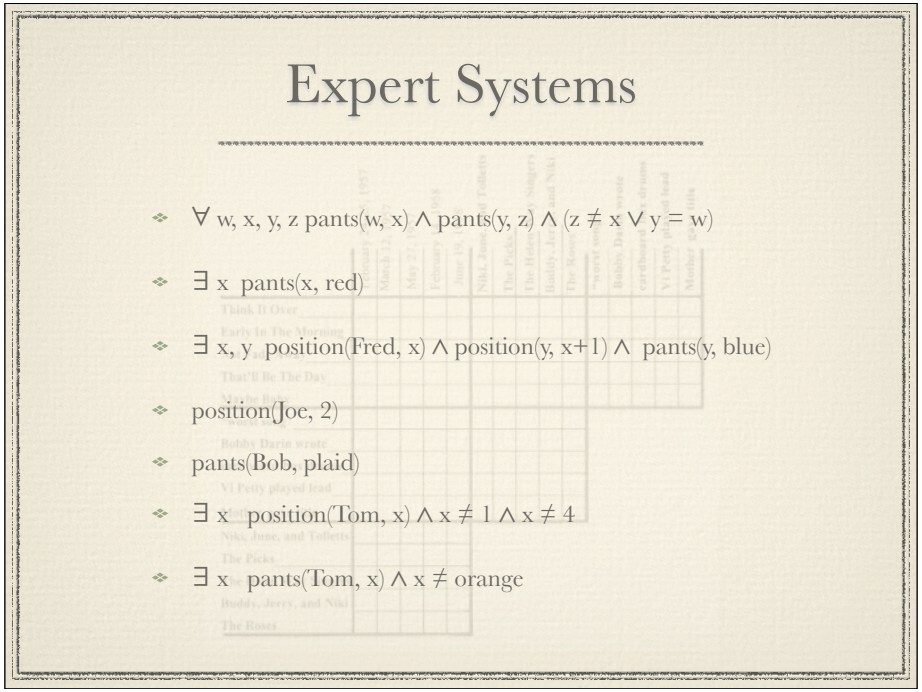
First of all... Jess stands for the "Java Expert System Shell". What is an "Expert System?"

well, let's say we have the following problem. How do we solve it?

Expert Systems

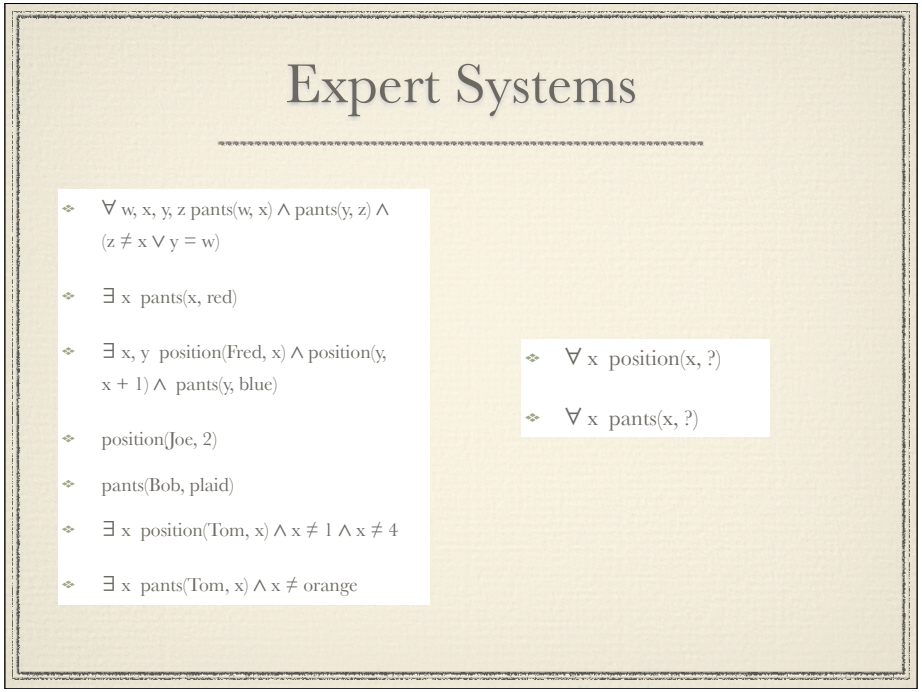
This sort of problem is precisely what I used to do as a child, calling it a “logic puzzle”. You’d enter all possible combinations of facts on a table such as this, and then tick and cross the boxes depending on whether the facts were true or false.

But this is a specific ALGORITHM for solving such a problem. What if we only had to specify it in some standard way, and could let a computer solve it for us?



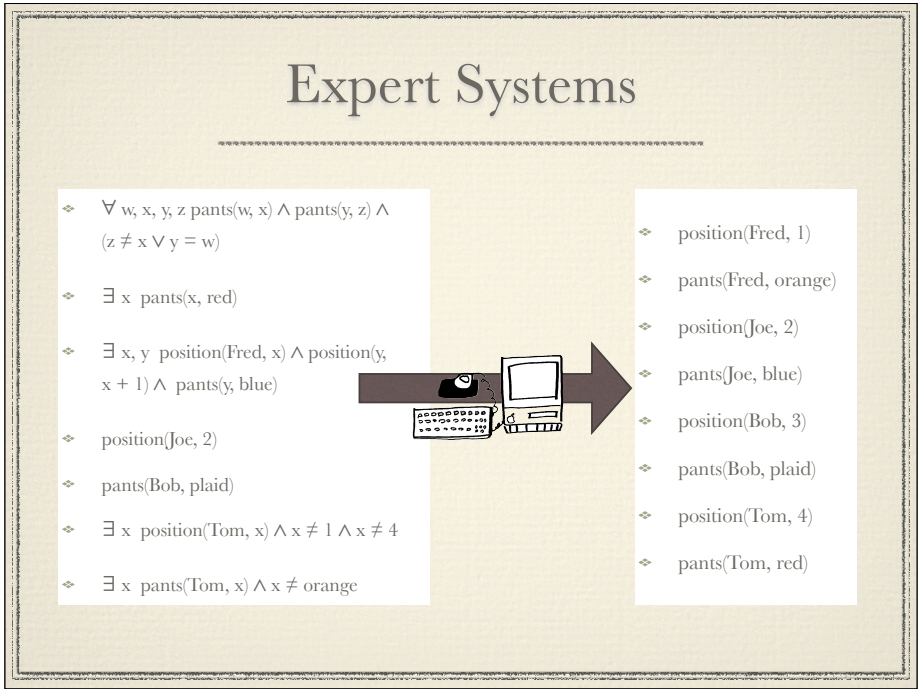
This sort of problem is precisely what I used to do as a child, calling it a “logic puzzle”. You’d enter all possible combinations of facts on a table such as this, and then tick and cross the boxes depending on whether the facts were true or false.

But this is a specific ALGORITHM for solving such a problem. What if we only had to specify it in some standard way, and could let a computer solve it for us?



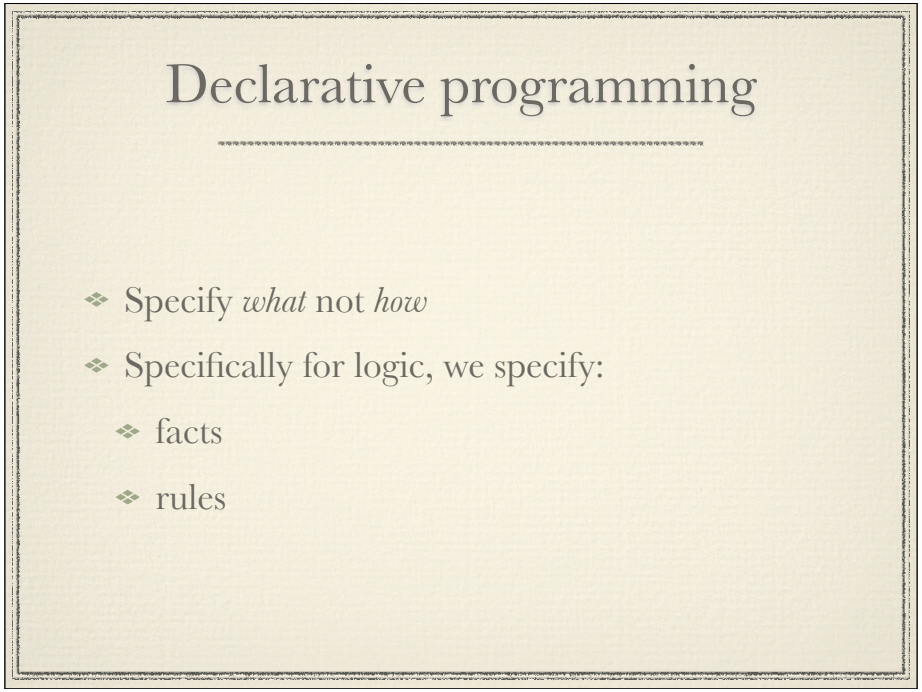
Specifically, what if we state the following facts, and then ask the following questions?

Expert system is precisely this: a system that can reason about the world, and take appropriate actions based on some kind of knowledge about the world.



Specifically, what if we state the following facts, and then ask the following questions?

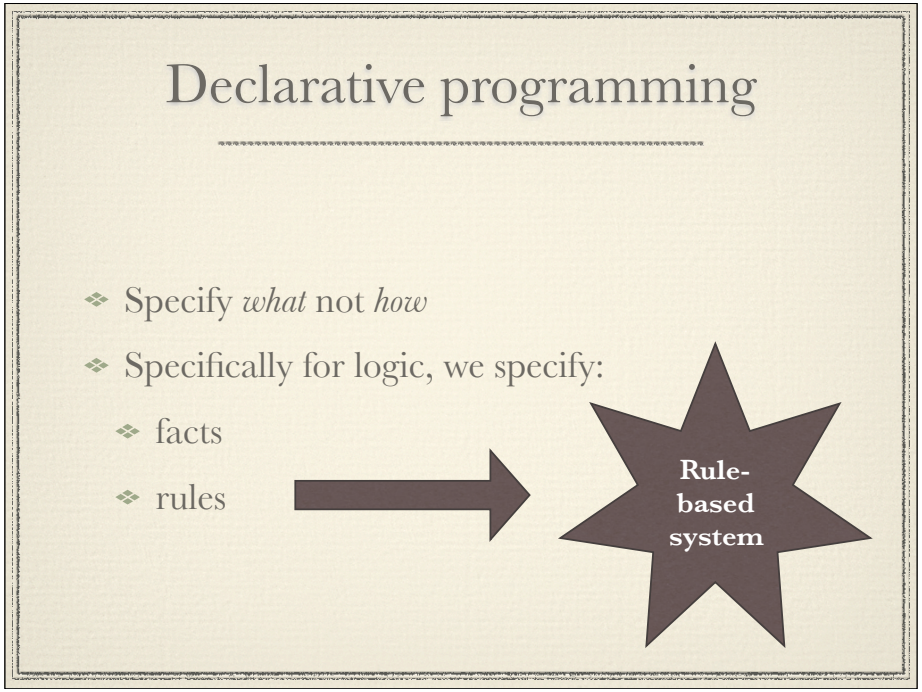
Expert system is precisely this: a system that can reason about the world, and take appropriate actions based on some kind of knowledge about the world.



This sort of thing – specifying what the system knows, and letting its internal algorithms decide what to do – comes under the general heading of “Declarative Programming”.

Imperative programming => specify steps to do something
Declarative programming => specify facts about something (eg, the definition of a function, or a logic rule).

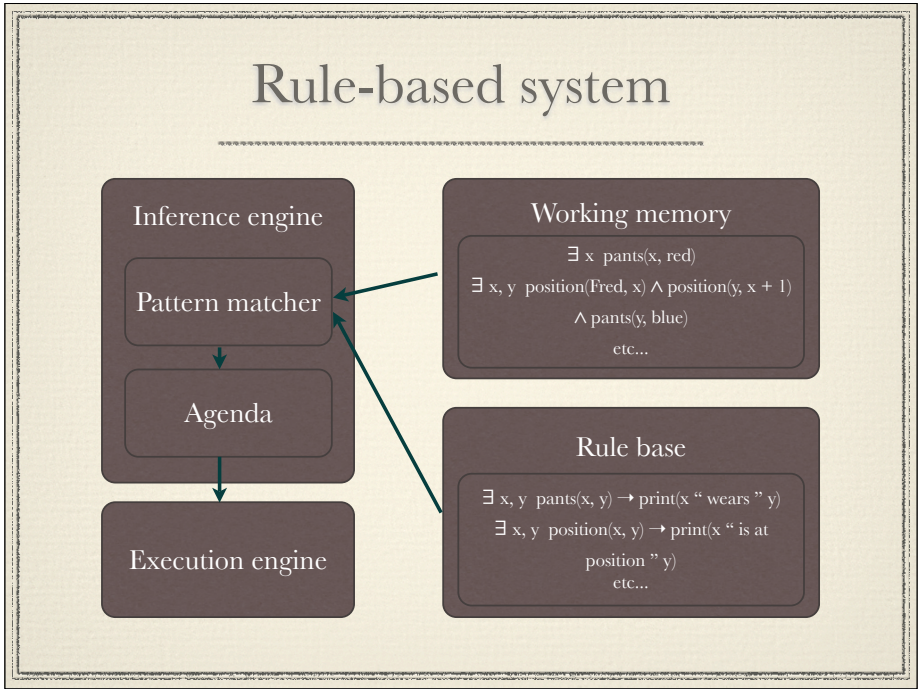
Here, we want to specify facts and rules, and let a reasoning engine decide the steps ==> this is called a “rule-based system”.



This sort of thing – specifying what the system knows, and letting its internal algorithms decide what to do – comes under the general heading of “Declarative Programming”.

Imperative programming => specify steps to do something
Declarative programming => specify facts about something (eg, the definition of a function, or a logic rule).

Here, we want to specify facts and rules, and let a reasoning engine decide the steps ==> this is called a “rule-based system”.



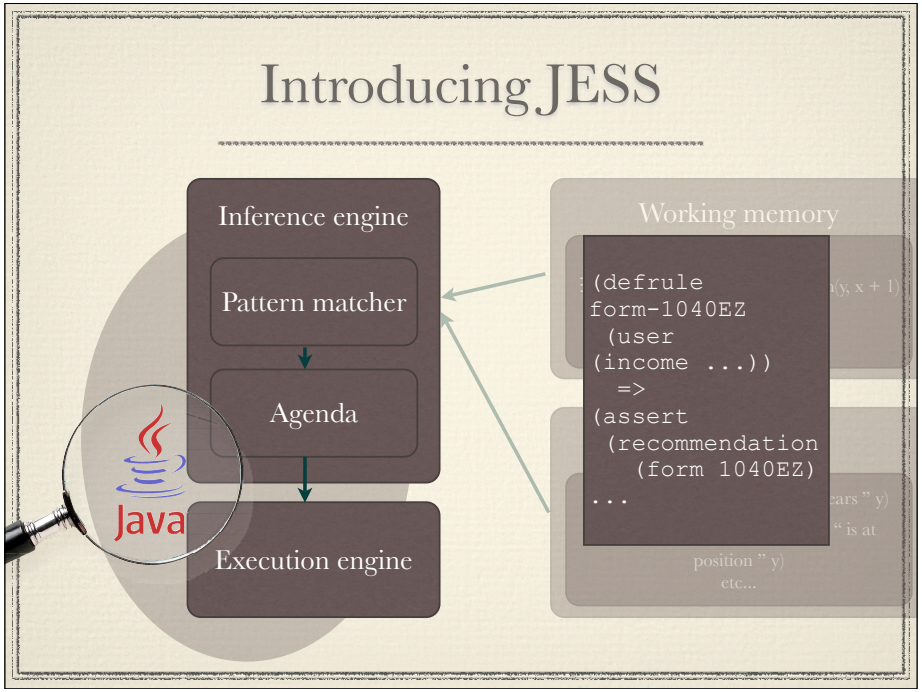
This is the general architecture of a rule-based system.

Working memory stores facts. also called "fact base". Here are our facts from earlier.

Rule base stores rules. The rules and the facts together are our "program".
The golfers was a simplistic example, since all information is in the facts, we just want to explicitly print out what we already implicitly know.

A more sophisticated system can have more complicated and powerful rules: for example, a tax form advice system.

The inference engine is the magic "black box" that does our reasoning.

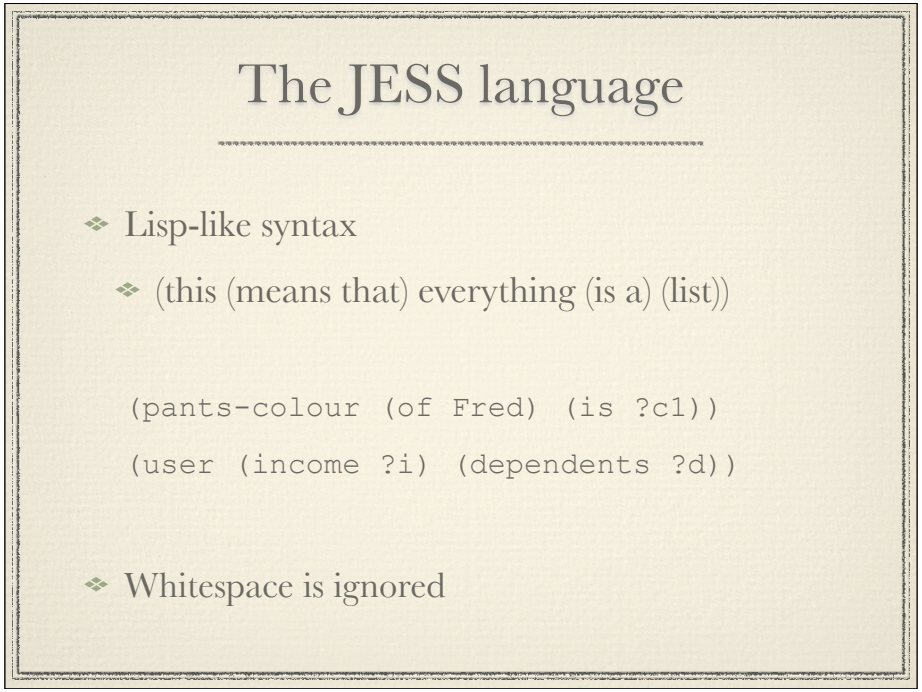


Jess is a rule engine (point) and a language (point) [scripting language, also specification language as mentioned before].

Implemented in Java.

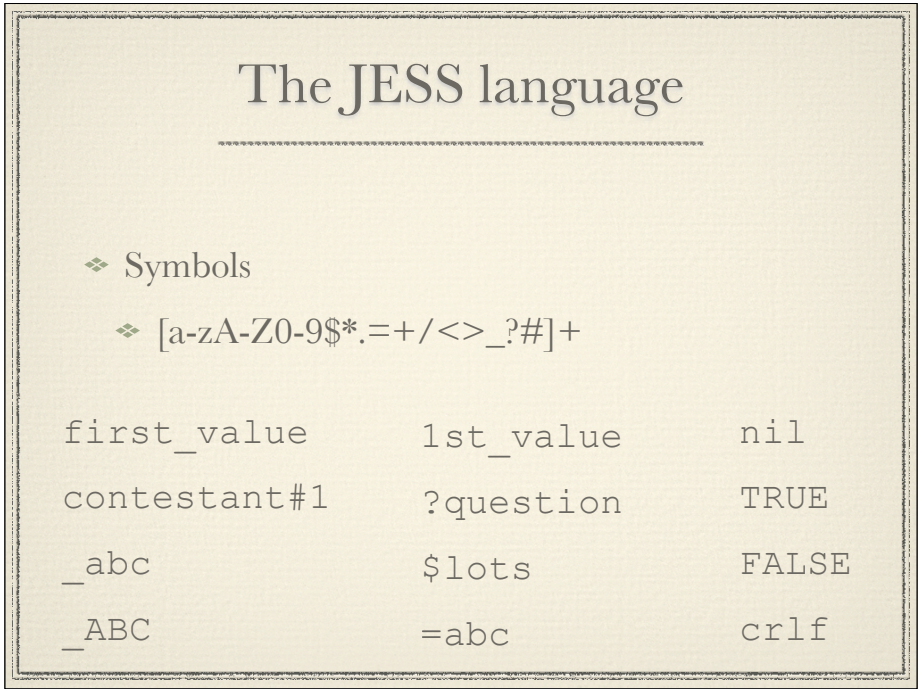
Inspired by CLIPS, so if you're familiar with CLIPS some of this might look a little familiar.

alt-tab and fire up Jess, so we can see what we're playing with.



I'm going to introduce the language itself.

The language has lisp-like syntax: everything is a list, and whitespace is ignored.



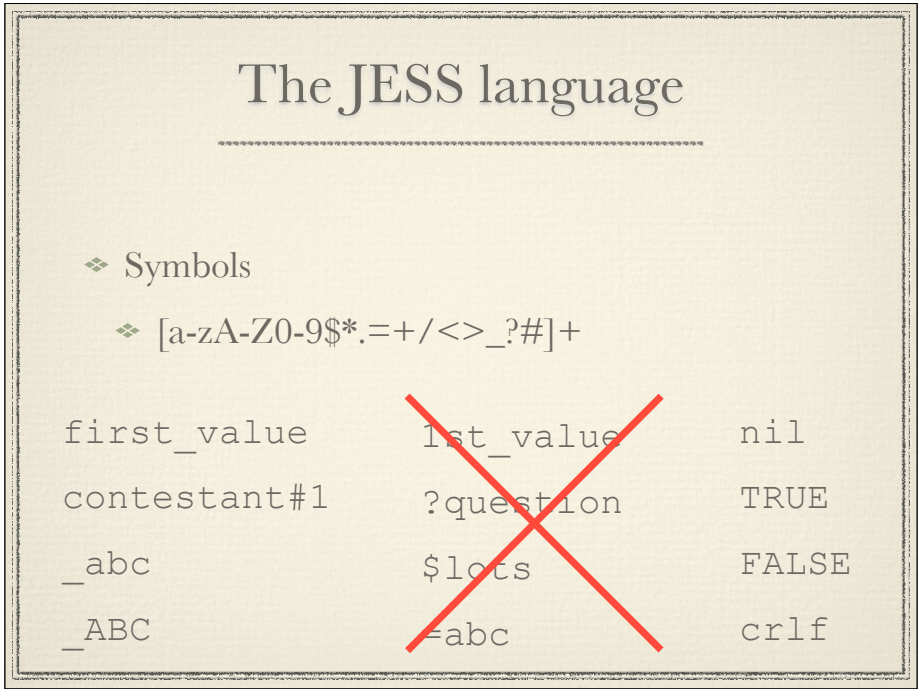
The basic unit of JESS is the symbol. => similar to identifiers in Java.

These are the valid characters for a symbol.

BUT - may not begin with a number, or $? and = (special meanings)

case sensitive.

special symbols:
nil == java null



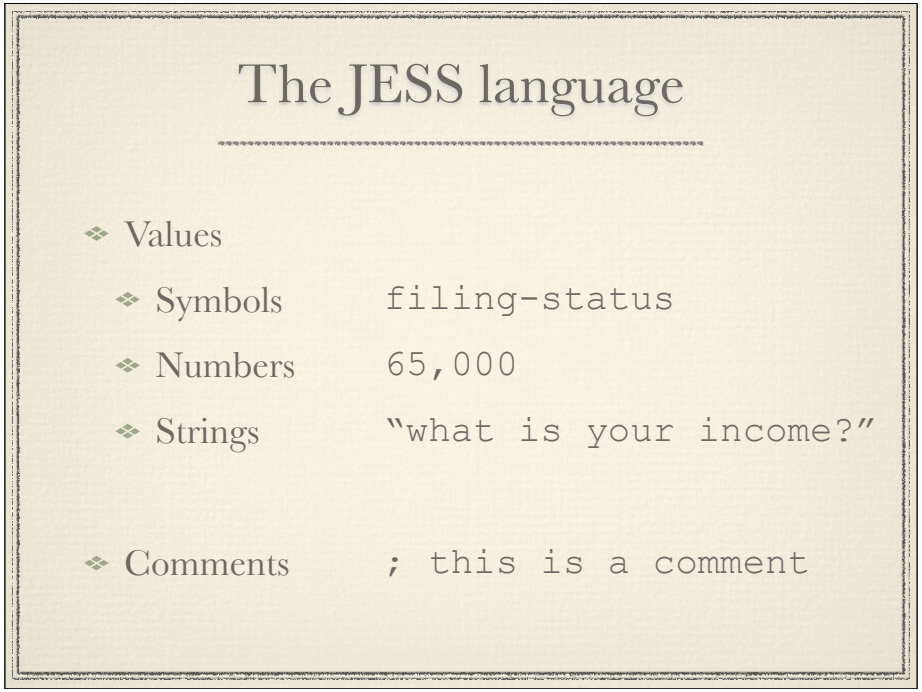
The basic unit of JESS is the symbol. => similar to identifiers in Java.

These are the valid characters for a symbol.

BUT - may not begin with a number, or $? and = (special meanings)

case sensitive.

special symbols:
nil == java null



Jess values are either symbols, numbers, or strings, or lists of these.

numbers are either integers or doubles (more next week)
you can't use Java escapes inside strings (except quotation mark), but newlines, tab characters, etc in a string will be respected

semicolon starts a comment.

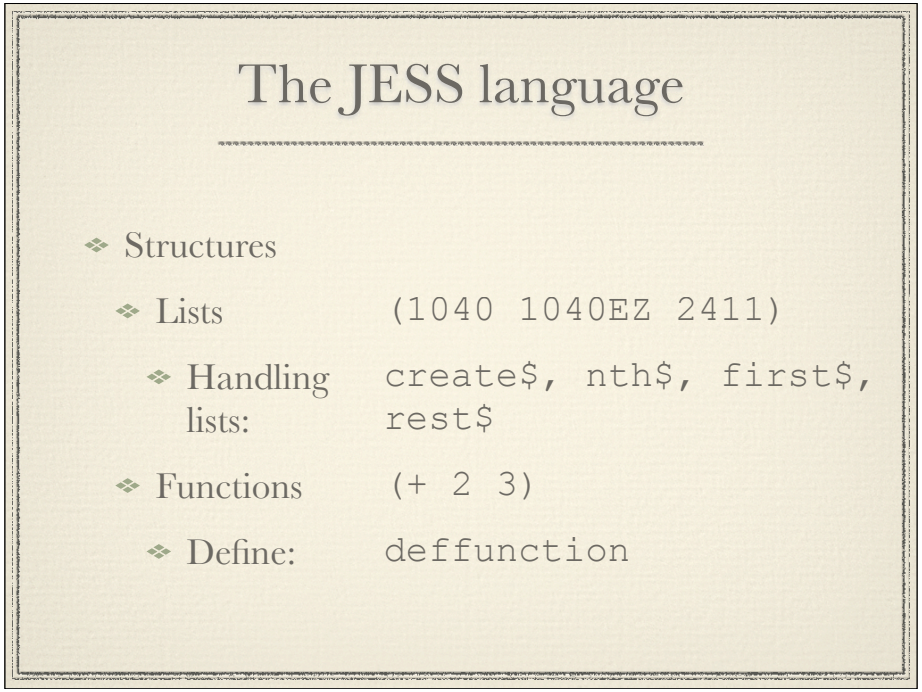
variables start with a question mark.

variables are untyped.

generally only use a-z, dashes and underscores in a variable name

the (reset) command initialises working memory -> clears any standard variables. global variables are persistent across (reset).

multifields are almost identical to normal variables except in two

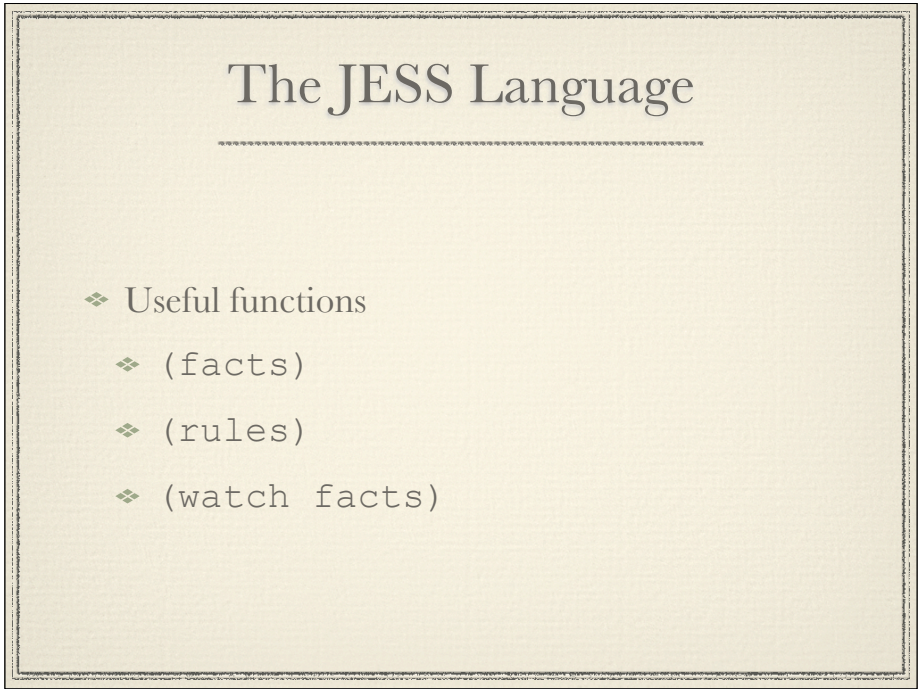


how do we define a list value? we create a list and bind it to some variable
jess> (bind ?something (create$ a b c d e))

Note: lists can't be nested, they get flattened out.

when writing a simple list at the prompt, jess thinks it's a function call.

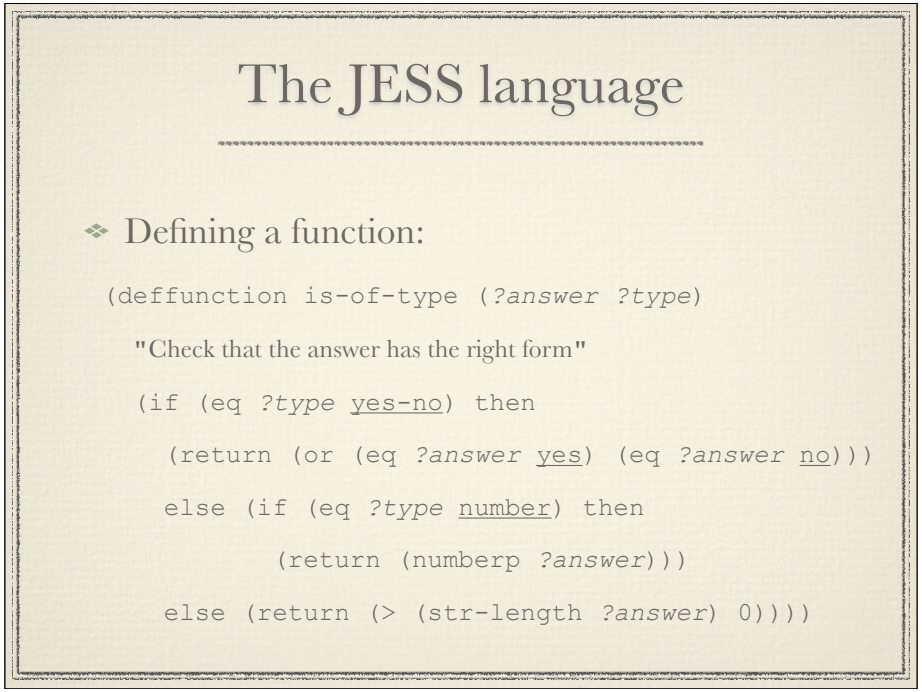
How do we define a function? deffunction.



before we continue - some useful functions that let you see what's going on.

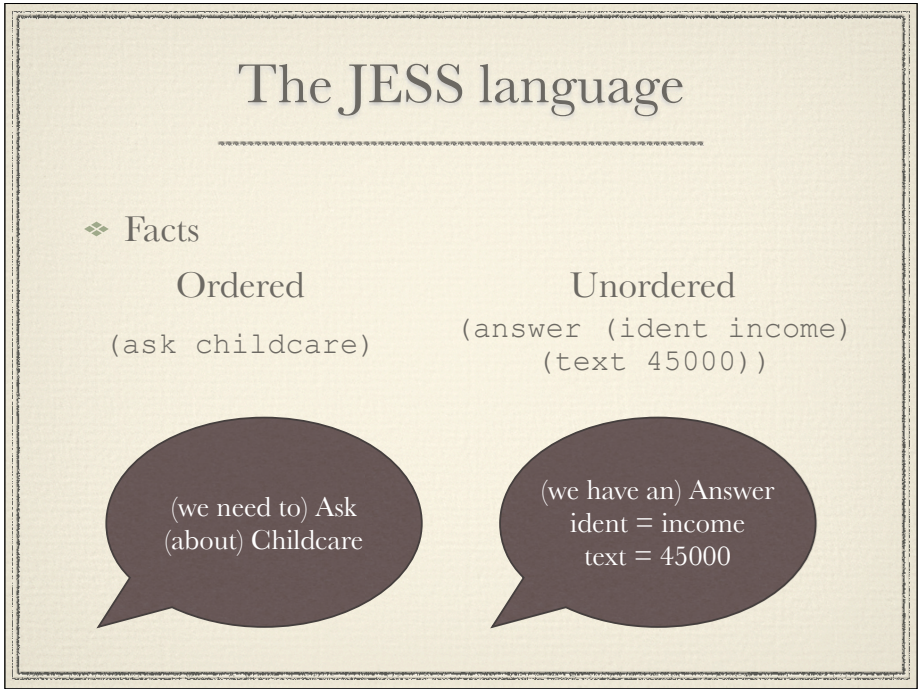
(facts) lists all the facts.
(rules) lists all the rules.

(watch) gets Jess to print output when interesting things happen. You can watch facts, rules, focus, or all. To turn watch off, type (unwatch whatever).



deffunction takes as arguments:
function name
function arguments
a comment (optional)
function body (list of expressions)

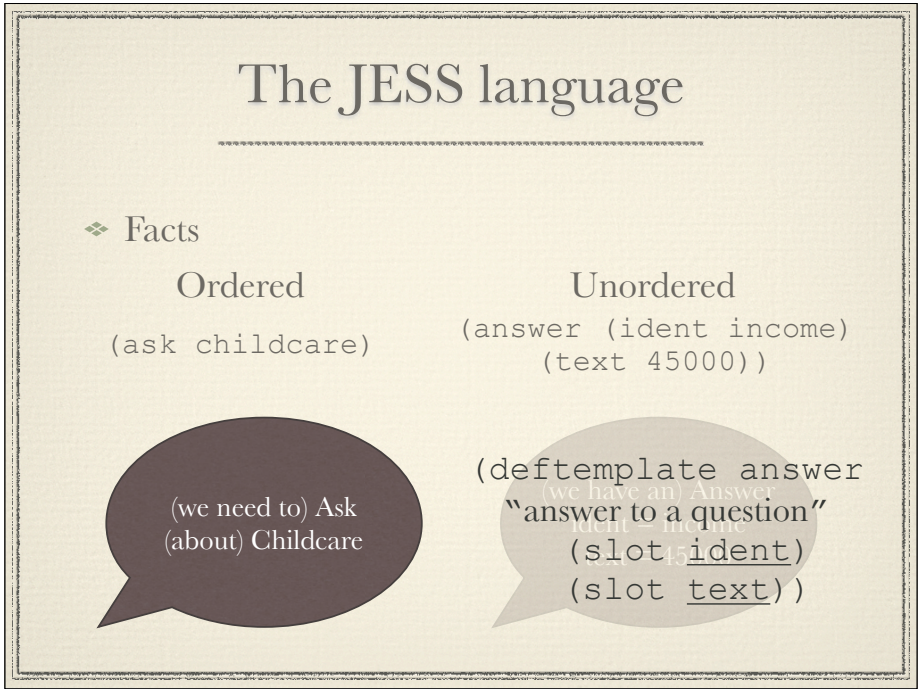
in the arguments we might need to use a multifield to specify “one or more arguments”.



Facts: “ask” or “answer” is the head of the fact; “ident” and “text” are slots.

the difference is: the “slots” in an unordered fact are labelled, hence they can come in any order, hence “unordered”.

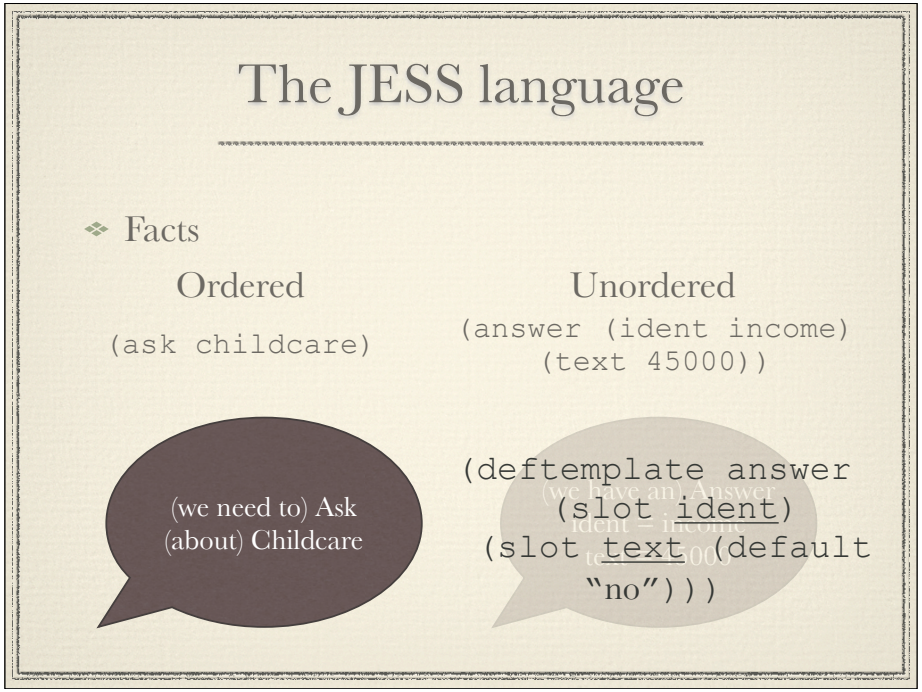
if you want to define a multislot, you have “multislot” here instead of “slot”. This is the other use of multifields.



Facts: "ask" or "answer" is the head of the fact; "ident" and "text" are slots.

the difference is: the "slots" in an unordered fact are labelled, hence they can come in any order, hence "unordered".

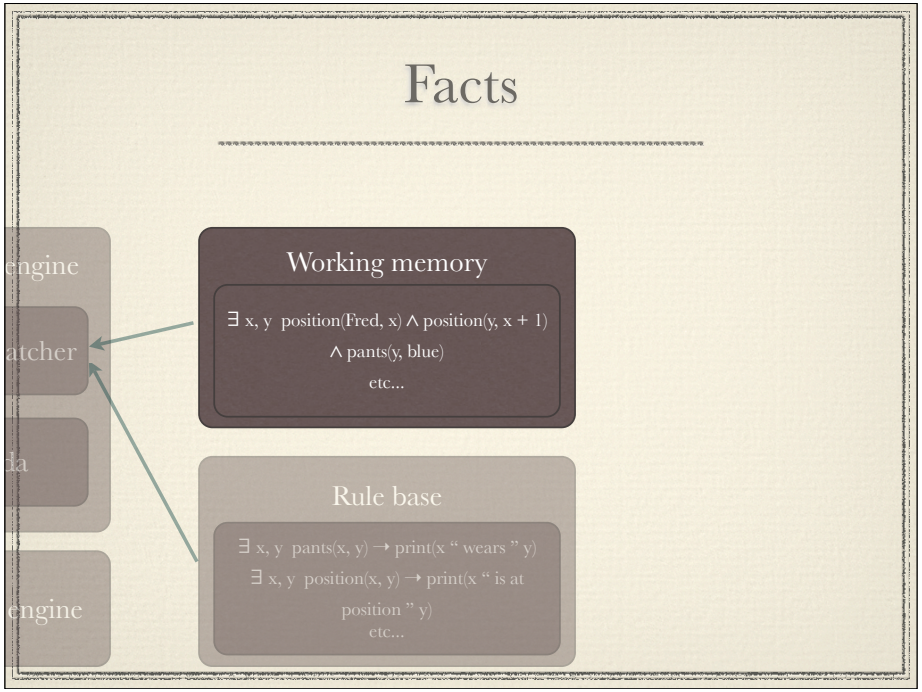
if you want to define a multislot, you have "multislot" here instead of "slot". This is the other use of multifields.



Facts: "ask" or "answer" is the head of the fact; "ident" and "text" are slots.

the difference is: the "slots" in an unordered fact are labelled, hence they can come in any order, hence "unordered".

if you want to define a multislot, you have "multislot" here instead of "slot". This is the other use of multifields.

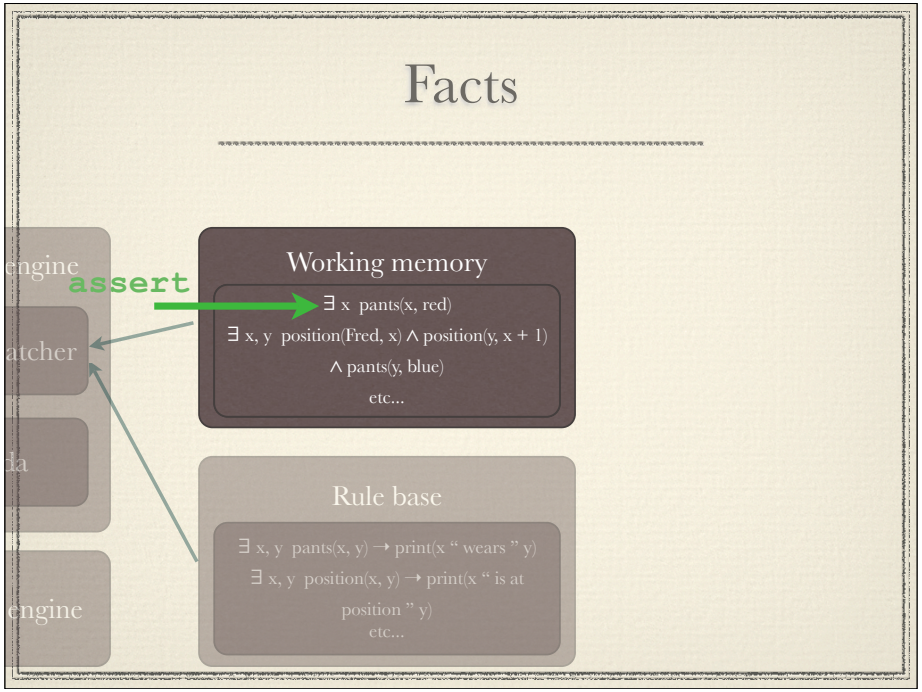


Facts live in the working memory.

Facts are added to our knowledge by ‘assert’,
and removed from our knowledge by ‘retract’.

We clear the working memory with ‘clear’. This wipes everything -> not something we want to do often!

We reset the working memory to its initial state using ‘reset’. By default, this initial state contains only one fact: initial-fact.

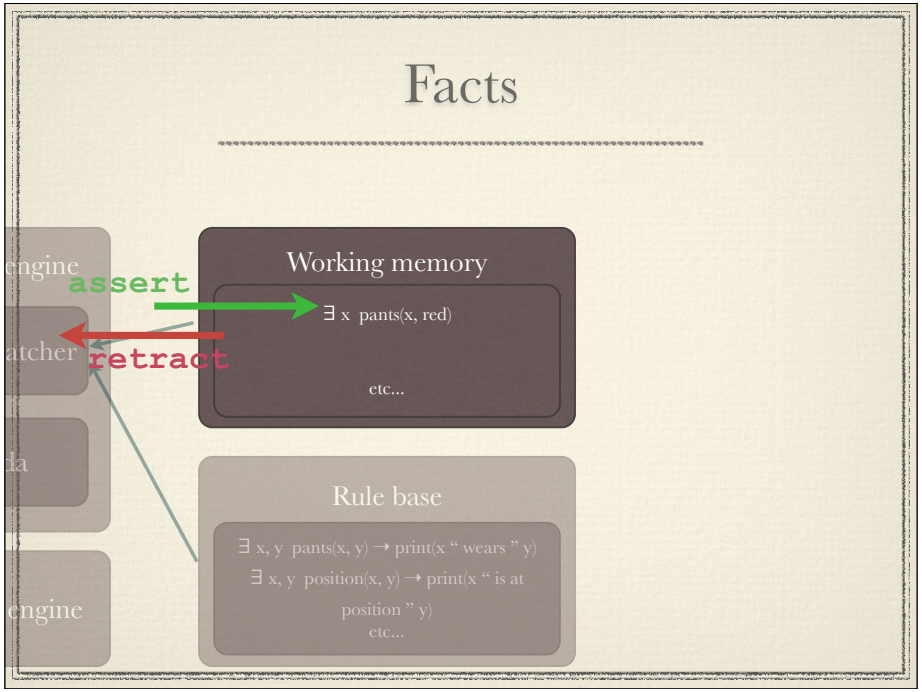


Facts live in the working memory.

Facts are added to our knowledge by ‘assert’,
and removed from our knowledge by ‘retract’.

We clear the working memory with ‘clear’. This wipes everything ->
not something we want to do often!

We reset the working memory to its initial state using ‘reset’. By
default, this initial state contains only one fact: initial-fact.

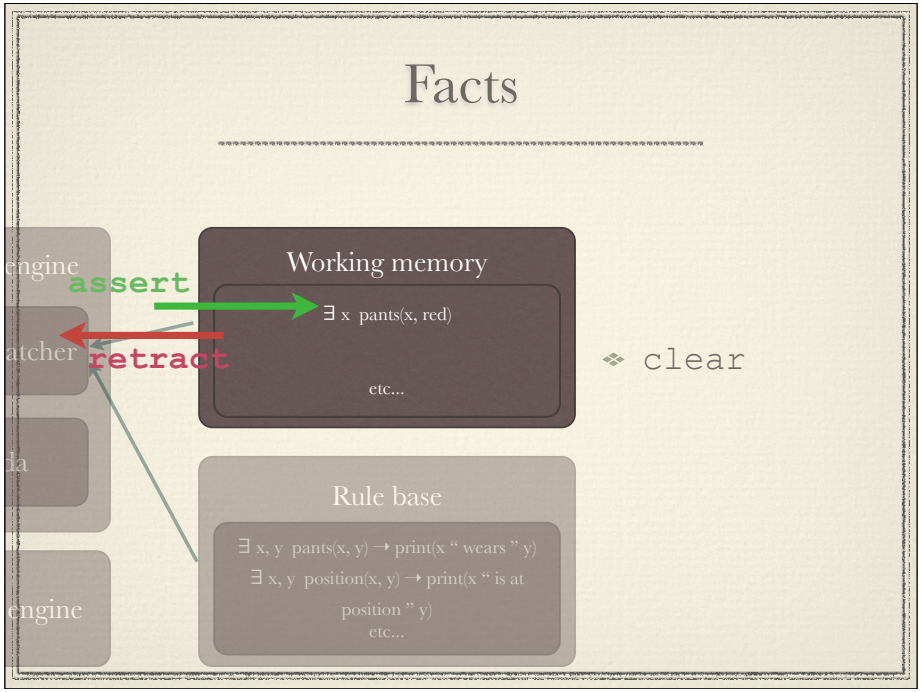


Facts live in the working memory.

Facts are added to our knowledge by ‘assert’,
and removed from our knowledge by ‘retract’.

We clear the working memory with ‘clear’. This wipes everything -> not something we want to do often!

We reset the working memory to its initial state using ‘reset’. By default, this initial state contains only one fact: initial-fact.

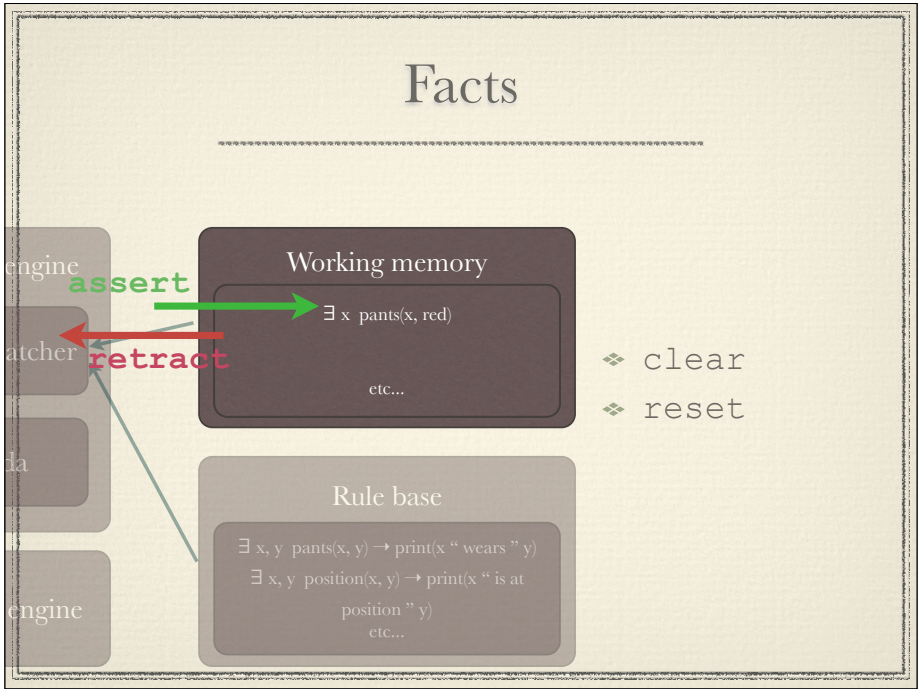


Facts live in the working memory.

Facts are added to our knowledge by ‘assert’,
and removed from our knowledge by ‘retract’.

We clear the working memory with ‘clear’. This wipes everything -> not something we want to do often!

We reset the working memory to its initial state using ‘reset’. By default, this initial state contains only one fact: initial-fact.

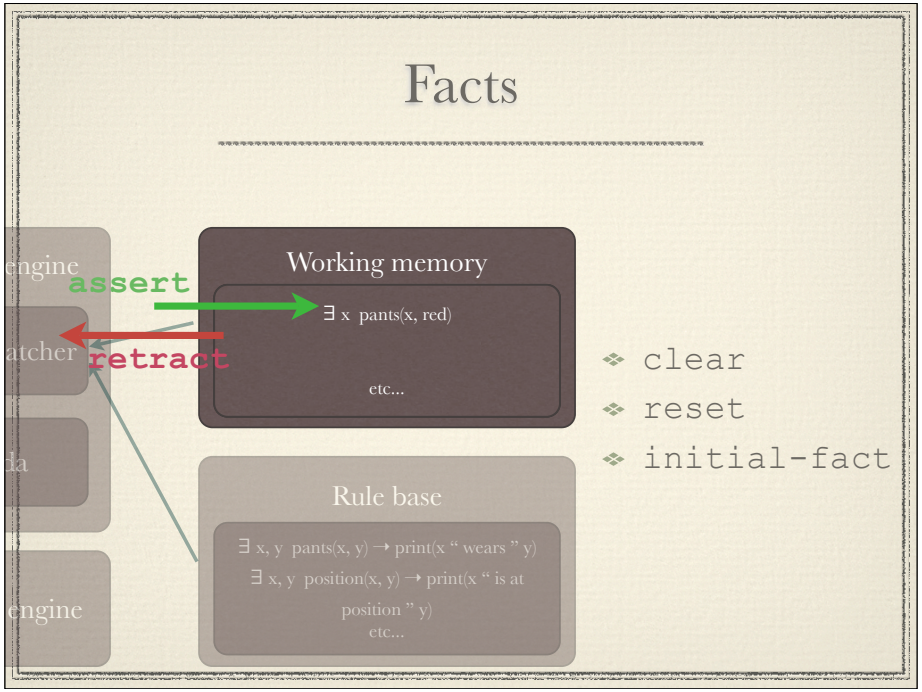


Facts live in the working memory.

Facts are added to our knowledge by ‘assert’,
and removed from our knowledge by ‘retract’.

We clear the working memory with ‘clear’. This wipes everything -> not something we want to do often!

We reset the working memory to its initial state using ‘reset’. By default, this initial state contains only one fact: initial-fact.

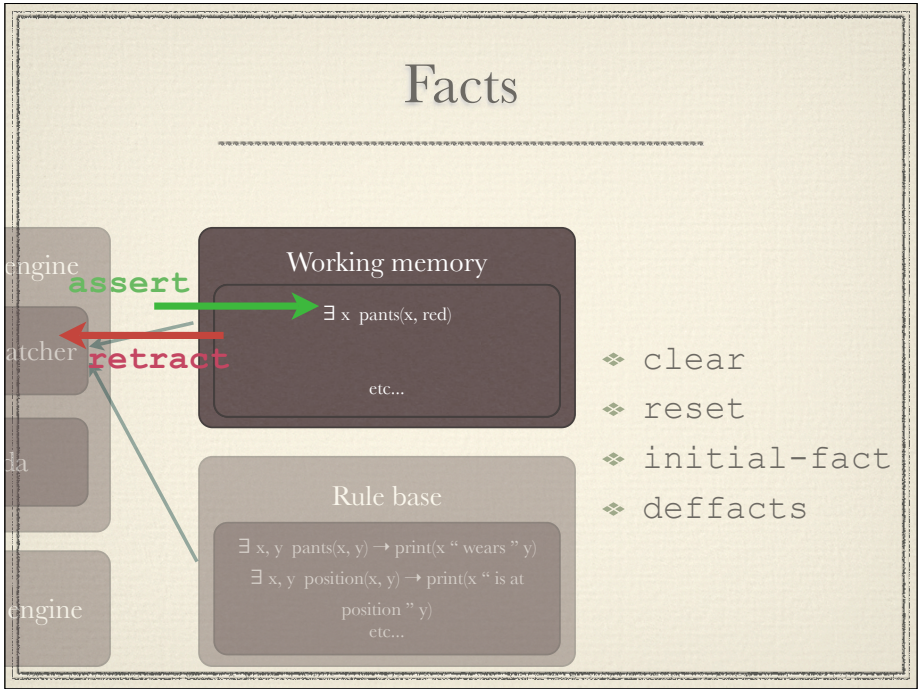


Facts live in the working memory.

Facts are added to our knowledge by ‘assert’,
and removed from our knowledge by ‘retract’.

We clear the working memory with ‘clear’. This wipes everything -> not something we want to do often!

We reset the working memory to its initial state using ‘reset’. By default, this initial state contains only one fact: initial-fact.

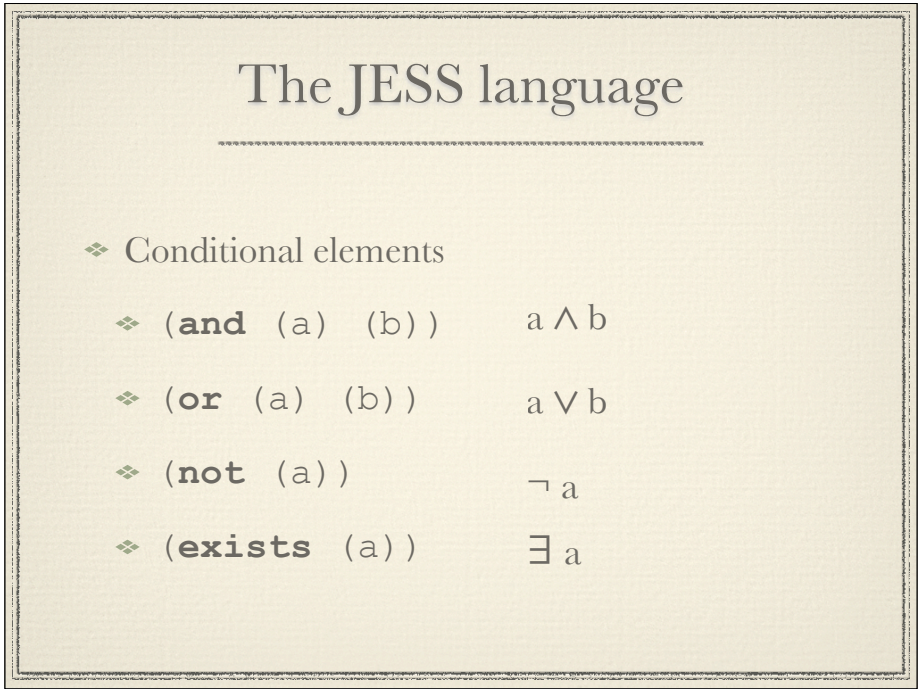


Facts live in the working memory.

Facts are added to our knowledge by ‘assert’,
and removed from our knowledge by ‘retract’.

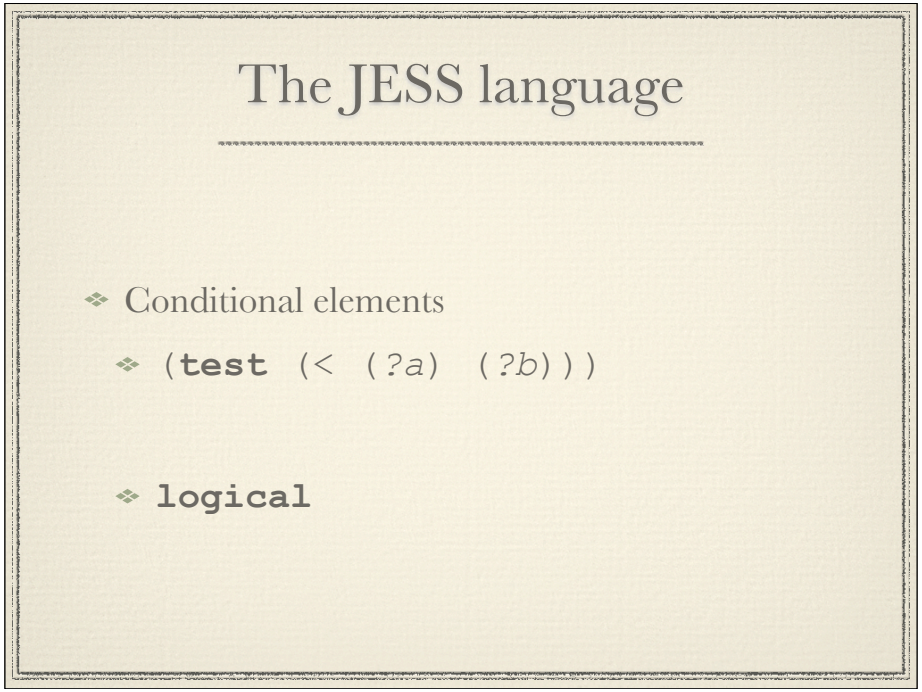
We clear the working memory with ‘clear’. This wipes everything -> not something we want to do often!

We reset the working memory to its initial state using ‘reset’. By default, this initial state contains only one fact: initial-fact.



In order to build up more complex logical statements, we want to be able to combine facts.

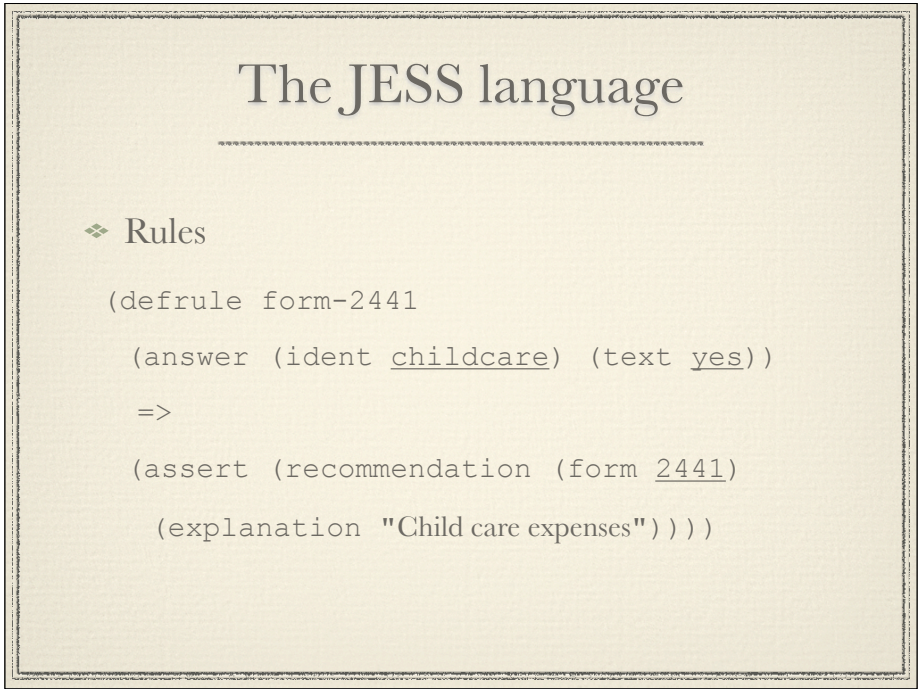
(note, exists (a)) is the same as (not (not (a)))



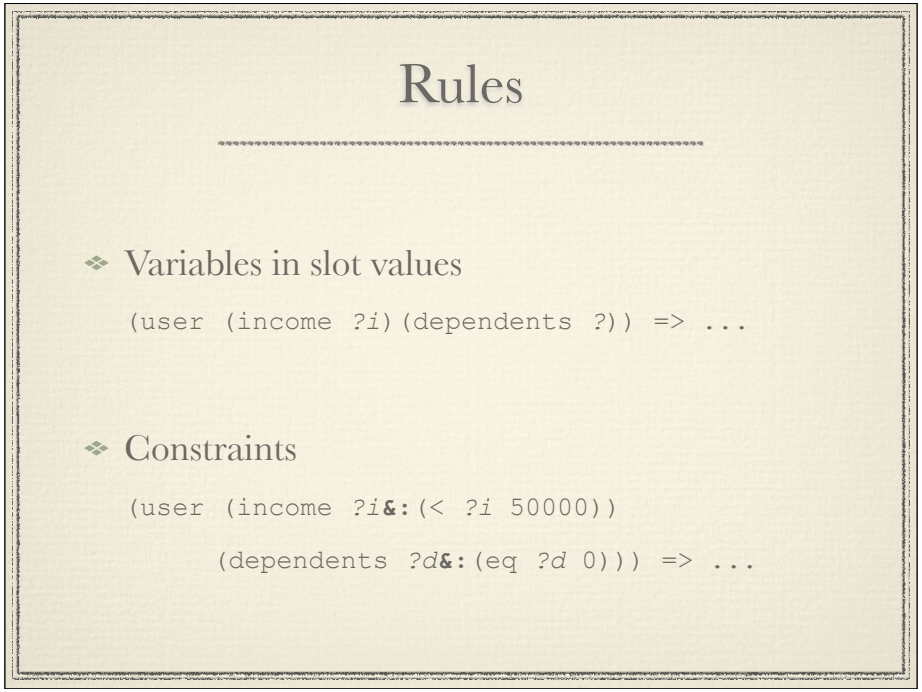
Two more conditional elements are necessary due to the way rules are defined.

Sometimes you want to call a function and evaluate its result: test calls a function and evaluates to true if the function returns anything except FALSE

logical we'll see properly when we define rules... next.

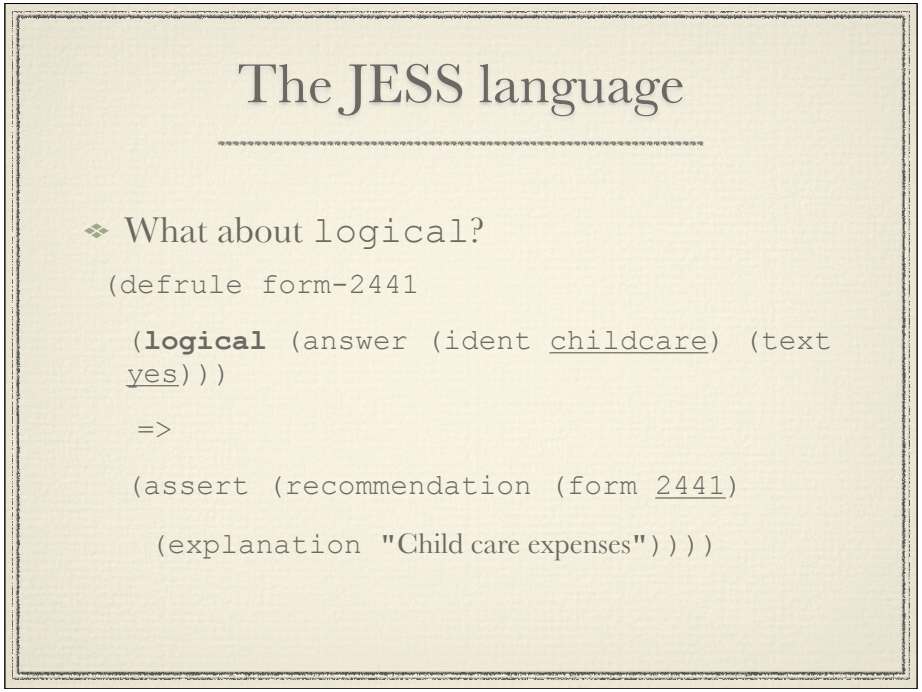


This is what a rule looks like: it has a LHS and a RHS. LHS is matched against facts in our working memory; if it matches, functions in RHS are executed.



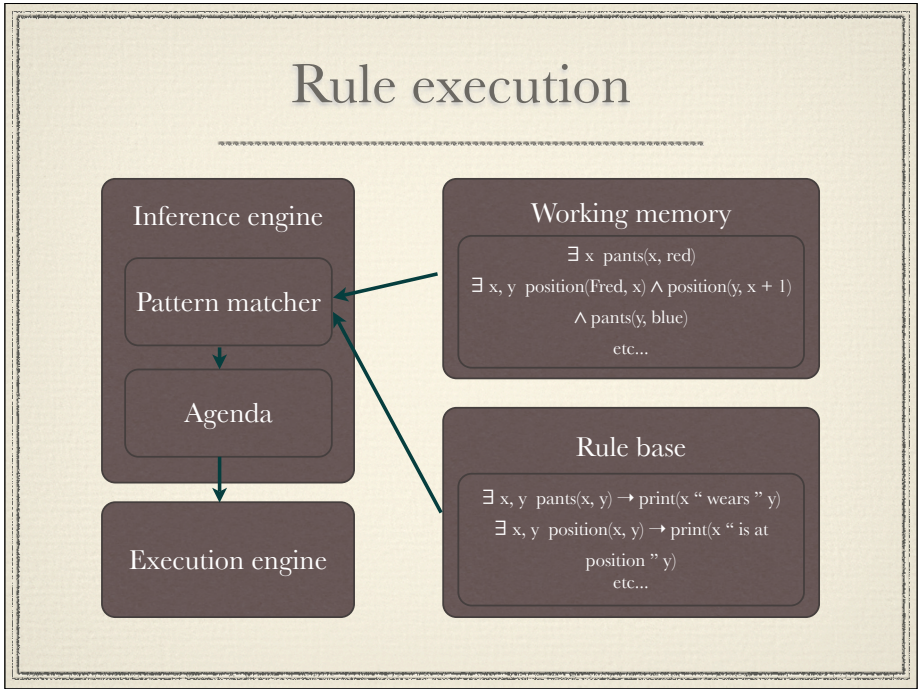
Note that the LHS slots don't have to be filled -> we can specify just variables too, and then we will match any fact that can fill that variable.

Not only can you have variables in the slots on the LHS, but you can also place constraints on those variables using question mark - colon.



logical asserts a logical dependency

that is, whenever (answer (ident childcare) (text yes)) is retracted, (recommendation etc etc) is retracted as well. We say that (recommendation etc etc) is “logically dependent” on (answer etc etc).

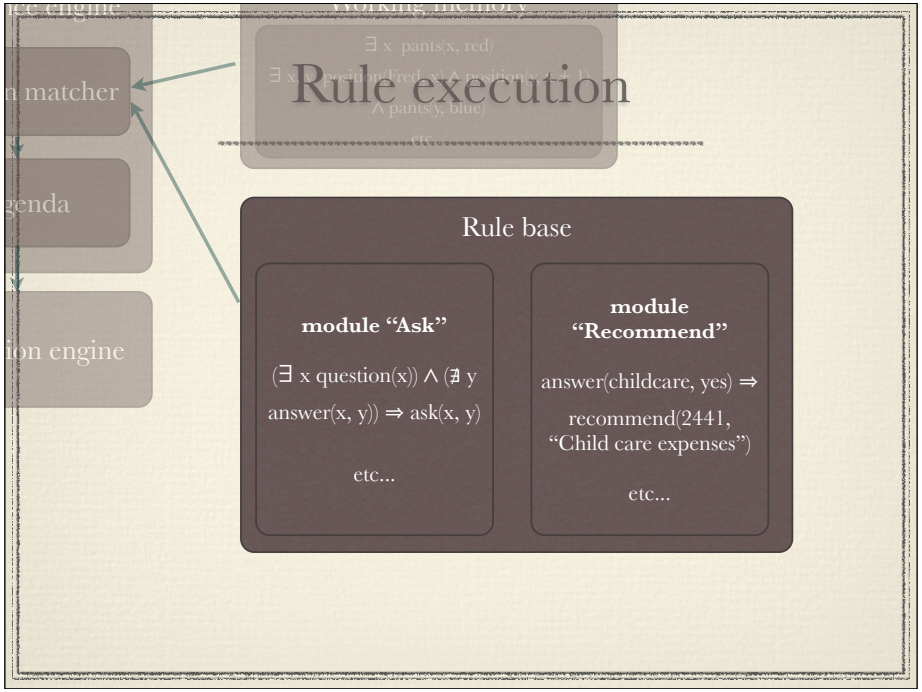


2 more things to say, and now we're getting into murkier waters.

Remember the execution cycle of a rule-based system...

Sometimes we need more control over the "conflict resolution" - the order in which activated rules from the agenda are fired.

You can change a rule's "salience" in the defrule construct - but now we're getting away from purely declarative programming! Not very clean.



Better: partition the rule base into modules. Then we can give different modules focus at different times, but still let the rule engine's own conflict resolution decide which rules from this module to fire.

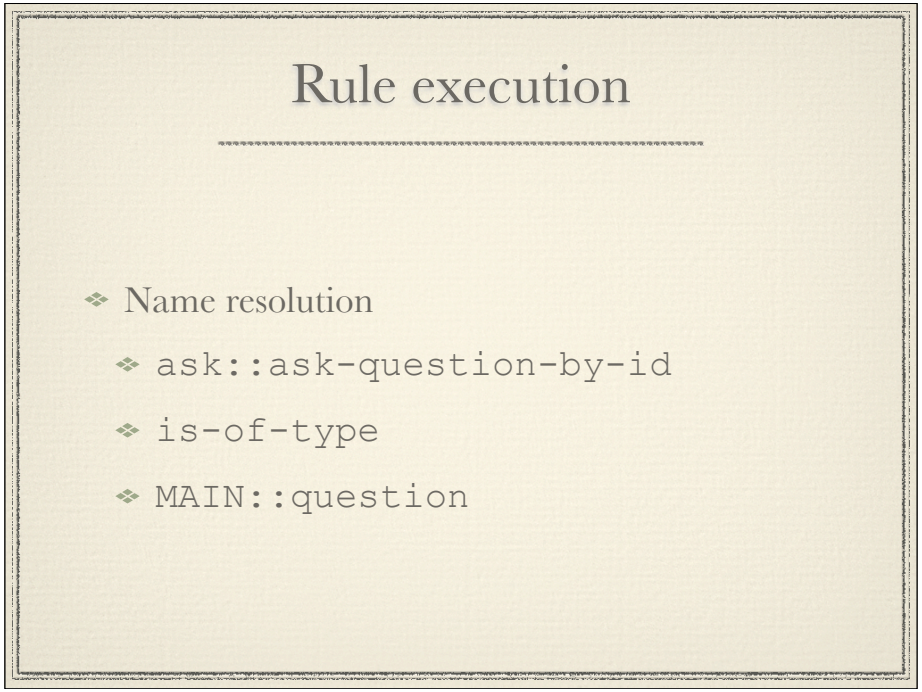
Here we've separated the rules into "ask questions" rules and "make recommendations" rules, so we can have the system ask all its questions at once first, and then make all its recommendations together at the end.

Rule execution

- (**defmodule** interview)
- (**focus** startup interview recommend report)
- (**clear-focus-stack**)
- (**pop-focus**)
- (**get-current-module**)
- (declare **auto-focus** TRUE) & (**return**)

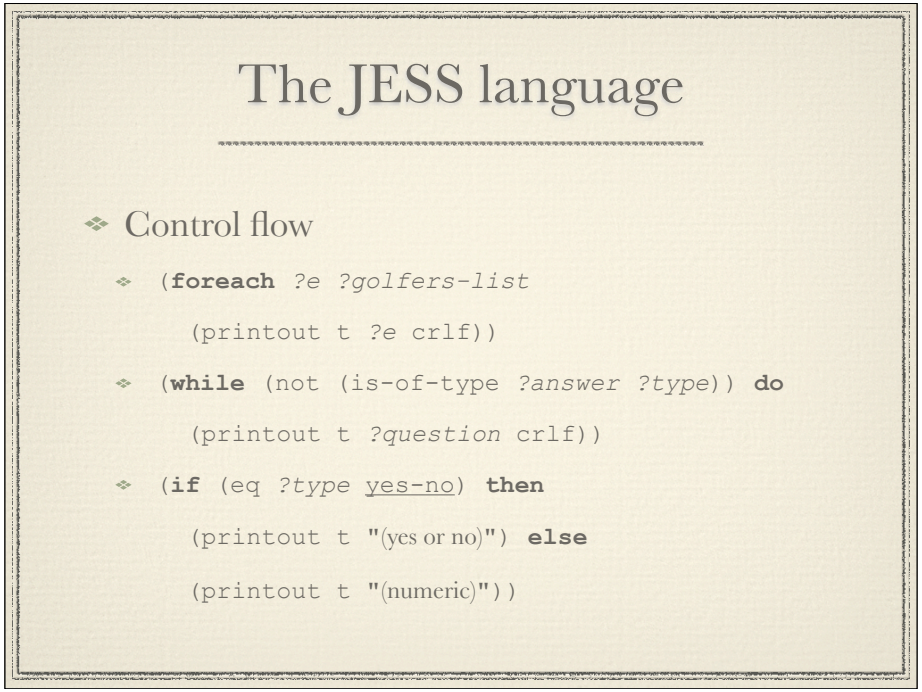
How do we actually do this?

defmodule defines a new module
focus pushes modules onto the focus stack. Focus moves from one
clear-focus-stack clears the stack
pop-focus pops the next module off the stack and gives it focus
get-current-module gets the current module
if a rule has auto-focus declared to be true, then whenever it is activated, it fires first and its module assumes focus. Return returns focus to where it came from. Otherwise, rules fire in the order their



All names that are not explicitly defined in a module are defined in the current module. If no modules have yet been defined, this is “MAIN”.

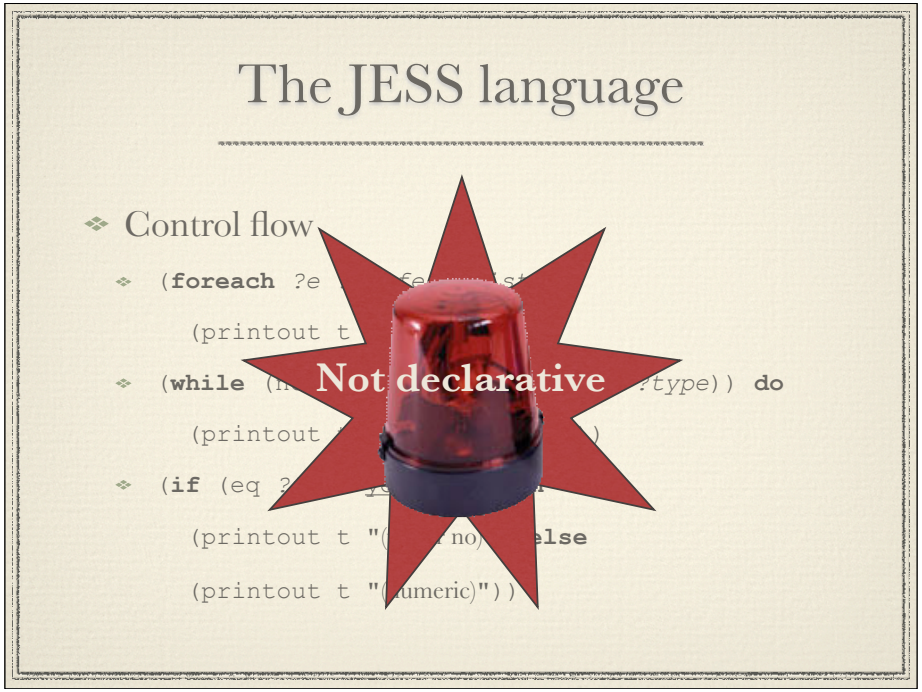
If a name is explicitly referenced, then only that module will be searched. Otherwise, the current module will first be searched, and if no name is matched then MAIN will be searched.



There are a few control-flow constructs you can use while writing a function or the RHS of a rule. They pretty much do what you expect them to do.

BUT: not declarative. [you're specifying the order things should happen.] use with caution.

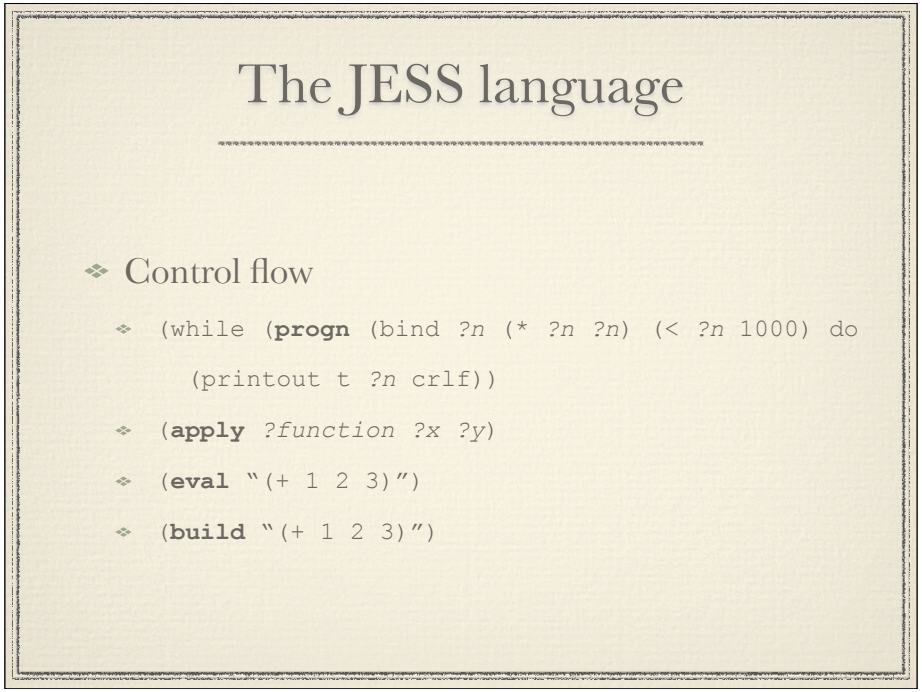
why? because you're turning down the advantages of the built-in rule engine and conflict resolution and reasoning and writing your own algorithm. You may get a more readable program by just asserting



There are a few control-flow constructs you can use while writing a function or the RHS of a rule. They pretty much do what you expect them to do.

BUT: not declarative. [you're specifying the order things should happen.] use with caution.

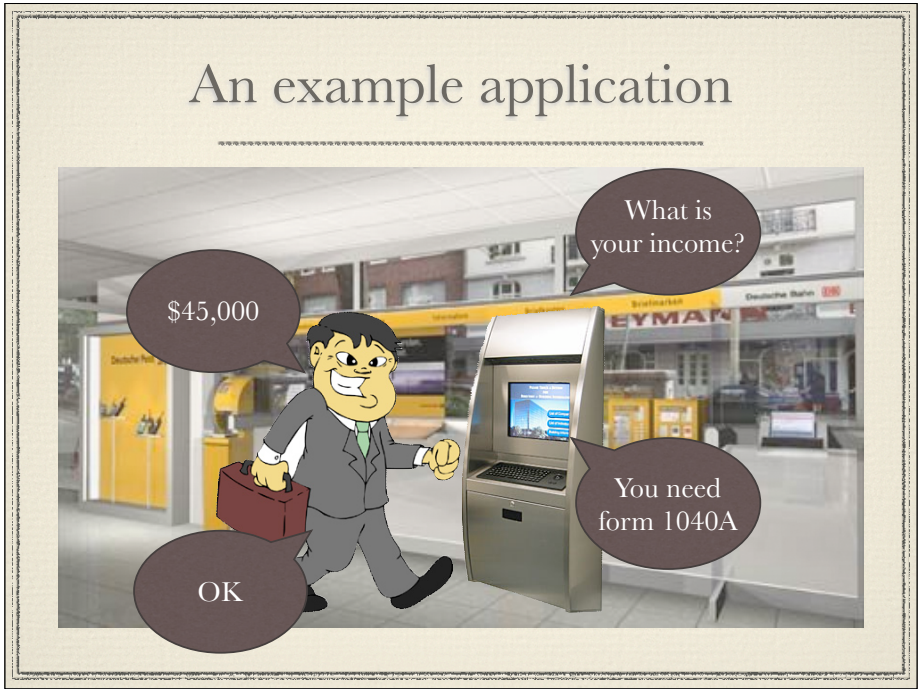
why? because you're turning down the advantages of the built-in rule engine and conflict resolution and reasoning and writing your own algorithm. You may get a more readable program by just asserting



A few more useful control flow statements:

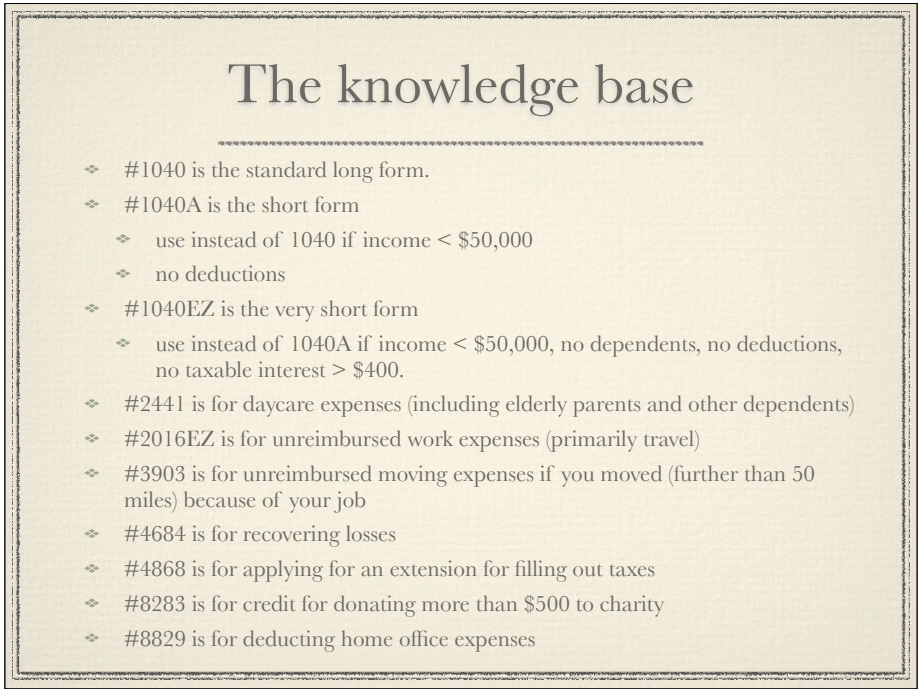
progn takes a list of expressions, executes them and returns the last one
-> used to group multiple into a single expression when syntax requires it (eg here)

apply applies whatever function is in ?function to the arguments. eg, *, +, whatever.



Okay, now we're ready to implement a complete program.

Remember the tax forms advisor I spoke of at the start? We've been seeing bits of the code for it throughout. We'll put it together now.



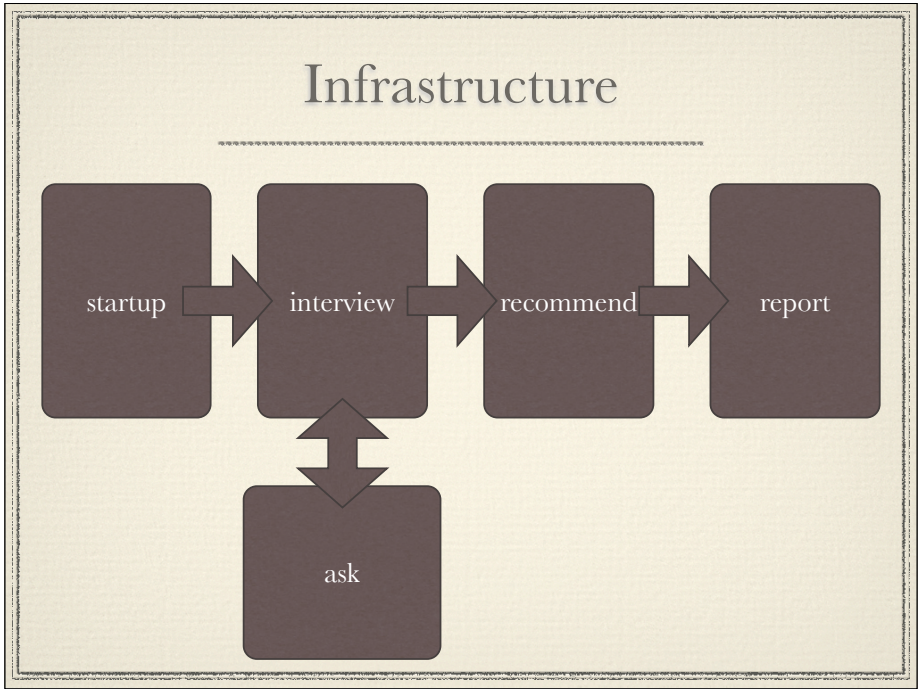
Let's collect all the knowledge we have about tax forms. (Let's say we have this knowledge).

The rules

```
(defrule form-1040EZ
  (user (income ?i&:(< ?i 50000))
        (dependents ?d&:(eq ?d 0)))
  (answer (ident interest) (text no))
  =>
  (assert (recommendation
            (form 1040EZ)
            (explanation "Income below
        threshold, no dependents"))))
```


Each of these points becomes a JESS rule that looks something like this.

And to be honest, that's 95% of the work done.

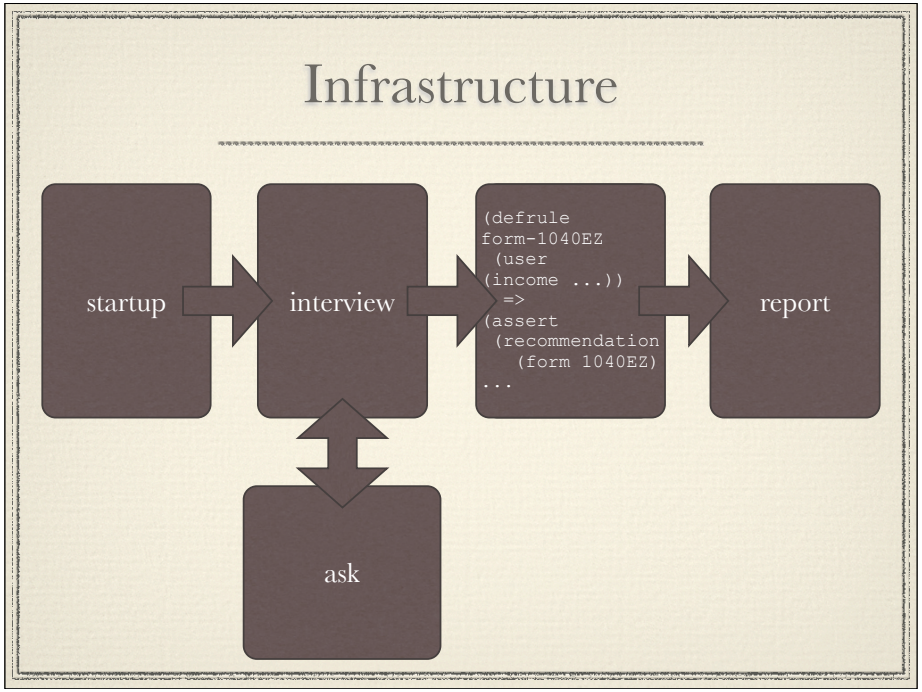


We'll need some infrastructure to interact with the system.

Execution: welcome, ask questions, decide what forms you need, tell you what forms you need. This suggests the modules we want.

Plus, we'll want somewhere for the "if we have a question but no answer, ask the question" rule to live. => 'ask' module.

since all our facts (questions, user answers, generated recommendations) are required by multiple modules, they'll all live in MAIN.

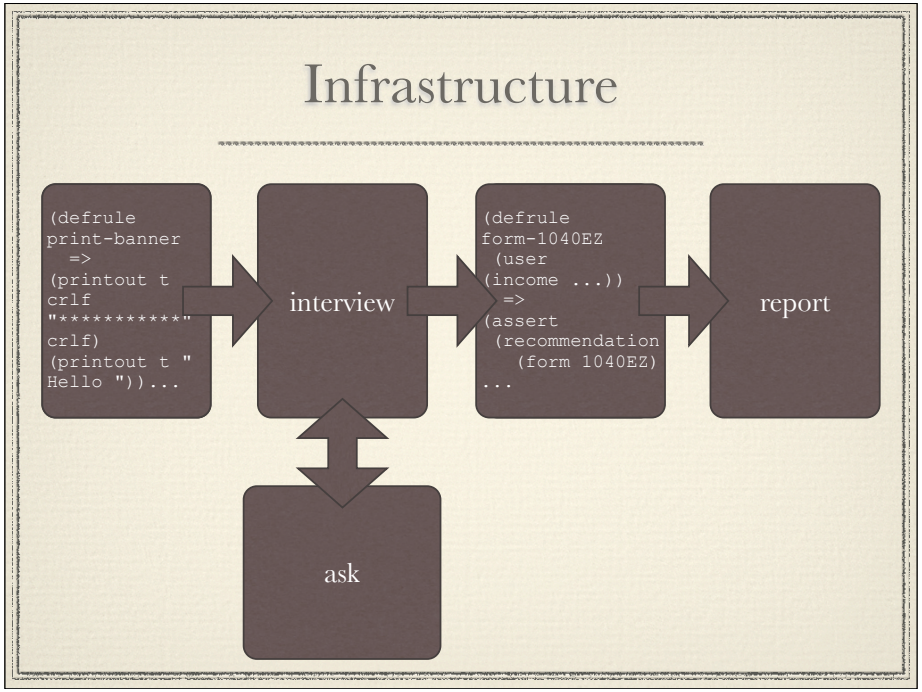


We'll need some infrastructure to interact with the system.

Execution: welcome, ask questions, decide what forms you need, tell you what forms you need. This suggests the modules we want.

Plus, we'll want somewhere for the "if we have a question but no answer, ask the question" rule to live. => 'ask' module.

since all our facts (questions, user answers, generated recommendations) are required by multiple modules, they'll all live in MAIN.

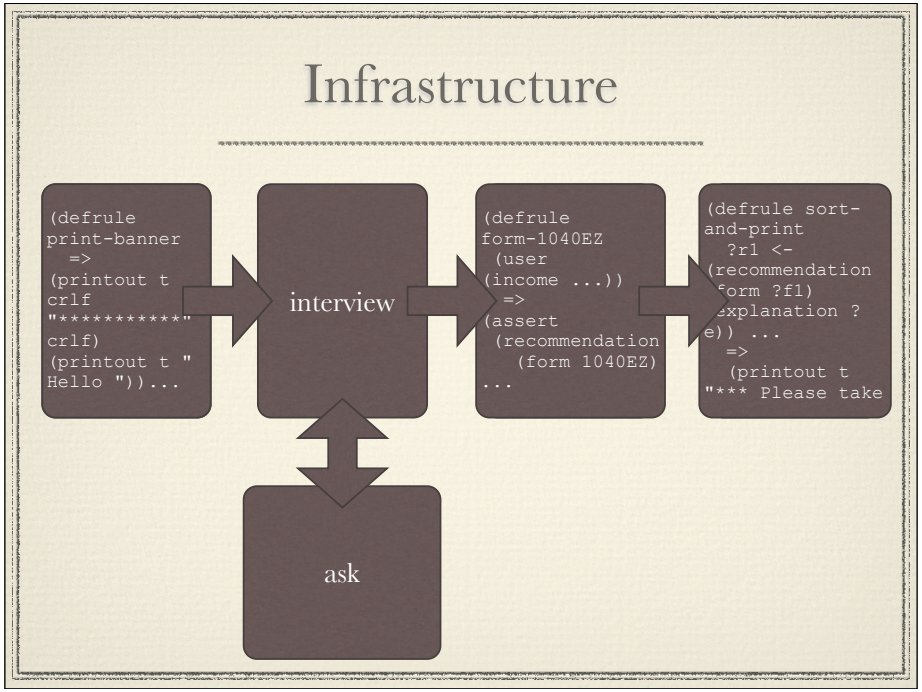


We'll need some infrastructure to interact with the system.

Execution: welcome, ask questions, decide what forms you need, tell you what forms you need. This suggests the modules we want.

Plus, we'll want somewhere for the "if we have a question but no answer, ask the question" rule to live. => 'ask' module.

since all our facts (questions, user answers, generated recommendations) are required by multiple modules, they'll all live in MAIN.



We'll need some infrastructure to interact with the system.

Execution: welcome, ask questions, decide what forms you need, tell you what forms you need. This suggests the modules we want.

Plus, we'll want somewhere for the "if we have a question but no answer, ask the question" rule to live. => 'ask' module.

since all our facts (questions, user answers, generated recommendations) are required by multiple modules, they'll all live in MAIN.

The Questions

```
(deffacts question-data
  "The questions the system can ask."
  (question (ident income) (type number)
    (text "What was your annual income?"))
  (question (ident interest) (type yes-no)
    (text "Did you earn more than $400 of taxable
    interest?"))
  (question (ident dependents) (type number)
    (text "How many dependents live with you?"))

...etc
```


We'll define the questions as a list of question facts, with unique identifiers, expected answer types, and their text. Then the 'ask' module can do some basic type checking on your answer.

The “ask” module

```
(defrule ask::ask-question-by-id
  "Given the identifier of a question, ask it and assert the answer"
  (declare (auto-focus TRUE))
  ?ask <- (MAIN::ask ?id)
  (MAIN::question (ident ?id) (text ?text)
  (type ?type))
  (not (MAIN::answer (ident ?id)))
 =>
  (bind ?answer (ask-user ?text ?type))
  (assert (answer (ident ?id) (text ?answer)))
  (retract ?ask)
  (return))
```


Here’s the ask a question rule.

The “interview” module

```
(defrule request-income
=>
  (assert (ask income)))

(defrule assert-user-fact
  (answer (ident income) (text ?i))
  (answer (ident dependents) (text ?d))
=>
  (assert (user (income ?i) (dependents ?d))))
```

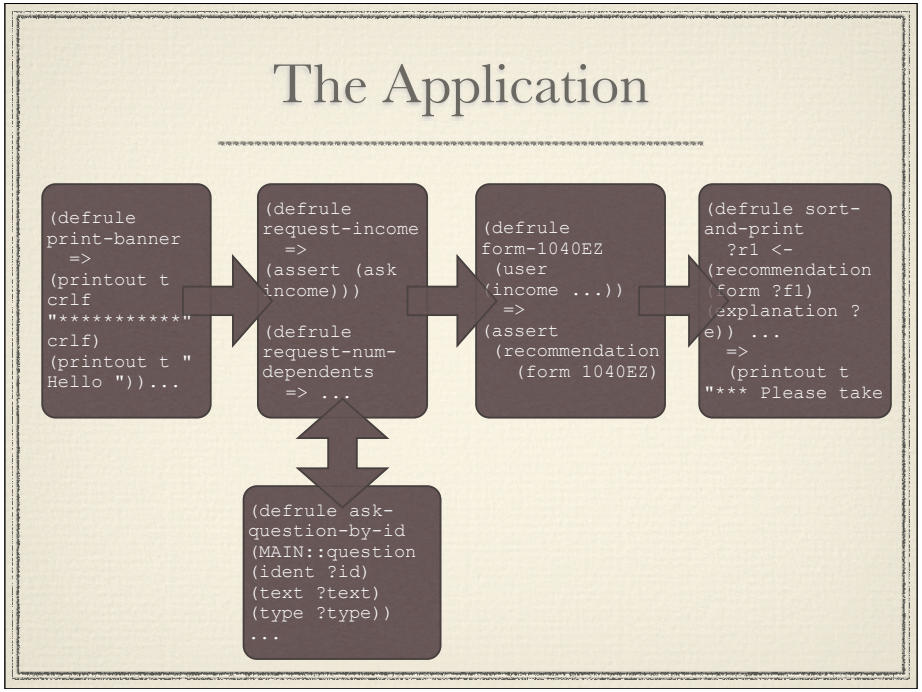

Most of the rules in the “interview” module have no LHS, so they can always be fired.

This second rule is special -> it asserts a ‘user’ fact once we know the user’s income and number of dependents.

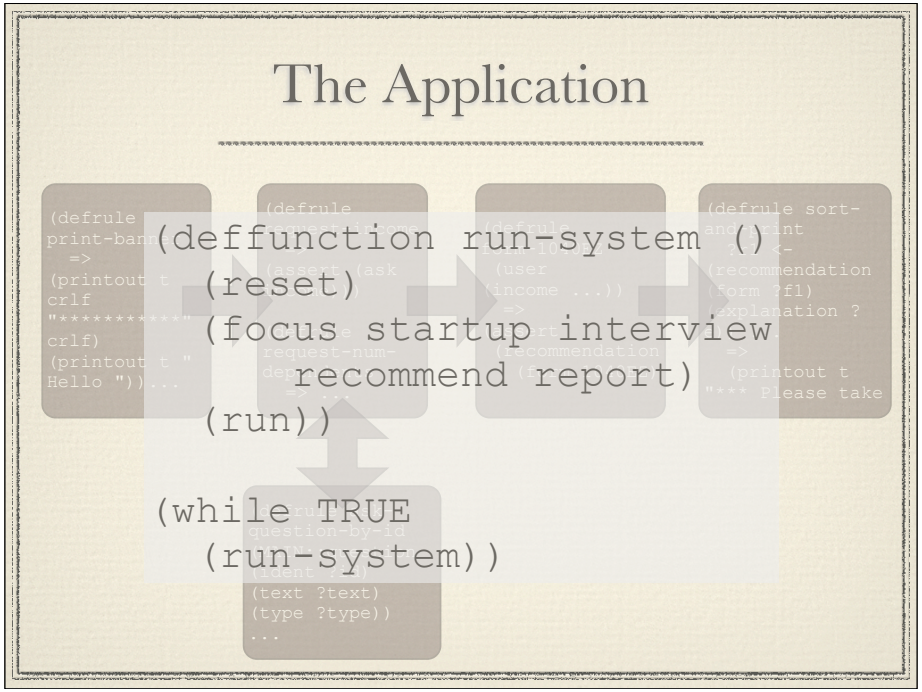
The “interview” module

```
(defrule request-childcare-expenses
  ;; If the user has dependents
  (answer (ident dependents) (text ?t&:(> ?t
0)))
  =>
  (assert (ask childcare)))
```


Some of the questions depend on other questions, so we have rules like this.



So now we have everything, we can run it.



So now we have everything, we can run it.