



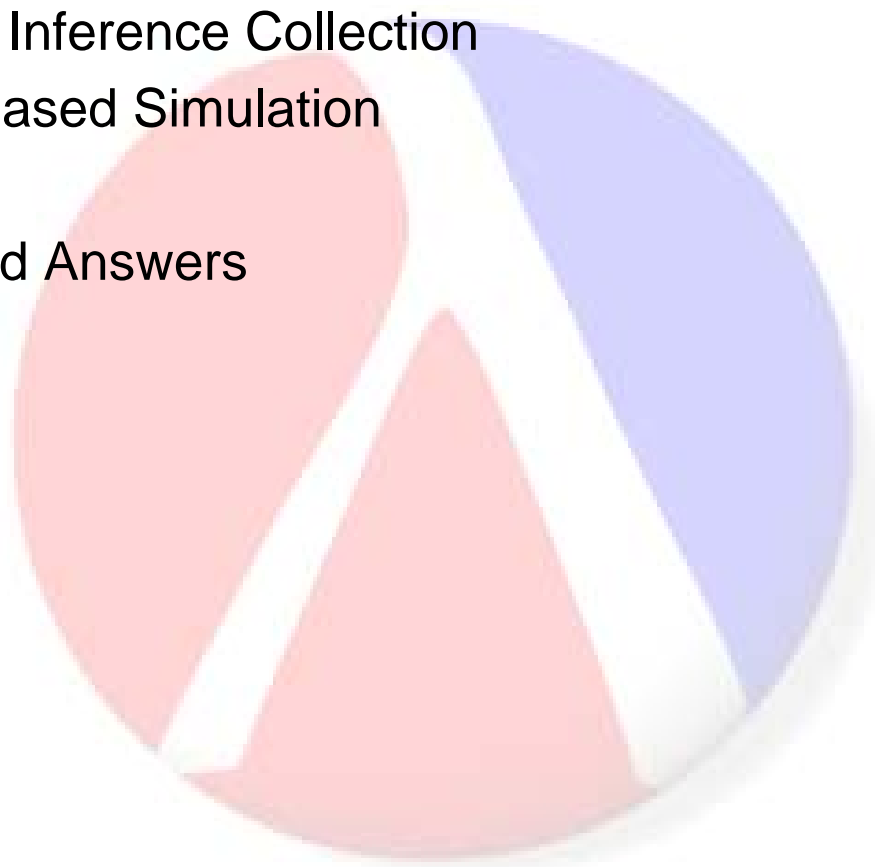
# **PLT Scheme Inference Collection**

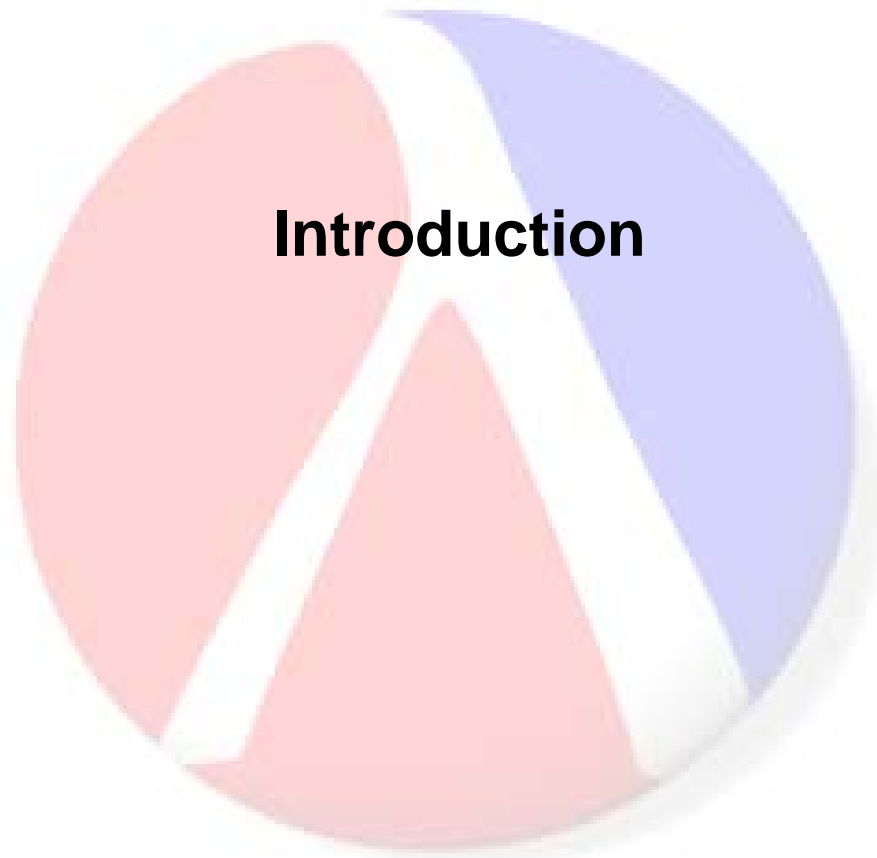
Dr. Doug Williams  
m.douglas.williams@saic.com  
October 26, 2006

---



- Introduction
- PLT Scheme Inference Collection
- Knowledge-Based Simulation
- Future Plans
- Questions and Answers
- Workshop







- Recreate a previously available knowledge-based simulation environment capability
  - Previously implemented in Symbolics Common LISP
  - Re-implement in PLT Scheme
    - Availability – free download ([www.drscheme.org](http://www.drscheme.org))
    - Portability – Windows, Linux, UNIX, Mac OS X
- Extend previous work
  - Provide a better mathematical framework
  - Implement a process-based simulation engine
  - Support combined discrete-event and continuous simulations
  - Implement an efficient rule-based inference engine
  - Support data-driven (forward chaining) and goal-driven (backward chaining) inferencing
- Provide a framework for implementing advanced knowledge-based simulations



**PLT Scheme**  
Science Collection

- **PLT Scheme Science Collection**
  - Provides the mathematical and analysis framework
  - Previously part of the simulation collection, but provides functionality that is useful outside of simulations
  - Inspired by the GNU Scientific Library (GSL)



**PLT Scheme**  
Simulation Collection

- **PLT Scheme Simulation Collection**
  - Provides a process-based, discrete-event simulation engine with automatic data collection
  - Supports combined discrete and continuous simulations
  - Designed to facilitate component-based simulation models



**PLT Scheme**  
Inference Collection

- **PLT Scheme Inference Collection**
  - Provides an efficient rule-based inference engine
  - Support both forward chaining (data-driven) and backward chaining (goal-driven) inferencing
  - (To be) Integrated with the simulation collection, i.e. inferencing can be done on simulation objects



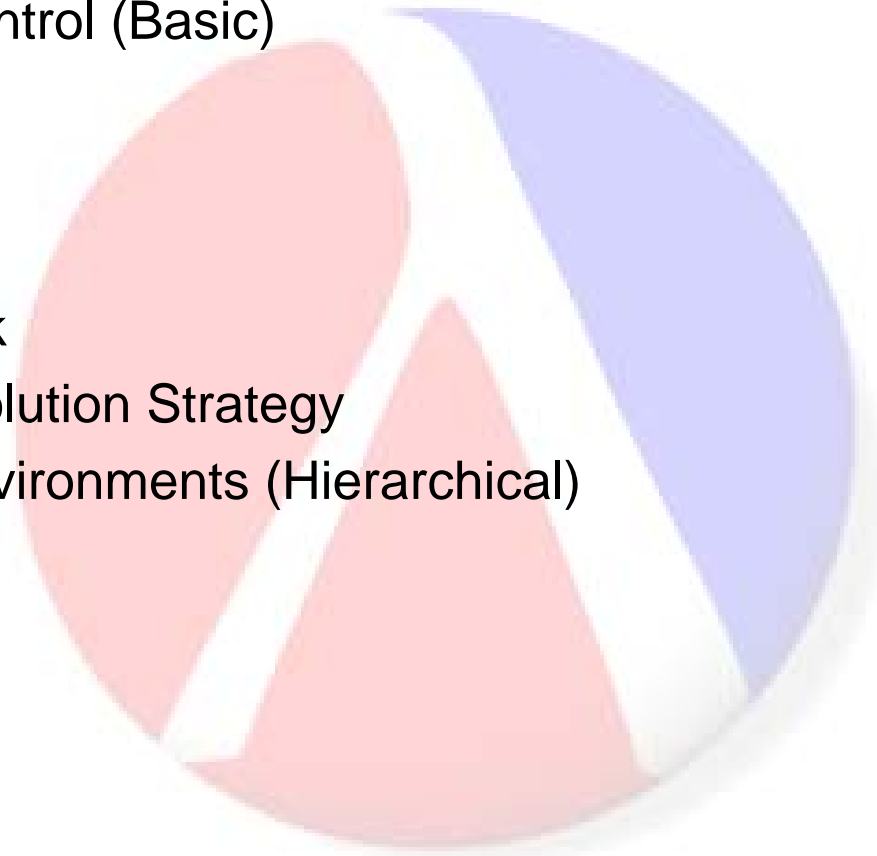
- Machine Constants
- Mathematical Constants and Functions
- Special Functions
- Random Number Generation
- Random Distributions
- Statistics
- Histograms
- Ordinary Differential Equations (Version 2.0)
- Chebyshev Approximations



- Simulation Environments (Basic)
- Simulation Control (Basic)
- Events
- Processes
- Resources
- Data Collection
- Sets
- Continuous Simulation Models
- Simulation Classes
- Simulation Control (Advanced)
- Simulation Environments (Hierarchical)
- Components



- Inference Environments (Basic)
- Inference Control (Basic)
- Assertions
- Rule Sets
- Rules
- Rule Network
- Conflict Resolution Strategy
- Inference Environments (Hierarchical)





- Development of all three collections is being moved to the Schematics project at SourceForge.
- Latest versions require PLT Scheme V301 or later
- PLT Scheme Science Collection
  - Version 2.4 available via PLaneT
  - Reference Manual included in Help Desk
- PLT Scheme Simulation Collection
  - Version 2.0 available via PLaneT
  - Reference Manual included in Help Desk
- PLT Scheme Inference Collection
  - Version 1.2 available via PLaneT



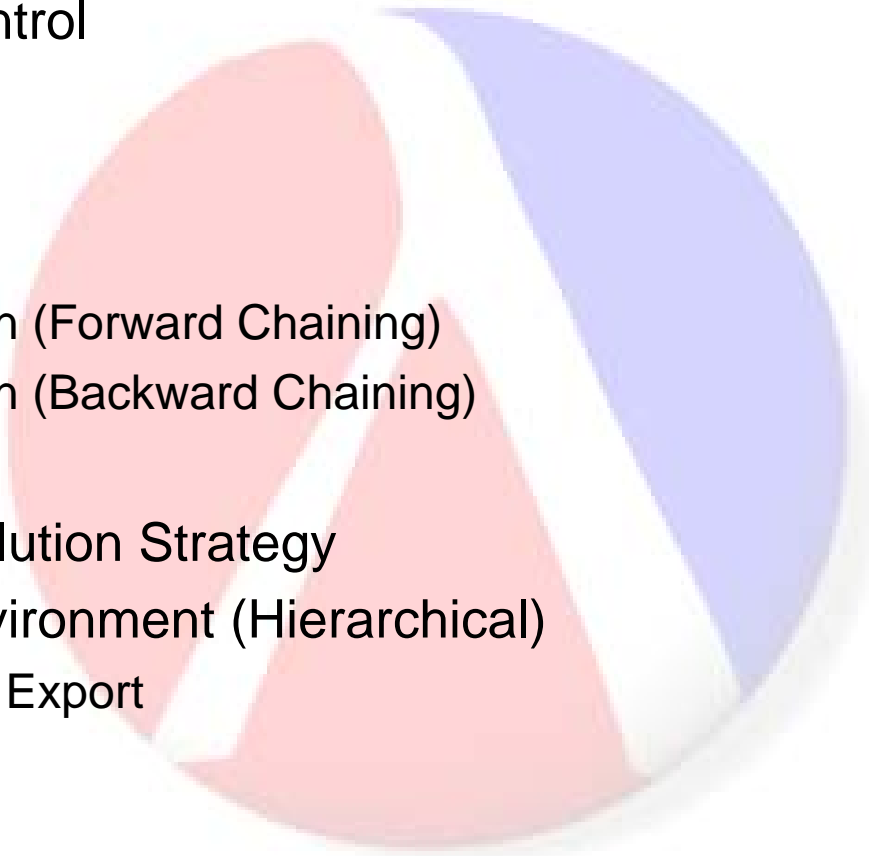
- **PLT Scheme Science Collection**
  - Release 1.0 October 2004 ✓
  - Release 2.0 December 2005 ✓
    - Ordinary Differential Equations (ODEs)
  - Release 2.1 February 2006 ✓
    - Additional CDFs
    - Beta, Incomplete Gamma, and Exponential Integral Special Functions
    - Additional ODE steppers
    - Bug fixes
  - Release 2.4 June 2006 ✓
    - Help Desk compatible (HTML) reference manual
- **PLT Scheme Simulation Collection**
  - Release 1.0 December 2005 ✓
  - Release 1.1 February 2006 ✓
    - Fixes to make models compatible with modules
    - Add linked events and renegeing
  - Release 2.0 June 2006 ✓
    - Help Desk compatible (HTML) reference manual
  - Release 3.0 December 2006
- **PLT Scheme Inference Collection**
  - Release 1.0 July 2006 ✓
  - Release 1.3 October 2006
  - Release 2.0 December 2006



# PLT Scheme Inference Collection



- Inference Environments (Basic)
- Inference Control
- Assertions
- Rule Sets
- Rules
  - Data-Driven (Forward Chaining)
  - Goal-Driven (Backward Chaining)
- Rule Network
- Conflict Resolution Strategy
- Inference Environment (Hierarchical)
  - Import and Export





## Inference Environments (Basic)

---

- An inference environment encapsulates the state of an inference
  - Rule network (nodes, data index, goal index, rule node list)
  - Next assertion id and assertion index
  - Exit continuation
  - Agenda
  - Current rule, trace, and conflict resolution strategy
- Multiple inference environments may exist at the same time
  - Multiple, independent (or cooperating) inferences may exist as part of a larger system.
  - Note that this is different than hierarchical inference environments, which are discussed later.
  - It is also different than assumption processing, which performs state-state searches by cloning inference environments.



- Fields in an inference environment:
  - `data-index` index into data match nodes
  - `goal-index` index into goal match nodes
  - `rule-nodes` list of rule nodes
  - `exit` continuation to exit the inference loop
  - `next-assertion-id` next (sequential) assertion id
  - `assertion-index` index into assertions
  - `trace` current trace state
  - `agenda` ordered list of rule instances
  - `rule` currently executing rule
  - `strategy` current conflict resolution strategy



- The parameter `current-inference-environment` represents the current inference environment.
  - Defaults to `default-inference-environment`
- Routines are provided to get and set fields in the `current-inference-environment`.
  - `(current-inference-data-index [hash-table?])`
  - `(current-inference-goal-index [hash-table?])`
  - `(current-inference-rule-nodes [list?])`
  - `(current-inference-exit [continuation?])`
  - `(current-inference-next-assertion-id [integer?])`
  - `(current-inference-assertion-index [hash-table?])`
  - `(current-inference-trace [boolean?])`
  - `(current-inference-agenda [list?])`
  - `(current-inference-rule [rule-instance?])`
  - `(current-inference-strategy [symbol?])`



- The `with-inference-environment` macro evaluates its body with `current-inference-environment` set to the specified inference environment.
  - `(with-inference-environment inference-environment body ...)`
- The `with-new-inference-environment` macro evaluates its body with `current-inference-environment` set to a new inference environment.
  - `(with-new-inference-environment body ...)`



- The `assert` procedure creates a new assertion and passes it into the rule network in the current inference environment.
  - `(assert fact reason)`
  - `(assert fact)`
- The `retract` procedure passes the retraction of an assertion into the rule network in the current inference environment.
  - `(retract assertion)`
- The `replace` procedure replaces (the fact stored in) an assertion and passes the replacement into the rule network in the current inference environment. [In effect it retracts the assertion and asserts the replacement.]
  - `(replace assertion fact reason)`
  - `(replace assertion fact)`



- The `modify` procedure modifies the fields in the fact stored in an assertion. The modified assertion is passed into the rule network in the current inference environment. [This is only applicable to association list and object facts.]
- The `with-fact-modifications` macro allows arbitrary modifications to the fact stored in an assertion. The modified assertion is passed into the rule network in the current inference environment.
- The `check` procedure checks a goal (or hypothesis). It initiates a goal-directed (backward chaining) inference.
  - `(check fact) -> match?`
- The `query` procedure returns a list of the assertions matching a pattern.
  - `(query pattern) -> list?`



- The `start-inference` procedure implements the data-driven (forward chaining) inference loop. It asserts the fact (`start`) and then continually executes rules from the agenda until there are no more rule instances to fire or if the loop is explicitly exited. [Note that the data-driven (forward chaining) inference loop will invoke goal-driven (backward chaining) as needed.]
  - `(start-inference)`
- The `stop-inference` procedure exists the current inference loop, optionally returning a value.
  - `(stop-inference return-value)`
  - `(stop-inference)`



- An assertion represents a fact to be used by the inference engine.
- Currently, an assertion has the following fields:
  - `fact` the fact represented by the assertion
  - `reason` rule instance or boolean?
- Assertions created using the `assert` procedure are passed into the rule network in the current inference environment and invoke data-driven inferencing.
- Assertions created using the `check` procedure are passed into the rule network in the current inference environment and invoke goal-driven inferencing.
  - Note that goal match nodes invoke the `check` procedure as needed
- Assertions matching a binding clause can be captured using an assertion variable in a pattern clause.
- In the future, confidences will be added to assertions.



- A rule set is a collection of rules that together solve some (portion of a) problem.
- A rule set is defined using the `define-ruleset` macro.
  - `(define-ruleset name)`
- A rule set must be activated in an inference environment to participate in inferencing. Activation creates the rule network for the rules in the rule set.
- The `activate` procedure activates a rule set in the current inference environment.
  - `(activate ruleset)`
- Multiple rule sets may be active in a single inference environment.
- There is currently no way to deactivate a rule set.



- A rule is a piece of problem solving knowledge. Rules are applied as directed by the inference engine.
- Every rule belongs to a rule set.
- Rules may be either data-directed (forward chaining) or goal-directed (backward chaining).
- A rule is defined using the `define-rule` macro:
  - ```
(define-rule (name ruleset [#:priority priority])  
  [clause ... ; goal clause(s)  
  <==]  
  [clause ... ; data clause(s)  
  ==>]  
  [expr ...] ; rule action(s)
```
- If any goal clauses are provided, the rule is goal-directed and invoked via a backward chaining control strategy. Otherwise, the rule is data-directed and invoked via a forward chaining control strategy.



- Rules that do not include any goal clauses are data-directed (forward chaining) rules.
- Data-directed rules must include at least one data clause and at least one action expression.
- As facts are asserted or retracted they are matched against the data clauses and (for data-directed rules) propagated through the rule network. This results in data-directed rule instances being added or removed from the agenda.
- Rule instances on the agenda are ordered by priority.
- The conflict resolution strategy determines the order of rule instances with equal priorities.
- The rule network maintains the state information on matches and joins.



- Rules that include at least one goal clause are goal-directed (backward chaining) rules.
- Goal-directed rules do not require any data clauses.
  - Useful as the base case for some goal-driven inferencing  

```
(define-rule (some-rule some-ruleset)  
  (ancestor ?name ?name))
```
  - You can also specify facts using the rule network  

```
(define-rule (some-rule some-ruleset)  
  (ancestor milton doug))
```
- Goal-directed rules may have action expressions.
- Goals (or hypotheses) are matched against goal clauses and propagated through the rule network. If propagated through all of the data clauses in a rule, the rule is executed and the goal is affirmed.
- The control strategy maintains the state information on matches and joins.



- I don't like using the terms forward chaining and backward chaining because they tend to connote a modality within the inference engine that doesn't really exist.
- Although a specific rule is considered either as data-directed or goal-directed based in the presence (or absence) of a goal clause, any rule may include data-directed or goal directed behavior. In particular:
  - Any rule, including goal-directed rules, may have clauses that match data and imply a data-directed behavior.
  - Any rule, included data-directed rules, may have clauses that match goals and imply a goal-directed behavior.
- The inference loop dynamically invokes a data-oriented or a goal-oriented control strategy based on the pattern clauses it encounters.
- At the highest level, `start-simulation` initiates execution of rule instances places on the agenda via the data-directed control strategy while `check` initiates a goal-directed control strategy for a specific goal (or hypothesis).



- A pattern clause specifies a goal or data pattern.
- Each clause contains a pattern and optionally either an assertion variable or an existential modifier.
- The forms of a clause are:
  - (`<assertion variable> <- <pattern>`)
  - (`<existential modifier> <pattern>`)
  - `<pattern>`
- An assertion variable is a variable, a symbol beginning with a `?` (e.g. `?var`), that is bound to the assertion matching the pattern.
- An existential modifier is one of the following:
  - `no` (or `notany`)      there are no matching assertions
  - `any`                      there is at least one matching assertion
  - `notall`                    there is at least one non-matching assertion
  - `all`                        there are no non-matching assertions



# Binding Pattern Clauses

- A binding pattern clause matches a pattern against asserted facts by binding variables against matching fact elements.
- The first occurrence of a variable in the (goal or data) clauses of the rule binds the value and subsequent occurrences must match that binding.
- Binding pattern clauses may include constraints that must be true for a match to propagate to subsequent clauses.
  - Match constraints are constraints that do not rely on bindings from previous clauses. For efficiency, match constraints are evaluated when a fact is asserted.
  - Join constraints are constraints that rely on bindings from previous clauses. Join constraints are evaluated when matches are joined between clauses.



- An existential pattern clause matches the presence or absence of pattern matches.
- While existential clauses may contain variables (or wildcards) – in fact they don't make much sense without them – they do not result in bindings for those variables.
- Existential clauses may not include an assertion variable – since they match just the presence or absence of matches, not the matches themselves.
- Existential modifiers:
  - `no` (or `notany`)      there are no matching assertions
  - `any`                            there is at least one matching assertion
  - `notall`                        there is at least one non-matching assertion
  - `all`                             there are no non-matching assertions
- The universe of potential matches for `all` (or `notall`) is all assertions where the first element of the fact matches the first element of the pattern.



- A pattern is a data structure (e.g. list or vector) that matches against facts stored in assertions.
- A pattern may include variables that become bound to corresponding elements in matched assertions.
- The initial element of a pattern must be a symbol.
- The other elements of a pattern may be:
  - a wildcard symbol, `?`, that matches (without binding) any fact element
  - a variable, a symbol beginning with `?` (e.g. `?var`), that is bound to a matching fact element
  - an atom (e.g. symbol, number, string) that will match an equivalent (i.e. `equiv?`) fact element
  - a list whose first element is a variable and the second element is a constraint expression (e.g. `(?n (> ?n 0))`)
  - a pair whose `car` is a symbol (naming a field) and whose `cdr` is a pattern element, an association



- List patterns match list facts.
- List patterns match pattern elements to fact elements by position.
- A list pattern may match variable length list facts by using a variable (or wildcard) in the tail position (i.e. an improper list).
- Examples:

```
- (define-rule (print-ancestors ancestors-ruleset)
  (?request <- (request ?name))
  (parents ?name ?mother ?father)
  ==>
  ...)

- (define-rule (rule-2 towers-rules)
  (no (move . ?)
  (ring ?size on (?peg (not (eq? ?peg `right))))
  (no (ring (?size-1 (> ?size-1 ?size))
        on (?peg-1 (not (eq? ?peg-1 `right))))))
  ==>
  ...)
```



```
(require (planet "inference.ss" ("williams" "inference.plt"))))

(define-ruleset ancestors-ruleset)

(define-rule (initialize ancestors-ruleset)
  (?start <- (start))
  ==>
  (retract ?start)
  (printf "Please enter the first name of a~n")
  (printf "person whose ancestors you would~n")
  (printf "like to find:~n")
  (assert `(request ,(read))))

(define-rule (print-ancestors ancestors-ruleset)
  (?request <- (request ?name))
  (parents ?name ?mother ?father)
  ==>
  (retract ?request)
  (when ?mother
    (printf "~a is an ancestor via ~a~n" ?mother ?name)
    (assert `(request ,?mother)))
  (when ?father
    (printf "~a is an ancestor via ~a~n" ?father ?name)
    (assert `(request ,?father))))

(define-rule (remove-request ancestors-ruleset #:priority -100)
  (?request <- (request ?))
  ==>
  (retract ?request))
```



## Ancestors – List Patterns (cont'd)

```
(define (find-ancestors)
  (with-new-inference-environment
    (activate ancestors-ruleset)
    (assert '(parents penelope jessica jeremy))
    (assert '(parents jessica mary-elizabeth homer))
    (assert '(parents jeremy jenny steven))
    (assert '(parents steven loree john))
    (assert '(parents loree #f jason))
    (assert '(parents homer stephanie #f))
    (start-inference)))
```

```
>(find-ancestors)
Please enter the first name of a
person whose ancestors you would
like to find:
penelope
jessica is an ancestor via penelope
jeremy is an ancestor via penelope
jenny is an ancestor via jeremy
steven is an ancestor via jeremy
loree is an ancestor via steven
john is an ancestor via steven
jason is an ancestor via loree
mary-elizabeth is an ancestor via jessica
homer is an ancestor via jessica
stephanie is an ancestor via homer
>
```



- Association list patterns match association list facts.
- Keys in an association list pattern may be in any order. The fact association list is searched for the key.
- As with any pattern (or fact), an association list pattern must begin with a symbol. Also, there may be any number of positional pattern elements prior to the association list.

- Examples:

```
- (define-rule (print-ancestors ancestors-ruleset)
  (?request <- (request ?name))
  (parents ?name (mother . ?mother) (father . ?father)))
==>
...)
```

```
- (define-rule (rule-2 towers-rules)
  (no (move . ?)
  (ring ?size (on ?peg (not (eq? ?peg `right))))
  (no (ring (?size-1 (> ?size-1 ?size))
  (on ?peg-1 (not (eq? ?peg-1 `right))))))
==>
...)
```



# Ancestors – Association List Patterns

```
(require (planet "inference.ss" ("williams" "inference.plt")))  
  
(define-ruleset ancestors-ruleset)  
  
(define-rule (initialize ancestors-ruleset)  
  (?start <- (start))  
  ==>  
  (retract ?start)  
  (printf "Please enter the first name of a~n")  
  (printf "person whose ancestors you would~n")  
  (printf "like to find:~n")  
  (assert `(request ,(read))))  
  
(define-rule (print-ancestors ancestors-ruleset)  
  (?request <- (request ?name))  
  (parents ?name (mother . ?mother) (father . ?father))  
  ==>  
  (retract ?request)  
  (when ?mother  
    (printf "~a is an ancestor via ~a~n" ?mother ?name)  
    (assert `(request ,?mother)))  
  (when ?father  
    (printf "~a is an ancestor via ~a~n" ?father ?name)  
    (assert `(request ,?father))))  
  
(define-rule (remove-request ancestors-ruleset #:priority -100)  
  (?request <- (request ?))  
  ==>  
  (retract ?request))
```



```
(define (find-ancestors)
  (with-new-inference-environment
    (activate ancestors-ruleset)
    (assert '(parents penelope (mother . jessica) (father . jeremy)))
    (assert '(parents jessica (mother . mary-elizabeth) (father . homer)))
    (assert '(parents jeremy (mother . jenny) (father . steven)))
    (assert '(parents steven (mother . loree) (father . john)))
    (assert '(parents loree (mother . #f) (father . jason)))
    (assert '(parents homer (mother . stephanie) (father . #f)))
    (start-inference)))
```

```
>(find-ancestors)
Please enter the first name of a
person whose ancestors you would
like to find:
penelope
jessica is an ancestor via penelope
jeremy is an ancestor via penelope
jenny is an ancestor via jeremy
steven is an ancestor via jeremy
loree is an ancestor via steven
john is an ancestor via steven
jason is an ancestor via loree
mary-elizabeth is an ancestor via jessica
homer is an ancestor via jessica
stephanie is an ancestor via homer
>
```



- Vector patterns match vector facts.
- Vector patterns match pattern elements to fact elements by position.
- Examples:

```
- (define-rule (print-ancestors ancestors-ruleset)
  (?request <- #(request ?name))
  #(parents ?name ?mother ?father)
  ==>
  ...)

- (define-rule (rule-2 towers-rules)
  (no (move . ?)
  #(ring ?size on (?peg (not (eq? ?peg 'right))))
  (no #(ring (?size-1 (> ?size-1 ?size))
  on (?peg-1 (not (eq? ?peg-1 'right))))))
  ==>
  ...)
```



```
(require (planet "inference.ss" ("williams" "inference.plt")))

(define-ruleset ancestors-ruleset)

(define-rule (initialize ancestors-ruleset)
  (?start <- (start))
  ==>
  (retract ?start)
  (printf "Please enter the first name of a~n")
  (printf "person whose ancestors you would~n")
  (printf "like to find:~n")
  (assert `#(request ,(read))))

(define-rule (print-ancestors ancestors-ruleset)
  (?request <- #(request ?name))
  #(parents ?name ?mother ?father)
  ==>
  (retract ?request)
  (when ?mother
    (printf "~a is an ancestor via ~a~n" ?mother ?name)
    (assert `#(request ,?mother)))
  (when ?father
    (printf "~a is an ancestor via ~a~n" ?father ?name)
    (assert `#(request ,?father))))

(define-rule (remove-request ancestors-ruleset #:priority -100)
  (?request <- #(request ?))
  ==>
  (retract ?request))
```



## Ancestors – Vector Patterns (cont'd)

```
(define (find-ancestors)
  (with-new-inference-environment
    (activate ancestors-ruleset)
    (assert `#(parents penelope jessica jeremy))
    (assert `#(parents jessica mary-elizabeth homer))
    (assert `#(parents jeremy jenny steven))
    (assert `#(parents steven loree john))
    (assert `#(parents loree #f jason))
    (assert `#(parents homer stephanie #f))
    (start-inference)))
```

```
>(find-ancestors)
Please enter the first name of a
person whose ancestors you would
like to find:
penelope
jessica is an ancestor via penelope
jeremy is an ancestor via penelope
jenny is an ancestor via jeremy
steven is an ancestor via jeremy
loree is an ancestor via steven
john is an ancestor via steven
jason is an ancestor via loree
mary-elizabeth is an ancestor via jessica
homer is an ancestor via jessica
stephanie is an ancestor via homer
>
```



- Structure patterns match structure facts.
- Structure patterns are vector patterns where the first element is a structure identifier.
- Examples:
  - ```
(define-struct parents (name mother father)
                      (make-inspector))
(define-rule (print-ancestors ancestors-ruleset)
  (?request <- #(request ?name))
  #(struct:parents ?name ?mother ?father)
=>
  ...)
```
  - ```
(define-rule (rule-2 towers-rules)
  (no (move . ?)
  #(struct:ring ?size on (?peg (not (eq? ?peg 'right))))
  (no #(struct:ring (?size-1 (> ?size-1 ?size))
      on (?peg-1 (not (eq? ?peg-1 'right)))))
=>
  ...)
```



```
(require (planet "inference.ss" ("williams" "inference.plt")))

(define-struct parents (name mother father) (make-inspector))

(define-ruleset ancestors-ruleset)

(define-rule (initialize ancestors-ruleset)
  (?start <- (start))
  ==>
  (retract ?start)
  (printf "Please enter the first name of a~n")
  (printf "person whose ancestors you would~n")
  (printf "like to find:~n")
  (assert `#(request ,(read))))

(define-rule (print-ancestors ancestors-ruleset)
  (?request <- #(request ?name))
  #(struct:parents ?name ?mother ?father)
  ==>
  (retract ?request)
  (when ?mother
    (printf "~a is an ancestor via ~a~n" ?mother ?name)
    (assert `#(request ,?mother)))
  (when ?father
    (printf "~a is an ancestor via ~a~n" ?father ?name)
    (assert `#(request ,?father))))

(define-rule (remove-request ancestors-ruleset #:priority -100)
  (?request <- #(request ?))
  ==>
  (retract ?request))
```



```
(define (find-ancestors)
  (with-new-inference-environment
    (activate ancestors-ruleset)
    (assert (make-parents `penelope `jessica `jeremy))
    (assert (make-parents `jessica `mary-elizabeth `homer))
    (assert (make-parents `jeremy `jenny `steven))
    (assert (make-parents `steven `loree `john))
    (assert (make-parents `loree #f `jason))
    (assert (make-parents `homer `stephanie #f))
    (start-inference)))
```

```
>(find-ancestors)
Please enter the first name of a
person whose ancestors you would
like to find:
penelope
jessica is an ancestor via penelope
jeremy is an ancestor via penelope
jenny is an ancestor via jeremy
steven is an ancestor via jeremy
loree is an ancestor via steven
john is an ancestor via steven
jason is an ancestor via loree
mary-elizabeth is an ancestor via jessica
homer is an ancestor via jessica
stephanie is an ancestor via homer
>
```



# Conflict Resolution Strategy

- When a rule instance is added to the agenda, it is placed after all rule instances of higher priority and before all rule instances of lower priority.
- When a rule instance is added to the agenda, for placement among rule instances of equal priority, the conflict resolution strategy determines the order.
- Conflict resolution strategies:
  - depth                      depth first
  - breadth                     breadth first
  - order                        rule order
  - simplicity                 simplest first
  - complex                     most complex first
  - random                      random
- The conflict resolution strategy is stored in the current inference environment and set or retrieved using:
  - `(current-inference-strategy [symbol?])`



- The depth first conflict resolution strategy adds new rule instances before any existing rule instances with equal priority.
- No information about the rules themselves is used in conflict resolution.
- This favors new rule instances in selecting the appropriate rule instance to execute.
- This is the default conflict resolution strategy.
- Depth first conflict resolution is specified by:
  - `(current-inference-strategy 'depth)`



```
(define (find-ancestors)
  (with-new-inference-environment
    (current-inference-strategy `depth)
    (activate ancestors-ruleset)
    (assert '(parents penelope jessica jeremy))
    (assert '(parents jessica mary-elizabeth homer))
    (assert '(parents jeremy jenny steven))
    (assert '(parents steven loree john))
    (assert '(parents loree #f jason))
    (assert '(parents homer stephanie #f))
    (start-inference)))
```

```
>(find-ancestors)
Please enter the first name of a
person whose ancestors you would
like to find:
penelope
jessica is an ancestor via penelope
jeremy is an ancestor via penelope
jenny is an ancestor via jeremy
steven is an ancestor via jeremy
loree is an ancestor via steven
john is an ancestor via steven
jason is an ancestor via loree
mary-elizabeth is an ancestor via jessica
homer is an ancestor via jessica
stephanie is an ancestor via homer
>
```



- The breadth first conflict resolution strategy adds new rule instances after any existing rule instances with equal priority.
- No information about the rules themselves is used in conflict resolution.
- This favors existing rule instances in selecting the appropriate rule instance to execute.
- Breadth first conflict resolution is specified by:
  - `(current-inference-strategy `breadth)`



## Breadth First – Example

```
(define (find-ancestors)
  (with-new-inference-environment
    (current-inference-strategy 'breadth)
    (activate ancestors-ruleset)
    (assert '(parents penelope jessica jeremy))
    (assert '(parents jessica mary-elizabeth homer))
    (assert '(parents jeremy jenny steven))
    (assert '(parents steven loree john))
    (assert '(parents loree #f jason))
    (assert '(parents homer stephanie #f))
    (start-inference)))
```

```
>(find-ancestors)
Please enter the first name of a
person whose ancestors you would
like to find:
penelope
jessica is an ancestor via penelope
jeremy is an ancestor via penelope
mary-elizabeth is an ancestor via jessica
homer is an ancestor via jessica
jenny is an ancestor via jeremy
steven is an ancestor via jeremy
stephanie is an ancestor via homer
loree is an ancestor via steven
john is an ancestor via steven
jason is an ancestor via loree
>
```



- The rule order conflict resolution strategy adds new rule instances after any rule instance that occur earlier in the rule set and before rule instances that occur later in the rule set.
- New rule instances are placed before instances of the same rule.
- This orders rule instances according to the order of the rules in the rule set.
- Rule order conflict resolution is specified by:
  - `(current-inference-strategy `order)`
- Note that for the order is calculated with a rule set. When there are multiple active rule sets, rule instances from the active rule sets are intermingled by rule order.



- Currently, rule instances are placed before other instances of the same rule. That is, like a depth first search.
- Should we have both `order-depth` and `order-breadth`?
- These would add rule instances either before, for `order-depth`, or after, for `order-breadth`, any existing rule instances of the same rule.
- Should we also use something like activation order as a secondary ordering?



```
(define (find-ancestors)
  (with-new-inference-environment
    (current-inference-strategy 'order)
    (activate ancestors-ruleset)
    (assert '(parents penelope jessica jeremy))
    (assert '(parents jessica mary-elizabeth homer))
    (assert '(parents jeremy jenny steven))
    (assert '(parents steven loree john))
    (assert '(parents loree #f jason))
    (assert '(parents homer stephanie #f))
    (start-inference)))
```

```
>(find-ancestors)
Please enter the first name of a
person whose ancestors you would
like to find:
penelope
jessica is an ancestor via penelope
jeremy is an ancestor via penelope
mary-elizabeth is an ancestor via jessica
homer is an ancestor via jessica
jenny is an ancestor via jeremy
steven is an ancestor via jeremy
stephanie is an ancestor via homer
loree is an ancestor via steven
john is an ancestor via steven
jason is an ancestor via loree
>
```



- Rule specificity is a measure of how specific a rule is to a given situation.
  - Currently, rule specificity is measured by the number of preconditions in a rule.
  - In the future, this will probably be changed into the number of comparisons, which will give a finer granularity to the specificity measure.
- The simplicity and complexity conflict resolution strategies both use rule specificity to determine the order of rule instances with equal priorities.
  - The simplicity strategy favors simpler (i.e. less specific) rule instances.
  - The complexity strategy favors more complex (i.e. more specific) rule instances.



- The simplicity conflict resolution strategy adds new rule instances after any simpler rule instances and before any more complex rule instances.
- This favors simpler rule instances, in terms of specificity, in selecting the appropriate rule instance to execute.
- This orders rule instances by increasing specificity (i.e. lower specificity first)
- Simplicity conflict resolution is specified by:
  - `(current-inference-strategy 'simplicity)`



```
(define (find-ancestors)
  (with-new-inference-environment
    (current-inference-strategy `simplicity)
    (activate ancestors-ruleset)
    (assert '(parents penelope jessica jeremy))
    (assert '(parents jessica mary-elizabeth homer))
    (assert '(parents jeremy jenny steven))
    (assert '(parents steven loree john))
    (assert '(parents loree #f jason))
    (assert '(parents homer stephanie #f))
    (start-inference)))
```

```
>(find-ancestors)
Please enter the first name of a
person whose ancestors you would
like to find:
penelope
jessica is an ancestor via penelope
jeremy is an ancestor via penelope
mary-elizabeth is an ancestor via jessica
homer is an ancestor via jessica
jenny is an ancestor via jeremy
steven is an ancestor via jeremy
stephanie is an ancestor via homer
loree is an ancestor via steven
john is an ancestor via steven
jason is an ancestor via loree
>
```



- The complexity conflict resolution strategy adds new rule instances after any more complexity rule instances and before any simpler instances.
- This favors more complex rule instances, in terms of specificity, in selecting the appropriate rule instance to execute.
- This order rule instances by decreasing specificity (i.e. higher specificity first).
- Complexity conflict resolution is specified by:
  - `(current-inference-strategy `complexity)`

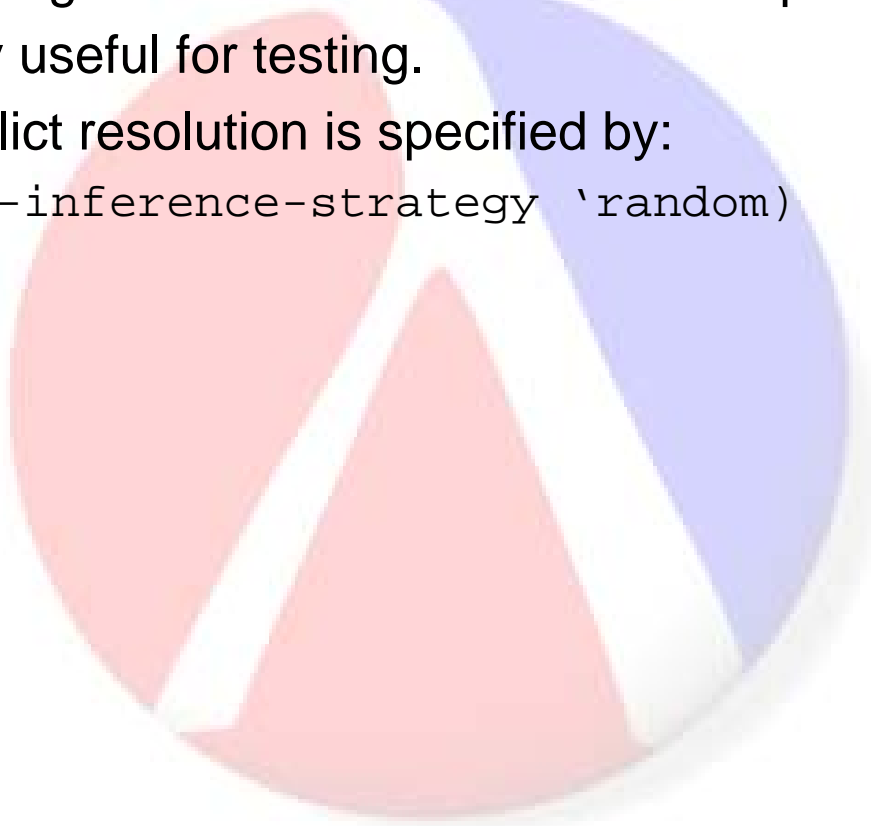


```
(define (find-ancestors)
  (with-new-inference-environment
    (current-inference-strategy 'complexity)
    (activate ancestors-ruleset)
    (assert '(parents penelope jessica jeremy))
    (assert '(parents jessica mary-elizabeth homer))
    (assert '(parents jeremy jenny steven))
    (assert '(parents steven loree john))
    (assert '(parents loree #f jason))
    (assert '(parents homer stephanie #f))
    (start-inference)))
```

```
>(find-ancestors)
Please enter the first name of a
person whose ancestors you would
like to find:
penelope
jessica is an ancestor via penelope
jeremy is an ancestor via penelope
mary-elizabeth is an ancestor via jessica
homer is an ancestor via jessica
jenny is an ancestor via jeremy
steven is an ancestor via jeremy
stephanie is an ancestor via homer
loree is an ancestor via steven
john is an ancestor via steven
jason is an ancestor via loree
>
```



- The random conflict resolution strategy adds rule instances randomly among rule instances with the same priority.
- This is mainly useful for testing.
- Random conflict resolution is specified by:
  - `(current-inference-strategy `random)`





```
(define (find-ancestors)
  (with-new-inference-environment
    (current-inference-strategy 'random)
    (activate ancestors-ruleset)
    (assert '(parents penelope jessica jeremy))
    (assert '(parents jessica mary-elizabeth homer))
    (assert '(parents jeremy jenny steven))
    (assert '(parents steven loree john))
    (assert '(parents loree #f jason))
    (assert '(parents homer stephanie #f))
    (start-inference)))
```

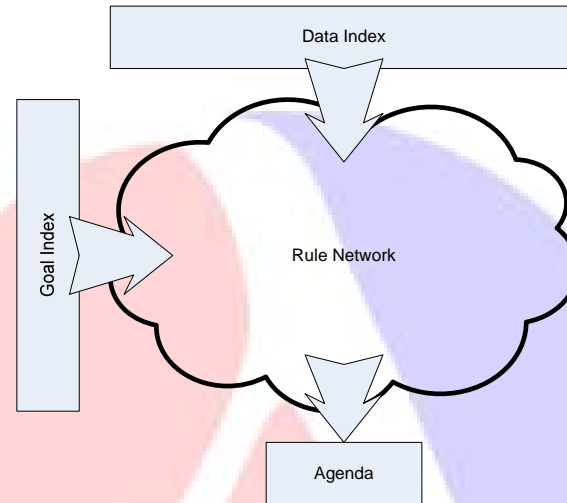
```
>(find-ancestors)
Please enter the first name of a
person whose ancestors you would
like to find:
penelope
jessica is an ancestor via penelope
jeremy is an ancestor via penelope
jenny is an ancestor via jeremy
steven is an ancestor via jeremy
loree is an ancestor via steven
john is an ancestor via steven
jason is an ancestor via loree
mary-elizabeth is an ancestor via jessica
homer is an ancestor via jessica
stephanie is an ancestor via homer
>
```



- For the ancestor rule set, what would happen if the `#:priority` option was removed from the `remove-request` rule?
  - Note that we define the correct behavior as printing all of the ancestors (in any particular order) with no duplicates.
- For each of the conflict resolution strategies, is the rule set correct using the specified strategy? Why or why not?
  - depth
  - breadth
  - order
  - simplicity
  - complexity
  - random



# Rule Networks – Indices and Agenda

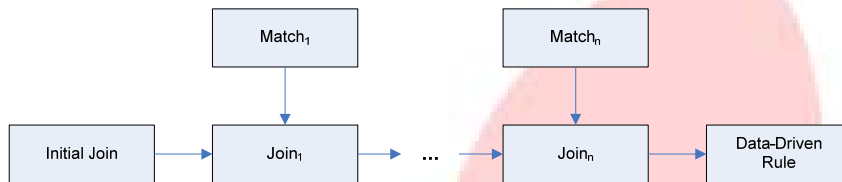


- There are two indices into the match nodes in the rule network:
  - The data index is an index into the data match nodes and is used for data-driven (forward chaining) inferencing.
  - The goal index is an index into the goal match nodes and is used for goal-driven (backward chaining) inferencing.
- Rule instances created (or deleted) during data-driven inferencing are maintained on the agenda.
- During goal-driven inferencing, rule instances are immediately executed rather than being added to the agenda.

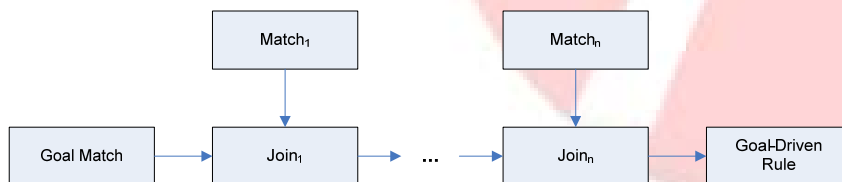


## Rule Networks – Nodes

### Data-Driven Rule



### Goal-Driven Rule



- The rule network is a collection of nodes that represent the rules in the active rule set(s).
- There is an initial node for each rule:
  - For data-driven rules, this is a join node with a single null match.
  - For goal-driven rules, this is a match node for the goal pattern.
- There is a match / join node pair for each data clause in the rule.
- There is a final rule node for each rule.
  - For goal-driven rules with no data clauses, the initial goal match node is connected to the rule node.



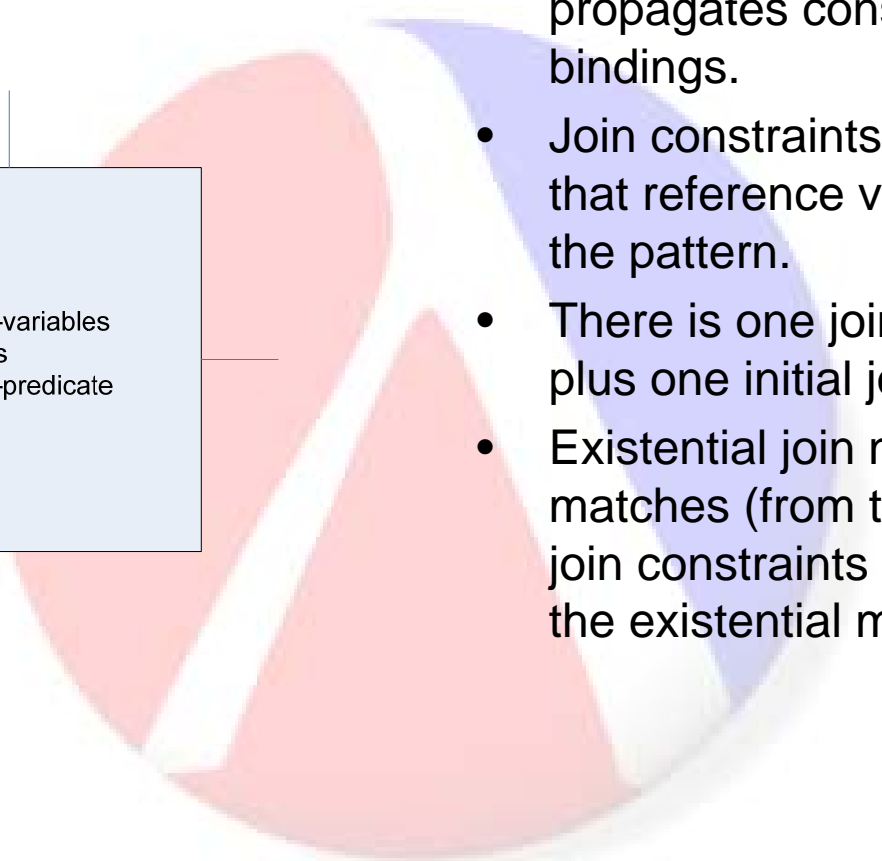
## Rule Networks – Match Nodes

successors  
matches  
assertion-variable  
pattern  
match-constraint-variables  
match-constraints  
match-constraints-predicate  
n

- Match nodes encapsulate the pattern for a clause.
- If the pattern matches a goal, this is a goal match node.
  - Matches are determined using backward chaining using `check`.
- Otherwise, it's a data goal node and matches are stored in the node.
- Match constraints are constraints that do not reference any variables outside the pattern.
- There is one match node per clause.



## Rule Networks – Join Nodes



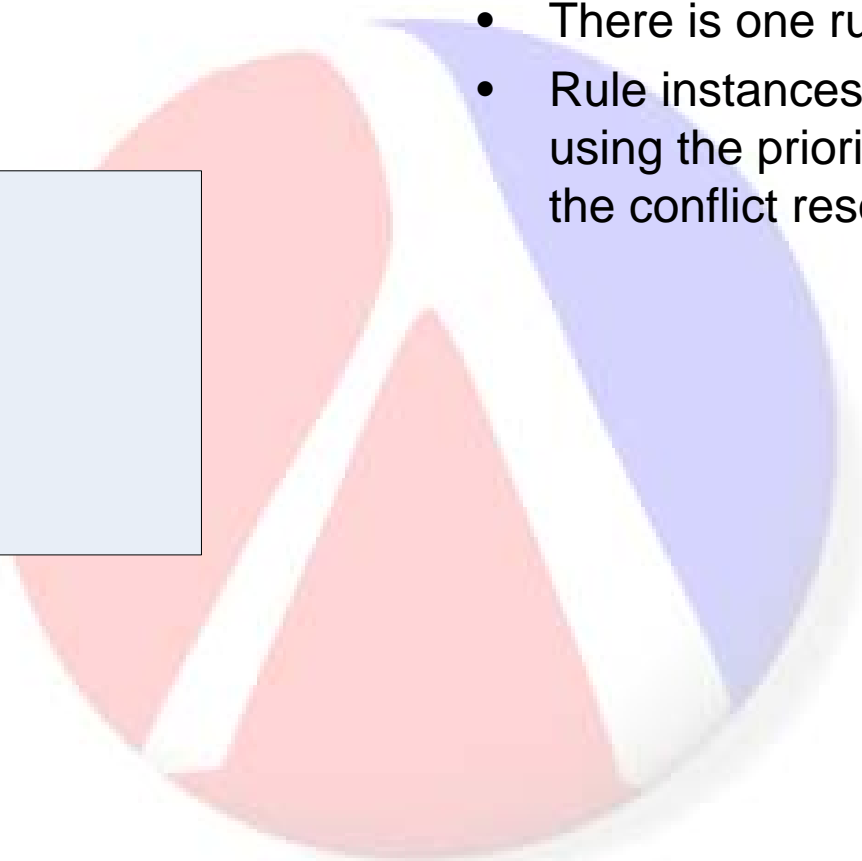
successors  
matches  
left  
right  
join-constraint-variables  
join-constraints  
join-constraint-predicate  
existential?  
match-counts

- Join nodes join matches and propagates consistent match bindings.
- Join constraints are constraints that reference variables outside the pattern.
- There is one join node per clause plus one initial join node.
- Existential join nodes propagate matches (from the left) when the join constraints are consistent with the existential modifier.



# Rule Networks – Rule Nodes

- Rule nodes represent rules.
- There is one rule node per rule.
- Rule instances are added agenda using the priority of the rule and the conflict resolution strategy.



successors  
matches  
rule  
join  
action



## Rule Networks – Rule Instances

- A rule instance represents a rule with its bindings that is eligible to fire (i.e. to be executed).
- There may be multiple instances (with different bindings) of the same rule eligible to fire at the same time.
- Rule instances are maintained in priority order on the agenda.
- The conflict resolution strategy determines the order of rule instances with the same priority.



- The inference main loop implements the forward chaining control strategy by continuously removing the next rule instance from the agenda and executing its actions. This continues until either the agenda is empty or a rule action explicitly exits the loop (by calling `stop-inference`).
- Backward chaining is either invoked directly using the `check` procedure or indirectly by a pattern that matches a goal clause in a goal directed rule.
- Addition and removal of rule instances to/from the agenda is done by the `assert` and `replace/modify` (or `with-fact-modifications`) procedures.



```
(require (planet "inference.ss" ("williams" "inference.plt")))

(define-ruleset simple-goal-test-rules)

(define-rule (rule-1 simple-goal-test-rules)
  (overworked ?x)
  <==
  (lecturing ?x)
  (marking-practicals ?x))

(define-rule (rule-2 simple-goal-test-rules)
  (lecturing alison)
  <==
  (month february))

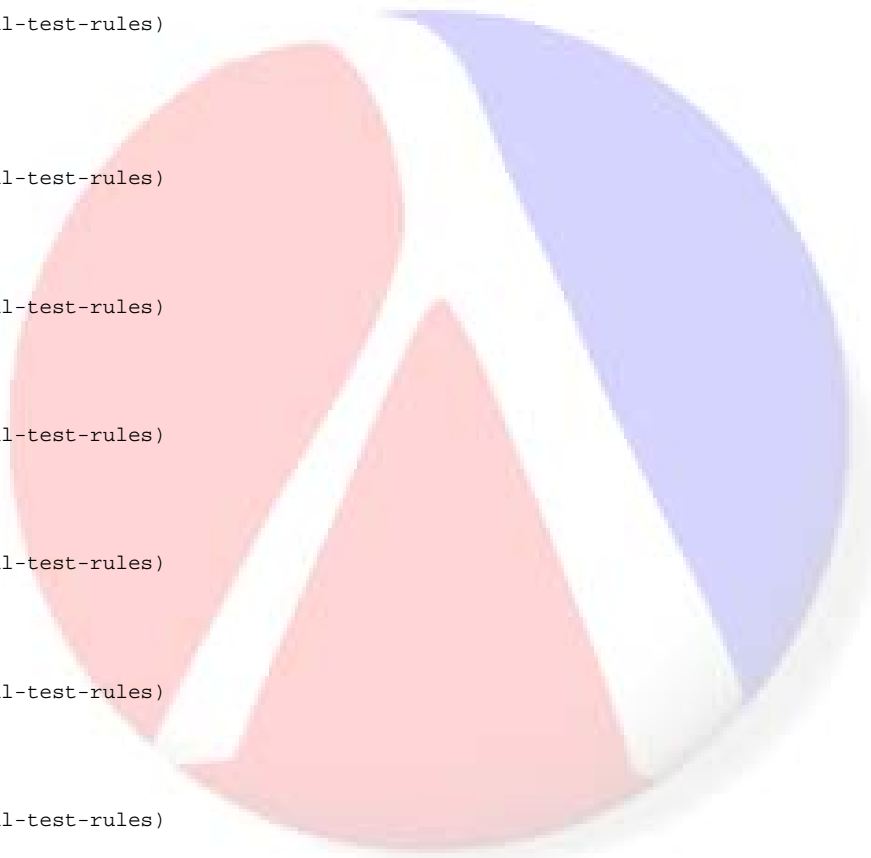
(define-rule (rule-3 simple-goal-test-rules)
  (marking-practicals ?x)
  <==
  (month february))

(define-rule (rule-4 simple-goal-test-rules)
  (bad-mood ?x)
  <==
  (overworked ?x))

(define-rule (rule-5 simple-goal-test-rules)
  (bad-mood ?x)
  <==
  (slept-badly ?x))

(define-rule (rule-6 simple-goal-test-rules)
  (weather cold)
  <==
  (month february))

(define-rule (rule-7 simple-goal-test-rules)
  (economy bad)
  <==
  (year 1993))
```





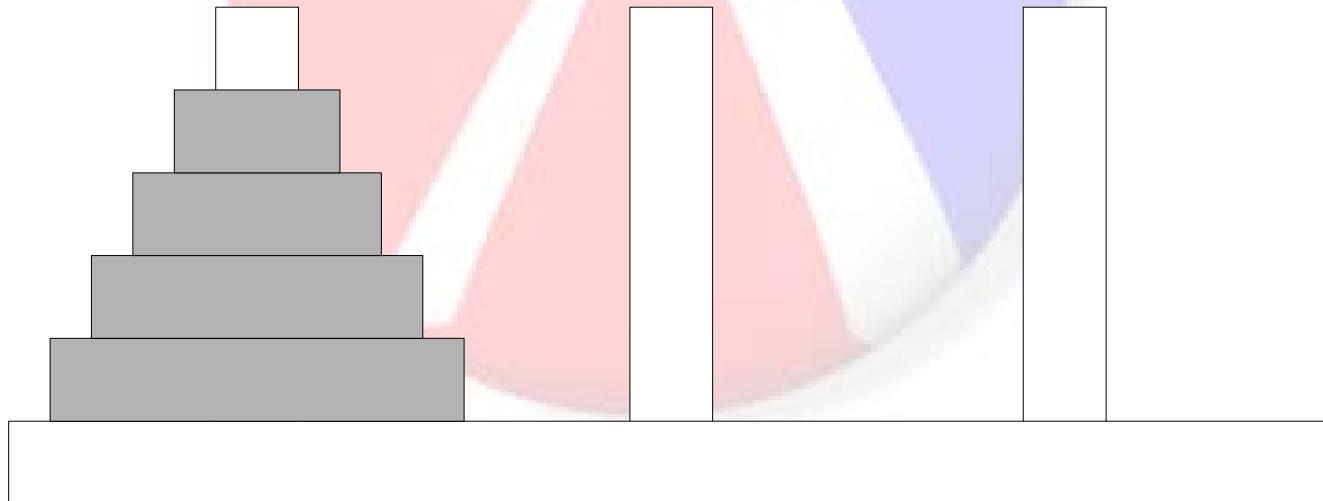
```
(define (solve-simple-goal-test)
  (with-new-inference-environment
    (activate simple-goal-test-rules)
    (current-inference-trace #t)
    (assert '(month february))
    (assert '(year 1993))
    (let ((result (check '(bad-mood alison))))
      (printf "(check '(bad-mood alison)) = ~a ~n" result)
      (if (not (null? result))
          (printf "Alison is in a bad mood.~n")
          (printf "Alison is not (known to be) in a bad mood.~n"))))))

>(solve-simple-goal-test)
>>> assertion-1: (month february)
>>> assertion-2: (year 1993)
?? assertion-3: (bad-mood alison)
?? assertion-4: (overworked alison)
?? assertion-5: (lecturing alison)
<== (rule-2 simple-goal-test-rules): ()
?? assertion-6: (marking-practicals alison)
<== (rule-3 simple-goal-test-rules): ((?x . alison))
<== (rule-1 simple-goal-test-rules): ((?x . alison))
<== (rule-4 simple-goal-test-rules): ((?x . alison))
(check '(bad-mood alison)) = (((assertion-3: (bad-mood alison)) (?x . alison)))
Alison is in a bad mood.
>
```



# Towers of Hanoi Example

- The rules of the game are:
  - (1) Move only one ring at a time, and
  - (2) Never put a larger ring on top of a smaller one,
- The objective is to transfer the entire pile from its starting peg to the target peg.
- Rules are from *Artificial Intelligence: Tools, Techniques, and Applications* by Tim O'Shea and Marc Eisenstadt.



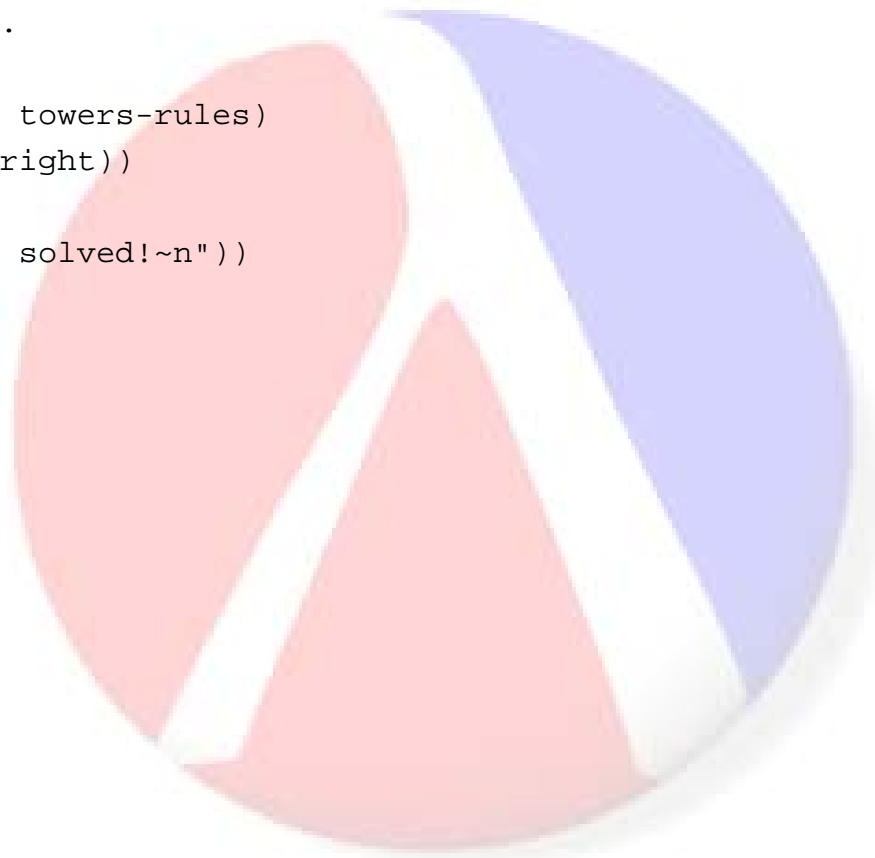


- If the target peg holds all of the rings 1 to n, stop because according to game rule (2) they must be in their original order and so the problem is solved.
- If there is no current goal – that is, if a ring has just been successfully moved, or if no rings have yet been moved – generate a goal. In this case the goal is to be that of moving to the target peg the largest ring that is not yet on the target peg.
- If there is a current goal, it can be achieved at once if there are no smaller rings on top of the ring to be moved (i.e., if the latter is at the top of its pile), and if there are no smaller rings on the peg to which it is to be moved (i.e., the ring to be moved is smaller than the top ring on the peg we intend to move it to). If this is the case, carry out the move and then delete the current goal so that rule 2 will apply next time.
- If there is a current goal but its disc cannot be moved as in rule 3, set up a new goal: that of moving the largest of the *obstructing* rings to the peg that is neither of those specified in the current goal (i.e., well out of the way of the current goal). Delete the current goal, so that rule 3 will apply to the new goal next time.



```
;; If the target peg hold all the rings 1 to n, stop because according  
;; to game rule (2) they must be in their original order and so the  
;; problem is solved.
```

```
(define-rule (rule-1 towers-rules)  
  (all (ring ? on right))  
  ==>  
  (printf "Problem solved!~n"))
```





```
;; If there is no current goal - that is, if a ring has just been  
;; successfully moved, or if no rings have yet to be moved - generate  
;; a goal. In this case the goal is to be that of moving to the  
;; target peg the largest ring that is not yet on the target peg.
```

```
(define-rule (rule-2 towers-rules)  
  (no (move . ?))  
  (ring ?size on (?peg (not (eq? ?peg 'right))))  
  (no (ring (?size-1 (> ?size-1 ?size))  
        on (?peg-1 (not (eq? ?peg-1 'right)))))  
=>  
  (assert `(move ,?size from ,?peg to right)))
```



```
;; If there is a current goal, it can be achieved at once if there is
;; no small rings on top of the ring to be moved (i.e. if the latter
;; is at the top of its pile), and there are no small rings on the
;; peg to which it is to be moved (i.e. the ring to be moved is
;; smaller than the top ring on the peg we intend to move it to). If
;; this is the case, carry out the move and then delete the current
;; goal so that rule 2 will apply next time.
```

```
(define-rule (rule-3 towers-rules)
  (?move <- (move ?size from ?from to ?to))
  (?ring <- (ring ?size on ?from))
  (no (ring (?size-1 (< ?size-1 ?size)) on ?from))
  (no (ring (?size-2 (< ?size-2 ?size)) on ?to))
  ==>
  (printf "Move ring ~a from ~a to ~a.\n" ?size ?from ?to)
  (replace ?ring `(ring ,?size on ,?to))
  (retract ?move))
```



```
;; If there is a current goal but its disc cannot be moved as in rule  
;; 3, set up a new goal: that of moving the largest of the obstructing  
;; rings to the peg that is neither of those specified in the current  
;; goal (i.e. well out of the way of the current goal). Delete the  
;; current goal, so that rule 2 will apply to the new goal next time.
```

```
(define-rule (rule-4 towers-rules)  
  (?move <- (move ?size from ?from to ?to))  
  (peg (?other (not (memq ?other (list ?from ?to)))))  
  (ring (?size-1 (< ?size-1 ?size))  
    on (?peg-1 (not (eq? ?peg-1 ?other))))  
  (no (ring (?size-2 (< ?size-1 ?size-2 ?size))  
    on (?peg-2 (not (eq? ?peg-2 ?other)))))  
=>  
  (replace ?move `(move ,?size-1 from ,?peg-1 to ,?other)))
```



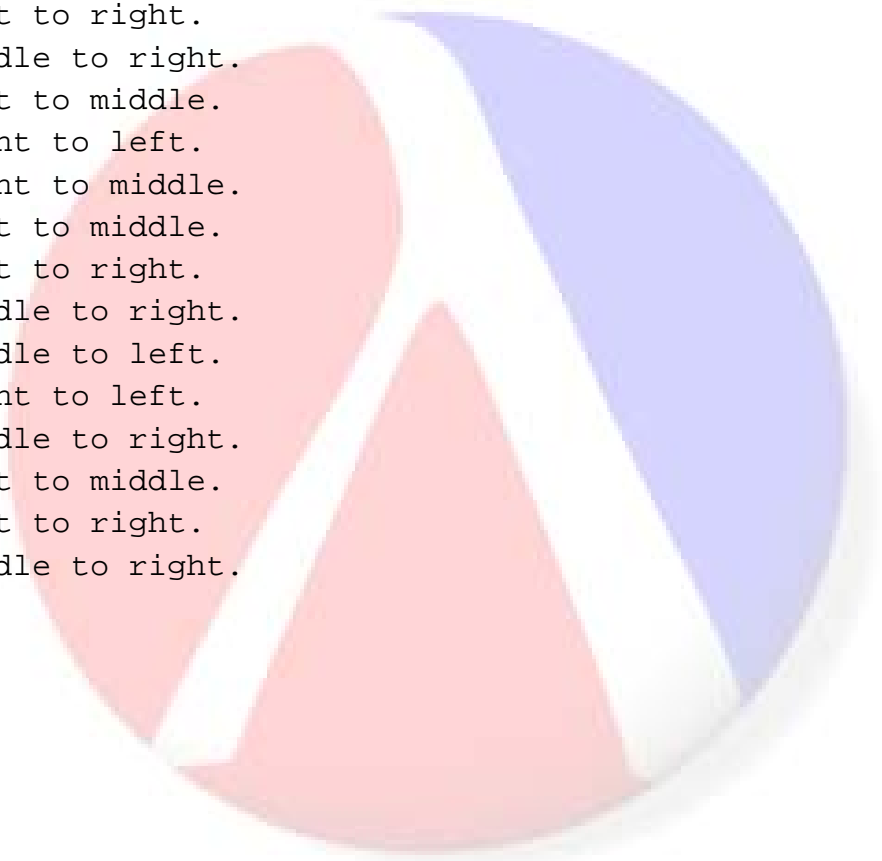
```
(define (solve-towers n)
  (with-new-inference-environment
    (activate towers-rules)
    ;(current-inference-trace #t)
    ;; Create pegs.
    (assert '(peg left))
    (assert '(peg middle))
    (assert '(peg right))
    ;; Create rings.
    (do ((i 1 (+ i 1)))
        ((> i n) (void))
        (assert `(ring ,i on left)))
    ;; Start inferencing.
    (start-inference)))
```





# Towers of Hanoi Output

```
>(solve-towers 4)
Move ring 1 from left to middle.
Move ring 2 from left to right.
Move ring 1 from middle to right.
Move ring 3 from left to middle.
Move ring 1 from right to left.
Move ring 2 from right to middle.
Move ring 1 from left to middle.
Move ring 4 from left to right.
Move ring 1 from middle to right.
Move ring 2 from middle to left.
Move ring 1 from right to left.
Move ring 3 from middle to right.
Move ring 1 from left to middle.
Move ring 2 from left to right.
Move ring 1 from middle to right.
Problem solved!
>
```





# Hierarchical Inference Environments

---

- Hierarchical inference environments have a parent – child relationship.
- A child inference environment is created by specifying the parent inference environment as the argument to `make-inference-environment`.
  - This is done automatically using the `with-new-child-inference-environment` macro.
- Assertions can be imported from a parent inference environment into a child inference environment or exported from a child inference environment to a parent inference environment.
  - `(import pattern)` – import assertions matching `pattern`
  - `(export pattern)` – export assertions matching `pattern`



# Sudoku Example

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
|   | 1 | 9 |   |   |   |   |   |   |
|   |   | 8 |   |   | 3 |   | 5 |   |
|   | 7 |   | 6 |   |   |   | 8 |   |
|   |   | 1 |   |   | 6 | 8 |   | 9 |
| 8 |   |   |   | 4 |   |   |   | 7 |
| 9 | 4 |   |   |   |   |   | 1 |   |
|   |   |   |   |   | 2 |   |   |   |
|   |   |   |   | 8 |   | 5 | 6 | 1 |
|   |   | 3 | 7 |   |   |   | 9 |   |

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 5 | 1 | 9 | 4 | 2 | 8 | 7 | 3 | 6 |
| 6 | 2 | 8 | 9 | 7 | 3 | 1 | 5 | 4 |
| 3 | 7 | 4 | 6 | 1 | 5 | 9 | 8 | 2 |
| 7 | 3 | 1 | 2 | 5 | 6 | 8 | 4 | 9 |
| 8 | 6 | 5 | 1 | 4 | 9 | 3 | 2 | 7 |
| 9 | 4 | 2 | 8 | 3 | 7 | 6 | 1 | 5 |
| 1 | 8 | 6 | 5 | 9 | 2 | 4 | 7 | 3 |
| 2 | 9 | 7 | 3 | 8 | 4 | 5 | 6 | 1 |
| 4 | 5 | 3 | 7 | 6 | 1 | 2 | 9 | 8 |

- Fill each row, column, and box with the digits 1 .. 9 such that there are no duplicates in any row, column, or box.
- Simple puzzles can be solved simply with logic.
- More complex puzzles – like the one above – may require ‘guessing’ at some point (i.e. search).

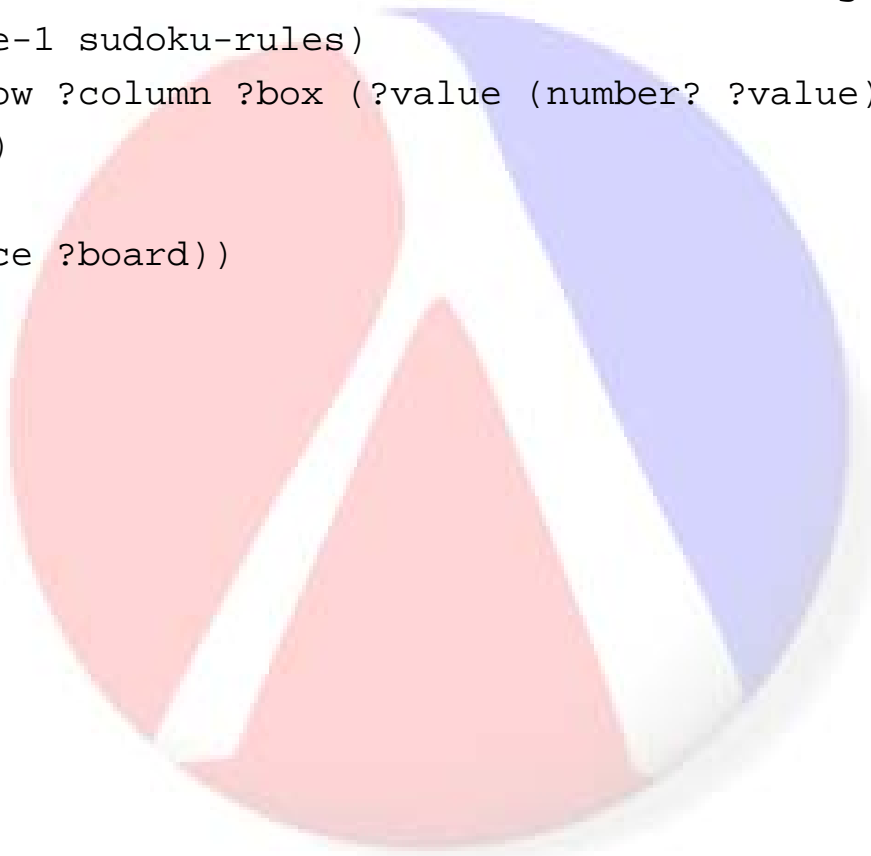


```
;; If there is a board and no cells, initialize the system.
;; For each row and column, make an assertion that is either the
;; value or all possible values.
(define-rule (initialize sudoku-rules)
  (board ?board)
  (no (cell . ?))
  ==>
  (printf "initialize:~n")
  (print-board ?board)
  (do ((row 0 (+ row 1)))
      ((= row 9) (void))
    (assert `(digit ,row))
    (do ((column 0 (+ column 1)))
        ((= column 9) (void))
      (let ((value (vector-ref (vector-ref ?board row) column))
            (box (+ (* (quotient row 3) 3)
                    (quotient column 3))))
        (if (eqv? value '_)
            (assert `(cell ,row ,column ,box (1 2 3 4 5 6 7 8 9)))
            (assert `(cell ,row ,column ,box ,value))))))
```



## Sudoku Rule 1 Code

```
;; If all of the cells are numbered, we've succeeded.  
;; Stop the inference and return the solution indicating success.  
(define-rule (rule-1 sudoku-rules)  
  (all (cell ?row ?column ?box (?value (number? ?value))))  
  (board ?board)  
=>  
  (stop-inference ?board))
```





```
;; If a cell has no possible values, we've failed.  
(define-rule (rule-2 sudoku-rules)  
  (cell ?row ?column ?box (?value (eq? ?value '())))  
  ==>  
  (fail))
```





```
;; If we have a single possible value in a cell,  
;; use it to number the cell.  
(define-rule (rule-3 sudoku-rules)  
  (?cell <- (cell ?row ?column ?box  
              (?value (and (list? ?value) (= (length ?value) 1))))  
  (board ?board)  
=>  
  (vector-set! (vector-ref ?board ?row) ?column (car ?value))  
  (replace ?cell `(cell ,?row ,?column ,?box ,(car ?value))))
```



```
;; If a cell is numbered and it conflicts with another numbered cell,  
;; fail.
```

```
(define-rule (rule-4a sudoku-rules)  
  (cell ?row ?column ?box (?value (number? ?value)))  
  (cell ?row (?column-1 (not (= ?column-1 ?column))) ?box-1  
    (?value-1 (and (number? ?value-1) (= ?value-1 ?value))))  
  ==>  
  (fail))
```

```
(define-rule (rule-4b sudoku-rules)  
  (cell ?row ?column ?box (?value (number? ?value)))  
  (cell (?row-1 (not (= ?row-1 ?row))) ?column ?box-1  
    (?value-1 (and (number? ?value-1) (= ?value-1 ?value))))  
  ==>  
  (fail))
```

```
(define-rule (rule-4c sudoku-rules)  
  (cell ?row ?column ?box (?value (number? ?value)))  
  (cell ?row-1 ?column-1 (?box (or (not (= ?row-1 ?row))  
    (not (= ?column-1 ?column))))  
    (?value-1 (and (number? ?value-1) (= ?value-1 ?value))))  
  ==>  
  (fail))
```



---

```
;; If a cell is numbered, remove that number from other cells in the  
;; same row, column, or box.
```

```
(define-rule (rule-5a sudoku-rules)  
  (cell ?row ?column ?box  
    (?value (number? ?value)))  
  (?cell-1 <- (cell ?row  
    (?column-1 (not (= ?column-1 ?column)))  
    ?box-1  
    (?value-1 (and (list? ?value-1)  
      (memv ?value ?value-1))))))
```

```
==>
```

```
(replace ?cell-1 `(cell ,?row ,?column-1 ,?box-1  
  ,(delete ?value ?value-1))))
```

```
(define-rule (rule-5b sudoku-rules)  
  (cell ?row ?column ?box  
    (?value (number? ?value)))  
  (?cell-1 <- (cell (?row-1 (not (= ?row-1 ?row)))  
    ?column  
    ?box-1  
    (?value-1 (and (list? ?value-1)  
      (memv ?value ?value-1))))))
```

```
==>
```

```
(replace ?cell-1 `(cell ,?row-1 ,?column ,?box-1  
  ,(delete ?value ?value-1))))
```



## Sudoku Rule 5 Code (cont'd)

```
(define-rule (rule-5c sudoku-rules)
  (cell ?row ?column ?box
    (?value (number? ?value)))
  (?cell-1 <- (cell ?row-1
    ?column-1
    (?box (or (not (= ?row-1 ?row))
      (not (= ?column-1 ?column))))
    (?value-1 (and (list? ?value-1)
      (memv ?value ?value-1))))))
==>
(replace ?cell-1 `(cell ,?row-1 ,?column-1 ,?box
  ,(delete ?value ?value-1))))
```



```
;; If there is a value that only occurs once as a possibility in any
;; row, column, or box, then make it the only possible value.
(define-rule (rule-6a sudoku-rules)
  (digit ?digit)
  (?cell <- (cell ?row ?column ?box (?value (and (list? ?value)
  (memv ?digit ?value))))))
  (no (cell ?row (?column-1 (not (= ?column-1 ?column))) ?
        (?value-1 (or (and (number? ?value-1)
                          (= ?value-1 ?digit))
                      (and (list? ?value-1)
                          (memv ?digit ?value-1)))))))
  ==>
  (replace ?cell `(cell ,?row ,?column ,?box ,(list ?digit))))

(define-rule (rule-6b sudoku-rules)
  (digit ?digit)
  (?cell <- (cell ?row ?column ?box (?value (and (list? ?value)
  (memv ?digit ?value))))))
  (no (cell (?row-1 (not (= ?row-1 ?row))) ?column ?
        (?value-1 (or (and (number? ?value-1)
                          (= ?value-1 ?digit))
                      (and (list? ?value-1)
                          (memv ?digit ?value-1)))))))
  ==>
  (replace ?cell `(cell ,?row ,?column ,?box ,(list ?digit))))
```



## Sudoku Rule 6 Code (cont'd)

```
(define-rule (rule-6c sudoku-rules)
  (digit ?digit)
  (?cell <- (cell ?row ?column ?box (?value (and (list? ?value)
  (memv ?digit ?value))))))
  (no (cell ?row-1 ?column-1 (?box (or (not (= ?row-1 ?row))
                                       (not (= ?column-1 ?column))))
      (?value-1 (or (and (number? ?value-1)
                        (= ?value-1 ?digit))
                    (and (list? ?value-1)
                        (memv ?digit ?value-1))))))
  ==>
  (replace ?cell `(cell ,?row ,?column ,?box ,(list ?digit))))
```

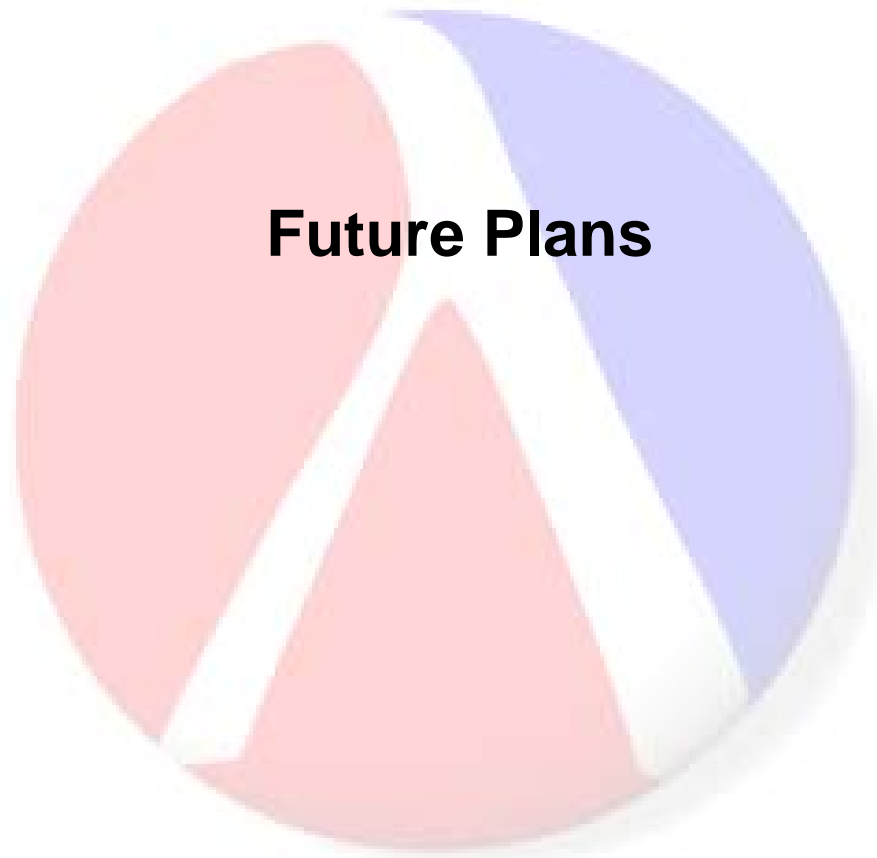


```
;; If the above rules don't find a solution (or fail), then create a
;; child inference to search using the shorted list of possibilities.
(define-rule (search sudoku-rules #:priority -100)
  (board ?board)
  (cell ?row ?column ?box (?value (and (list? ?value)
   (> (length ?value) 1))))
  (no (cell ? ? ? (?value-1 (and (list? ?value-1)
                                  (< (length ?value-1)
                                     (length ?value))))))
  ==>
  (printf "search: row = ~a, column = ~a, box = ~a, values = ~a~n"
         ?row ?column ?box ?value)
  (for-each
   (lambda (value)
     (let ((new-board (copy-board ?board)))
       (vector-set! (vector-ref new-board ?row) ?column value)
       (let ((result
              (with-new-child-inference-environment
               (activate sudoku-rules)
               (assert `(board ,new-board))
               (let ((result (start-inference)))
                 (printf "Rules fired = ~a~n" (current-inference-rules-fired))
                 result))))
         (when (vector? result)
           (stop-inference result))))
       ?value)
   (fail))
```



```
(define (sudoku-solver board)
  (printf "Initial Board~n")
  (print-board board)
  (with-new-inference-environment
   (activate sudoku-rules)
   (assert `(board ,board))
   (let ((result (start-inference)))
     (cond ((eq? result #:fail)
            (printf "Problem cannot be solved!~n"))
           ((not result)
            (printf "No solution found!~n")
            (let* ((board (cdr (assq '?board (cdar (query '(board ?board))))))
                  (print-board board)))
            (else
             (printf "Solution found!~n")
             (print-board result))))
           (printf "Rules fired = ~a~n" (current-inference-rules-fired))))))

(sudoku-solver '#(#(_ 1 9 _ _ _ _ _)
                  #(_ _ 8 _ _ 3 _ 5 _)
                  #(_ 7 _ 6 _ _ _ 8 _)
                  #(_ _ 1 _ _ 6 8 _ 9)
                  #(8 _ _ _ 4 _ _ _ 7)
                  #(9 4 _ _ _ _ _ 1 _)
                  #(_ _ _ _ _ 2 _ _)
                  #(_ _ _ _ 8 _ 5 6 1)
                  #(_ _ 3 7 _ _ _ 9 _)))
```





- Implement object pattern matching - similar to structure pattern matching with fields.
  - Fields are available using `object-info`
- Implement `modify` for association lists and objects.
- Implement assumption processing as an extension to hierarchical inference environment.
- Work on improving performance:
  - Index matches in rule network
  - Use hash tables for bindings
  - Benchmark performance against other rule-based engines
- Clean up rule syntax.



- Add regular expression matching for strings.
- Add (S)XML matching.
- Integrate with the simulation collection.
- Any ideas are welcome.





# Questions and Answers





- The purpose of the workshop is to help anyone interested in getting the PLT Scheme Inference Collection, the PLT Scheme Simulation Collection, and the PLT Scheme Science Collection up and running.
- Install PLT Scheme (also known as DrScheme)
  - The current versions of the science and simulation collections require PLT Scheme Version 301.
    - There are source code differences that preclude running them on Version 209.
    - We need Version 301 or later. Earlier 299/300 versions had a bug in PPlaneT that prevented proper compilation as well as a bug in the PLoT package that caused an error. (2D plots still error, but a fix is available.)
    - There are version for Windows (95 and up), Mac OS X, Mac OS Darwin, Linux (various flavors), UNIX (various flavors), or from source code.
  - Use the ‘Pretty Big (includes MrEd and Advanced)’ language option.
- Install the collections using the PPlaneT command line.



- Test the installation by running the examples.
  - PLT Scheme Science Collection examples
  - PLT Scheme Simulation Collection examples
  - PLT Scheme Inference Collection examples
- Reference manuals are available in the Help Desk.
  - The PLT Scheme Science Collection Reference Manual, Version 2.0 is complete.
  - The PLT Scheme Simulation Collection Reference Manual, Version 2.0 is complete.
- Workshop Examples



- M. Douglas Williams, PhD  
Sr. Scientist  
Science Applications International Corporation  
Denver, CO  
[m\\_douglas\\_williams@saic.com](mailto:m_douglas_williams@saic.com)  
(303) 229-0315

