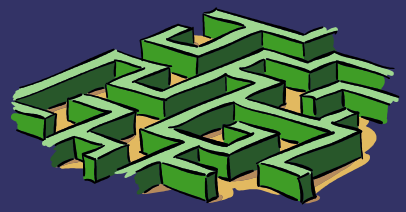
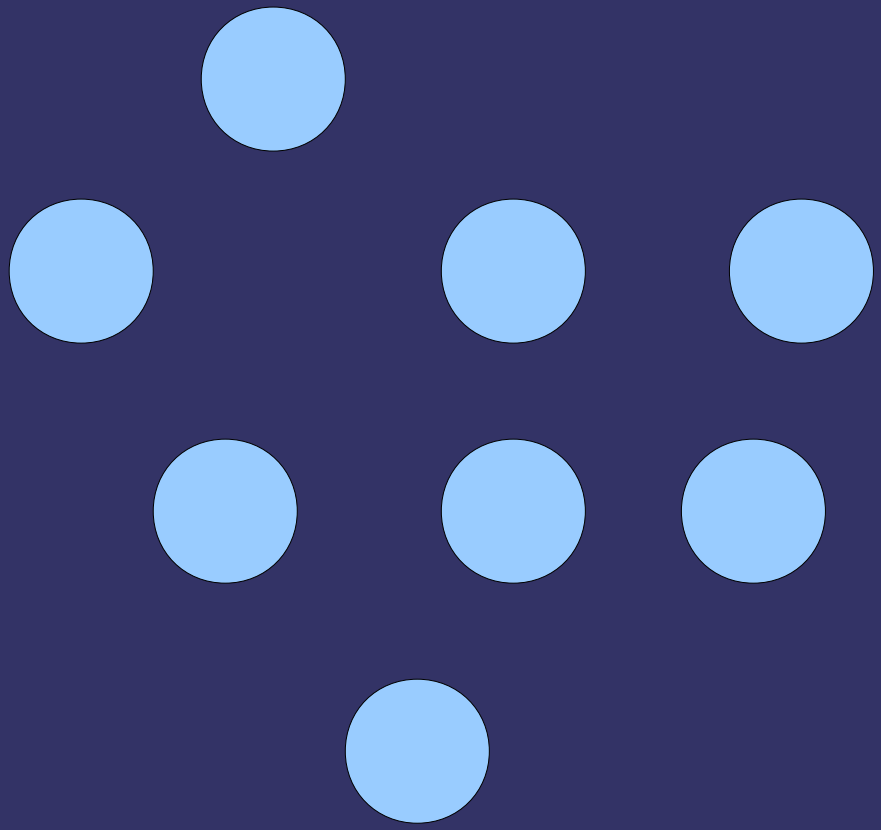
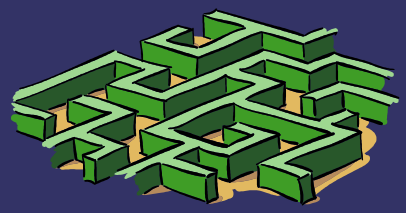
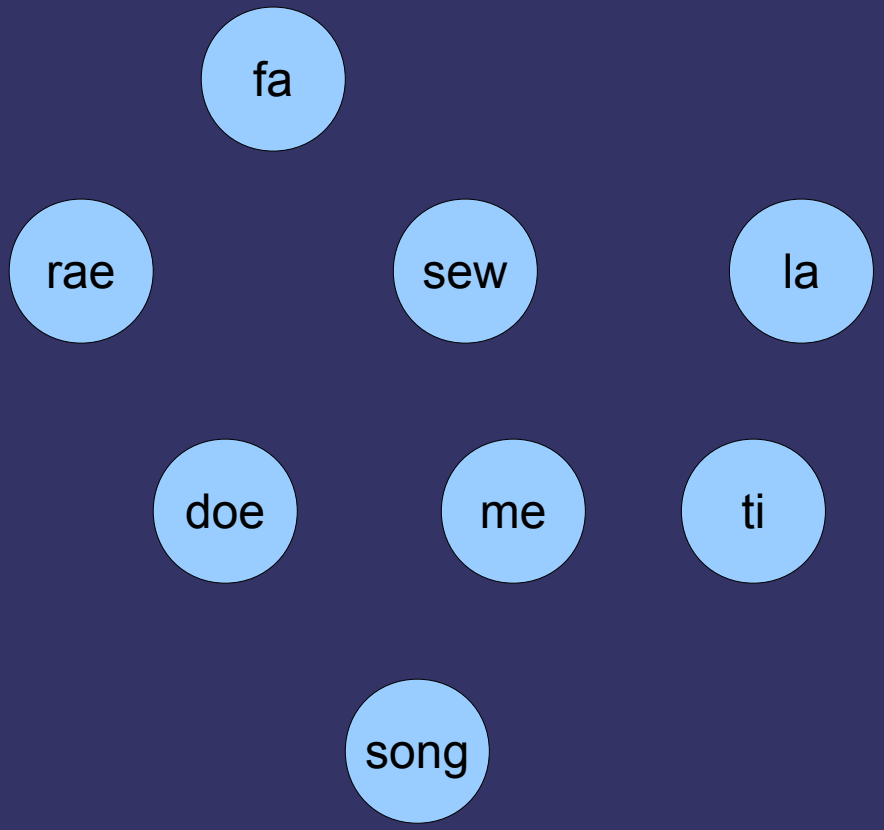
Logic Programming in Clojure



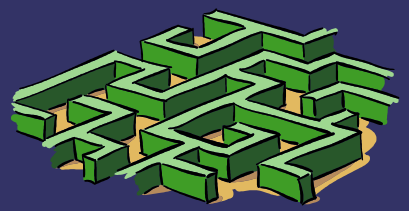


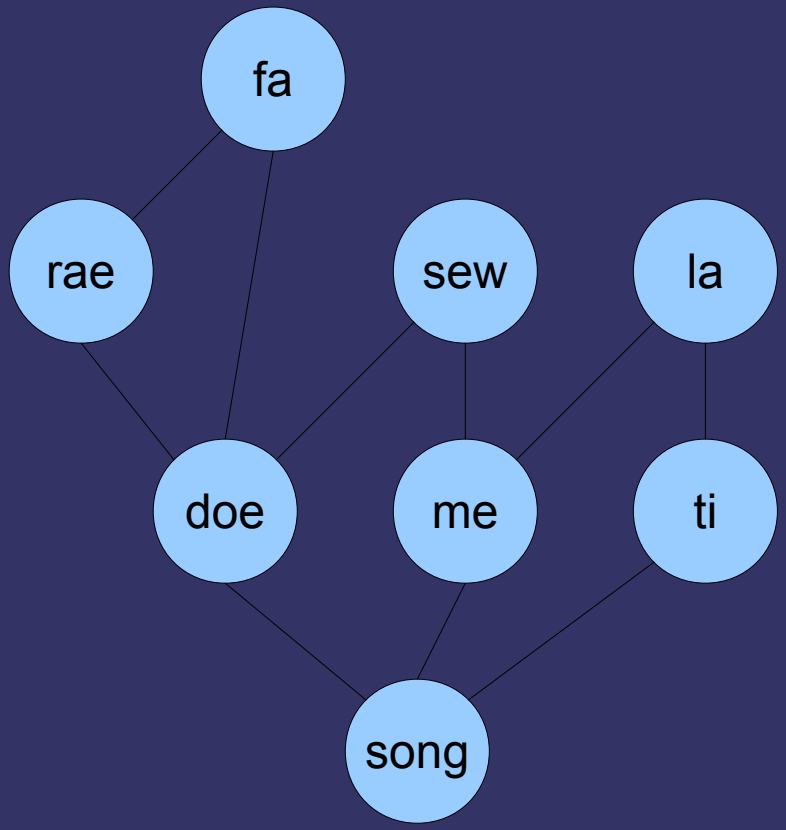
8 “things”



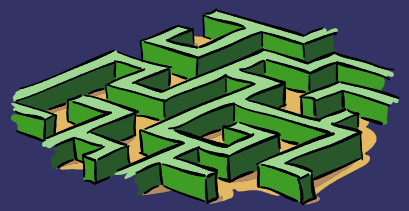


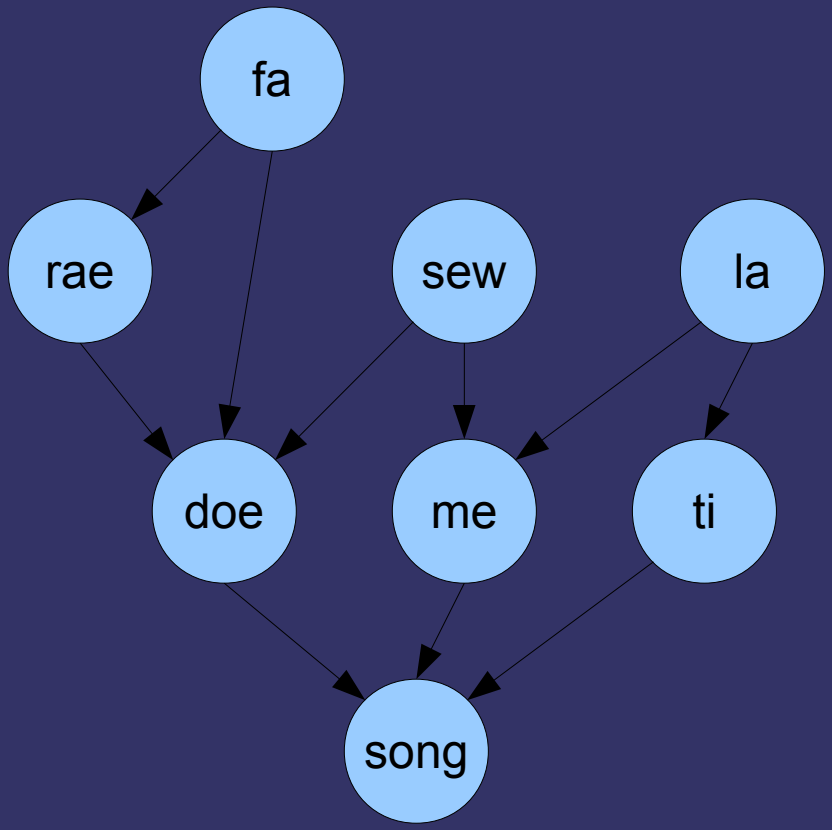
8 “named things”



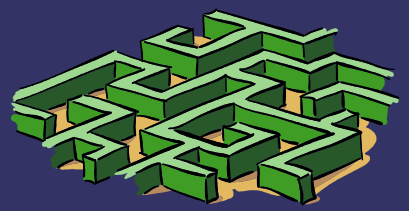


8 “connected things”



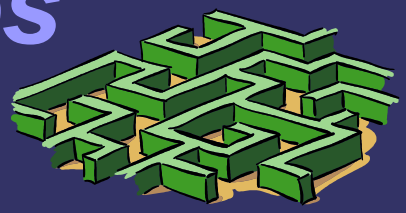


Dependency Graph



```
(def rae
    (ref {:name “rae”
          :last-update (Date.)
          :deps [fa]}))

(def doe
    (ref {:name “doe”
          :last-update (Date.)
          :deps [rae fa sew]}))
```

Dependency Graph Nodes

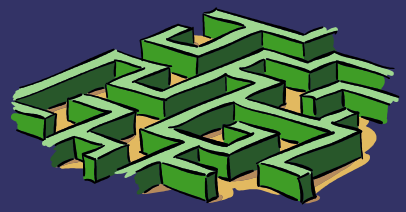


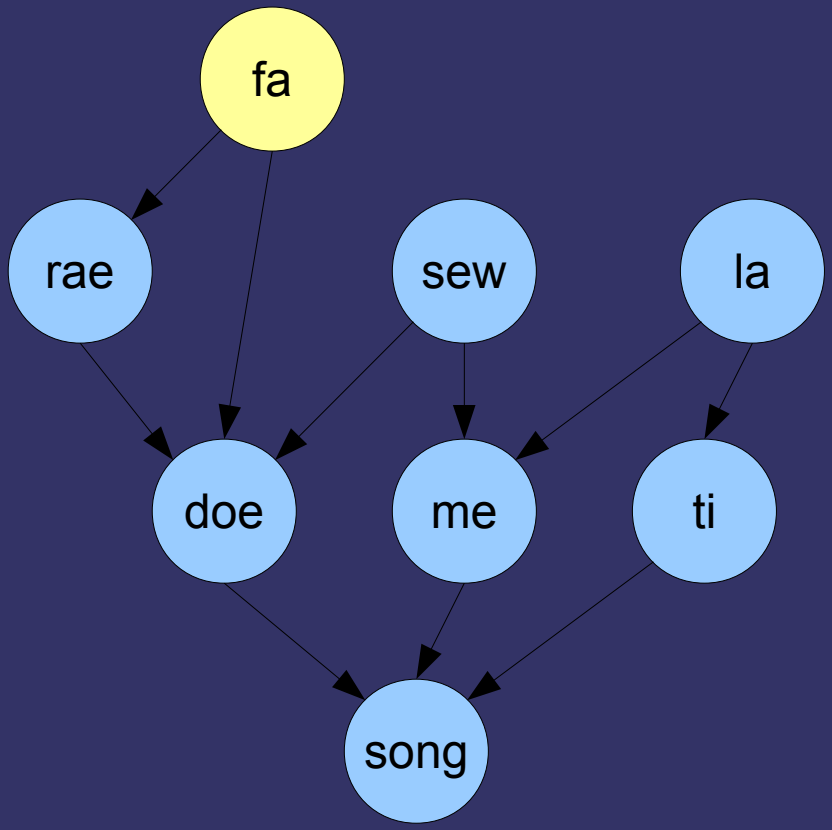
What are the leaves?

What is the root?

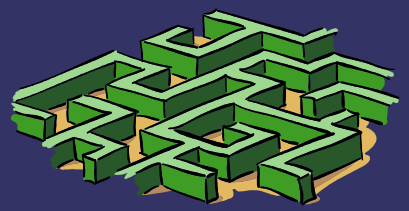
How long are the paths from the root to each leaf?

Dependency Graph

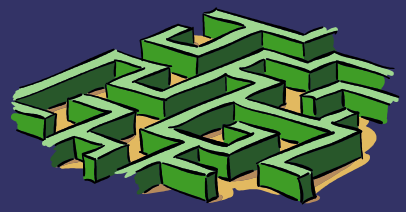




Dependency Graph

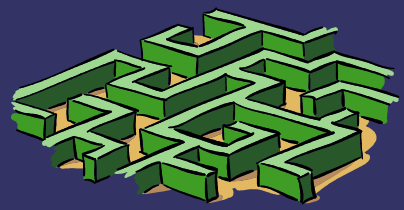


Logic Programming is about searching graphs



Logic Programming is about searching graphs

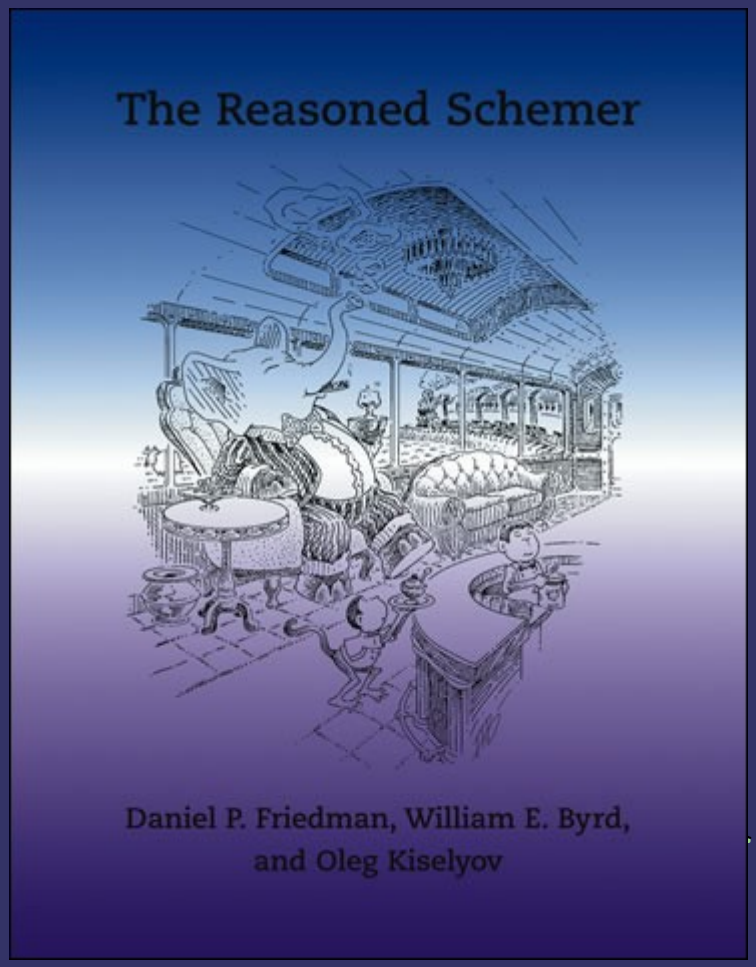
* Social Media
* Web search
* Turn-by-turn navigation
* Recommendation engines
* Dependency relationships
* Variable type analysis
* Natural language parsing
* Protein folding
* Business rules engines



Mini-Kanren (in Clojure)

Subset of Kanren

Described in “The Reasoned Schemer”



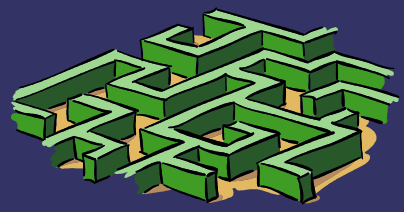
Mini-Kanren (in Clojure)

run:

Logic functions cannot be executed like 'normal' Clojure functions.

'run' queries a logic function for results.

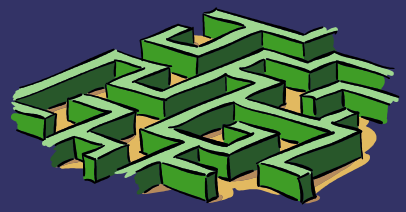
May return none, one or many results.



Mini-Kanren (in Clojure)

```
(run q
    ; some logic functions
 )
```

'q' is a logic variable

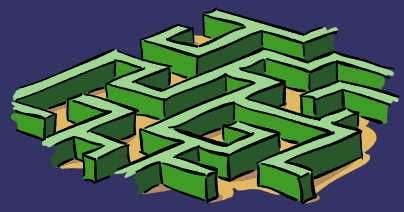


Mini-Kanren (in Clojure)

Binding:

Logic variables are not like normal, mutable variables.

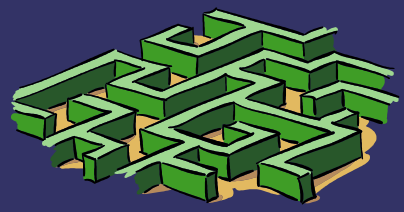
Logic variables are 'bound' to values or other logic variables with the logic function '&'.



Mini-Kanren (in Clojure)

```
(run q
   (& 7 q))

(run q
   (& q 7))
```

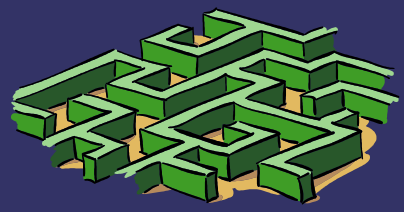
Both return (7) as a result.



Mini-Kanren (in Clojure)

```
(run q
   (& 9 7))

(run q
   (& 9 q)
   (& q 7))
```

Both return the empty list () as the result.



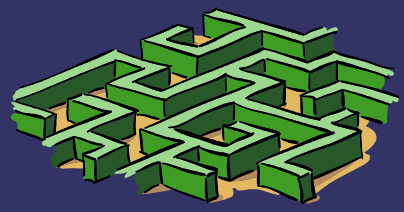
Mini-Kanren (in Clojure)

Alternatives:

When a variable needs to be bound to more than one value, use 'cond-e'

```
(run q
  (cond-e
    [(& q 4)]
    [(& q “not”)]))
```

Returns the list (4 “not”) as the result.

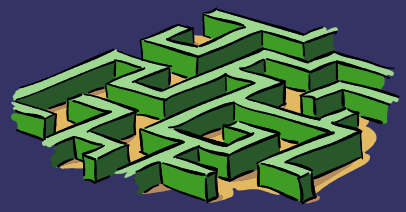


Mini-Kanren (in Clojure)

New logic variables:

```
(run q
    (exist [a b c]
        (& q b)
        (& a 9)
        (& b “string”)))
```

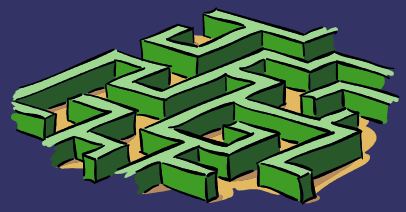
Returns a result of (“string”)



Zebra

9. Milk is drunk in the middle house.

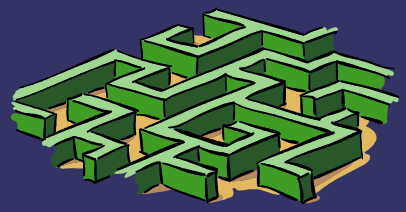
(& [_ _ [_ _ :milk _ _] _ _] h)



Zebra

10. The Norwegian lives in the first house on the left.

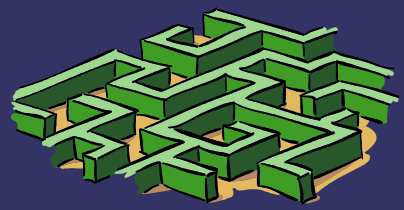
(& h [[:norwegian _ _ _ _] | _])



Zebra

15. The Norwegian lives next to the blue house.

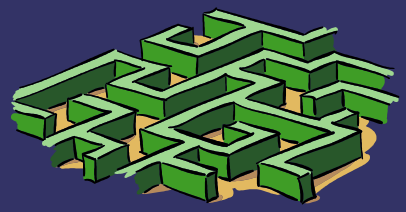
```
(next-to [ _ _ _ _ :blue]
         [:norwegian _ _ _ _ ]
         h)
```

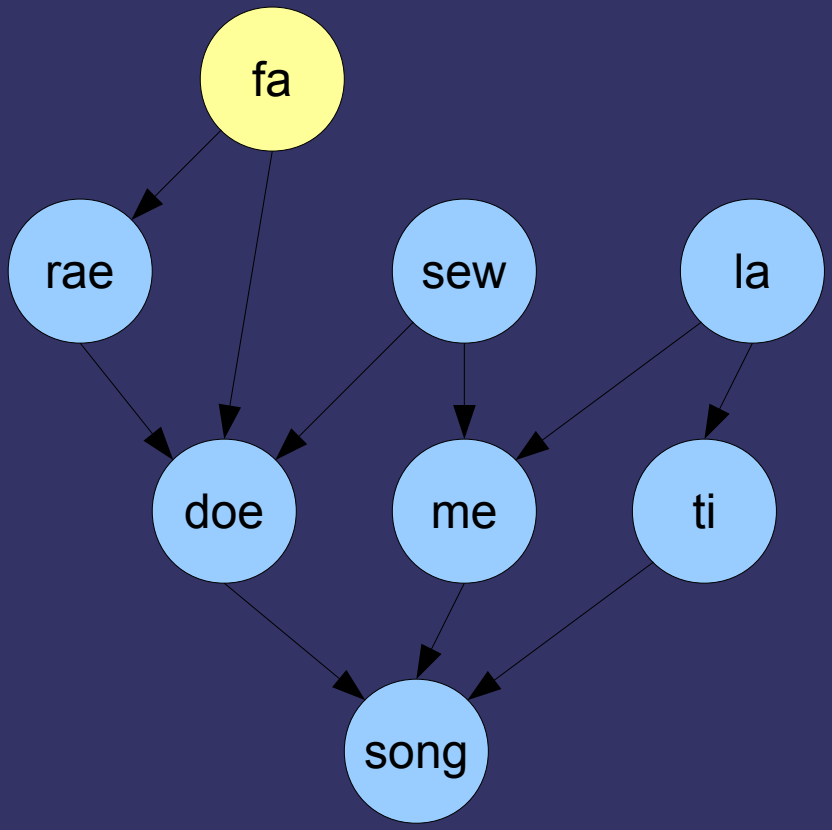


Zebra

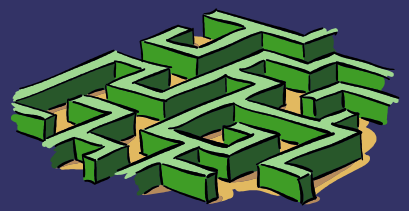
2. The Englishman lives in the red house.

(member-o [:englishman _ _ _ :red] h)



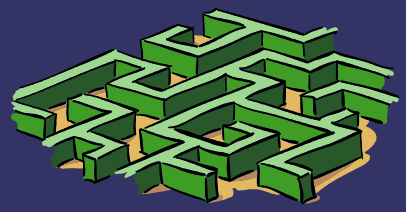


Dependency Graph



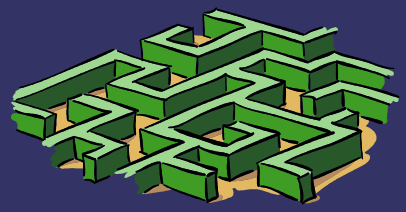
Dependency Graph

```
(def rae
    (ref {:name “rae”
          :last-update (Date.)
          :deps [fa]}))

(defn dependency [a b]
    (cond-e
        [(& a rae) (& b fa)]))
```



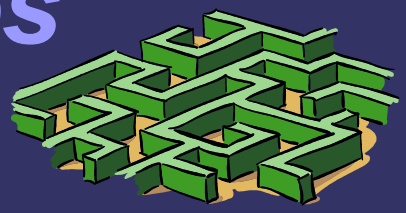
Dependency Graph

```
(defn dependency [a b]
    (cond-e
        [(& a rae) (& b fa)]
        [(& a doe) (& b fa)]
        [(& a doe) (& b rae)]
        [(& a doe) (& b sew)]
        [(& a me) (& b sew)]
        [(& a me) (& b la)]
        [(& a ti) (& b la)]
        [(& a song) (& b doe)]
        [(& a song) (& b me)]
        [(& a song) (& b ti)]))
```



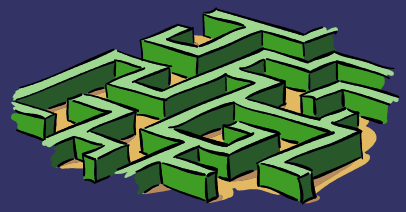
```
(def rae
    (ref {:name “rae”
          :last-update (Date.)}))

(def doe
    (ref {:name “doe”
          :last-update (Date.)}))
```

Dependency Graph Nodes



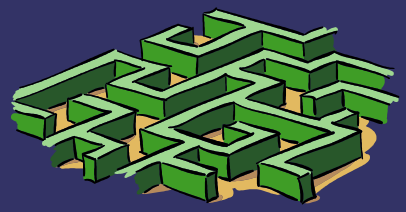
Dependency Graph

```
(defn dependency [a b]
    (cond-e
        [(& a rae) (& b fa)]
        [(& a doe) (& b fa)]
        [(& a doe) (& b rae)]
        [(& a doe) (& b sew)]
        [(& a me) (& b sew)]
        [(& a me) (& b la)]
        [(& a ti) (& b la)]
        [(& a song) (& b doe)]
        [(& a song) (& b me)]
        [(& a song) (& b ti)]))
```



Dependency Graph

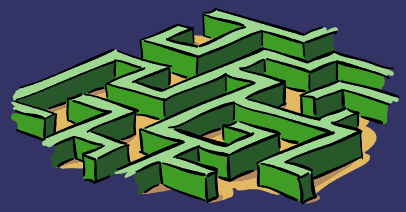
```
(run q
   (dependency doe q))
```



Dependency Graph

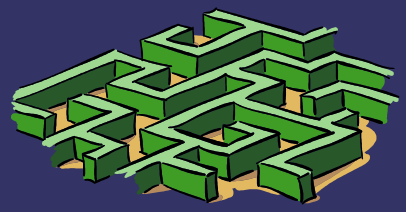
```
(run q
   (dependency doe q))
```

Result: (fa rae sew)



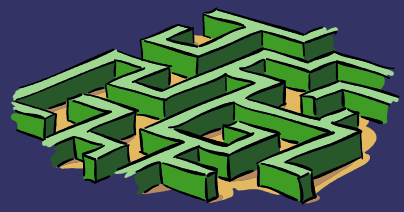
Dependency Graph

```
(defn dependency [a b]
    (cond-e
        [(& a rae) (& b fa)]
        [(& a doe) (& b fa)]
        [(& a doe) (& b rae)]
        [(& a doe) (& b sew)]
        [(& a me) (& b sew)]
        [(& a me) (& b la)]
        [(& a ti) (& b la)]
        [(& a song) (& b doe)]
        [(& a song) (& b me)]
        [(& a song) (& b ti)]))
```



Dependency Graph

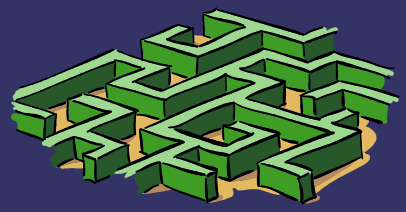
```
(run q
   (dependency q doe))
```



Dependency Graph

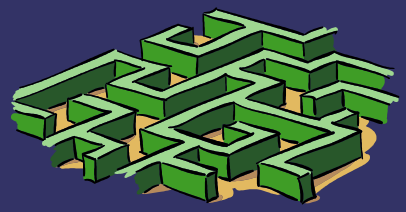
```
(run q
   (dependency q doe))
```

Result: (song)



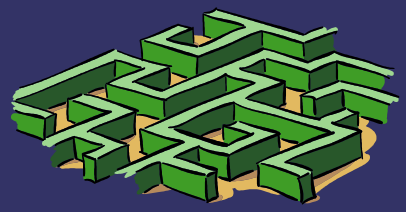
Dependency Graph

```
(defn dependency [a b]
    (cond-e
        [(& a rae) (& b fa)]
        [(& a doe) (& b fa)]
        [(& a doe) (& b rae)]
        [(& a doe) (& b sew)]
        [(& a me) (& b sew)]
        [(& a me) (& b la)]
        [(& a ti) (& b la)]
        [(& a song) (& b doe)]
        [(& a song) (& b me)]
        [(& a song) (& b ti)]))
```



Dependency Graph

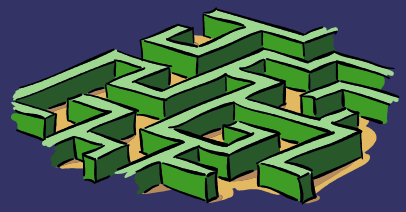
```
(run q
   (dependency q _))
```



Dependency Graph

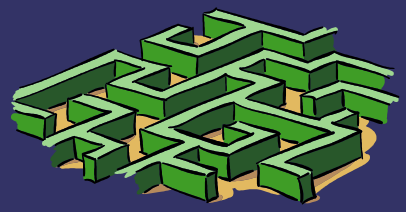
```
(run q
    (dependency q _))
```

Result: (rae doe doe doe me me ti song song song)



Dependency Graph

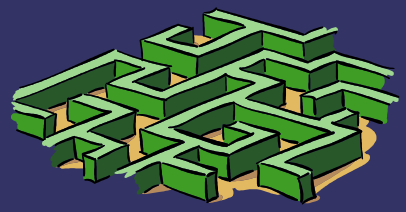
```
(defn dependency [a b]
    (cond-e
        [(& a rae) (& b fa)]
        [(& a doe) (& b fa)]
        [(& a doe) (& b rae)]
        [(& a doe) (& b sew)]
        [(& a me) (& b sew)]
        [(& a me) (& b la)]
        [(& a ti) (& b la)]
        [(& a song) (& b doe)]
        [(& a song) (& b me)]
        [(& a song) (& b ti)]))
```

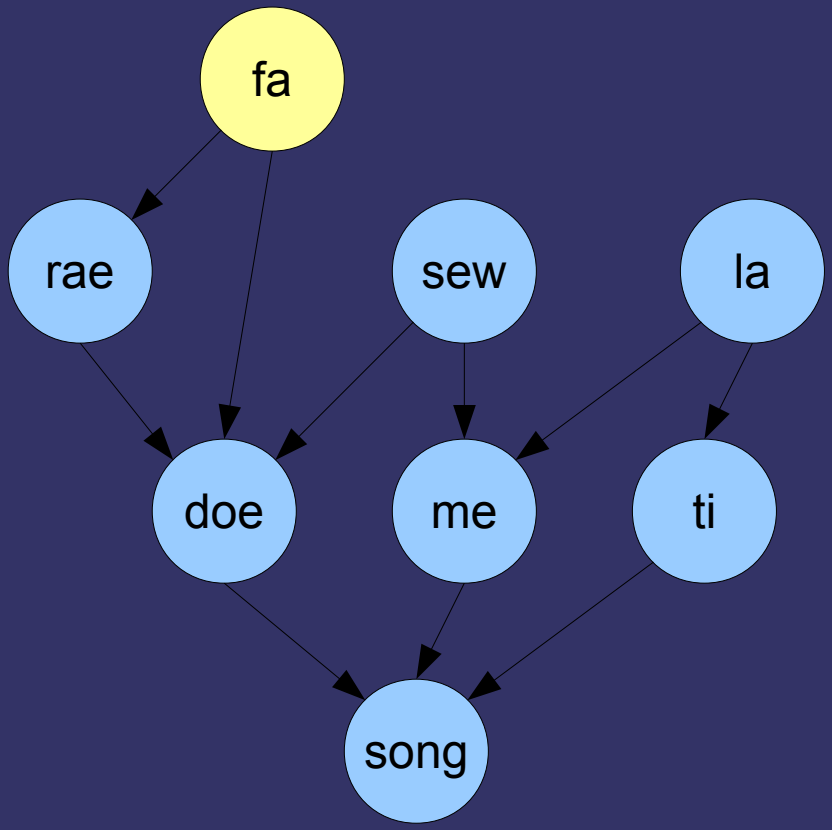


Dependency Graph

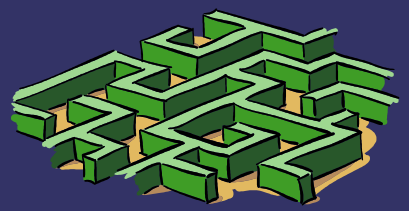
```
(set
    (run q
        (dependency q _)))
```

Result: #{rae doe me ti song}



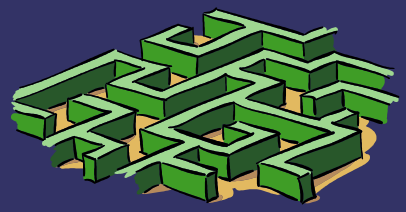


Dependency Graph



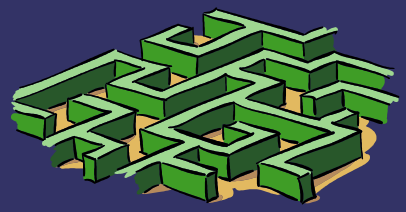
Dependency Graph

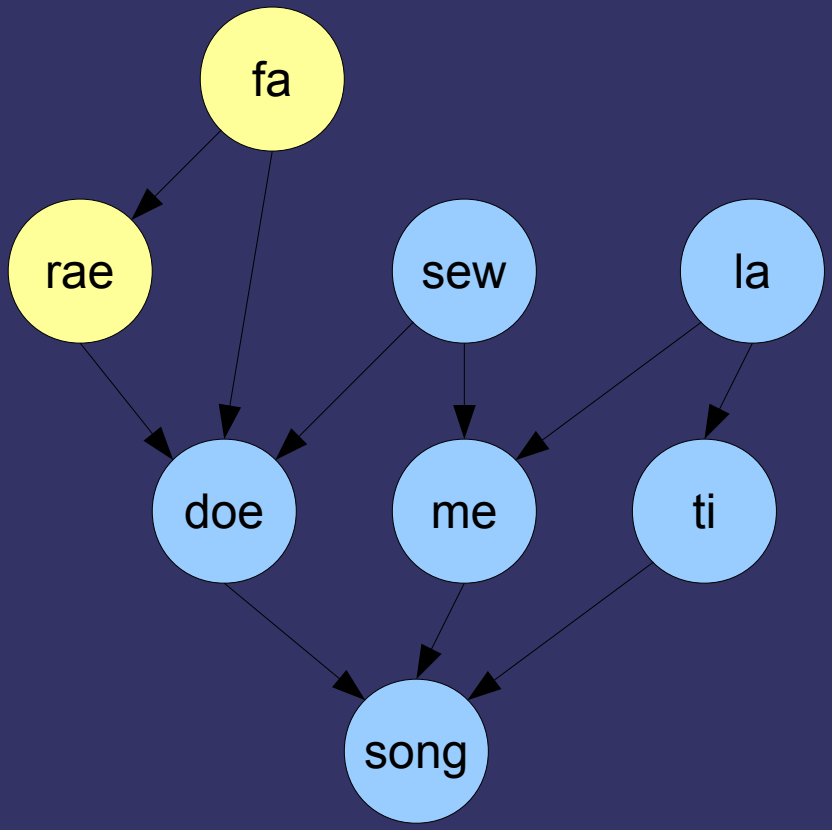
```
(defn make [a q]
    (exist [ dep ]
        (cond-e
            [(dependency a dep)
             (make dep q)]

            [(dependency a dep)
             (newer-than dep a)
             (& a q)])))
```



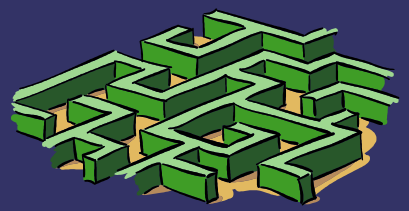
Dependency Graph

```
(def nodes
    (run q
        (make song q)))


(map update nodes)
```



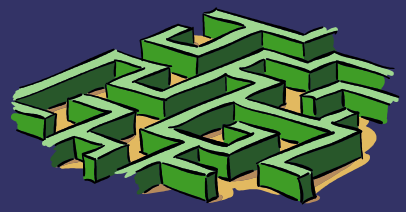


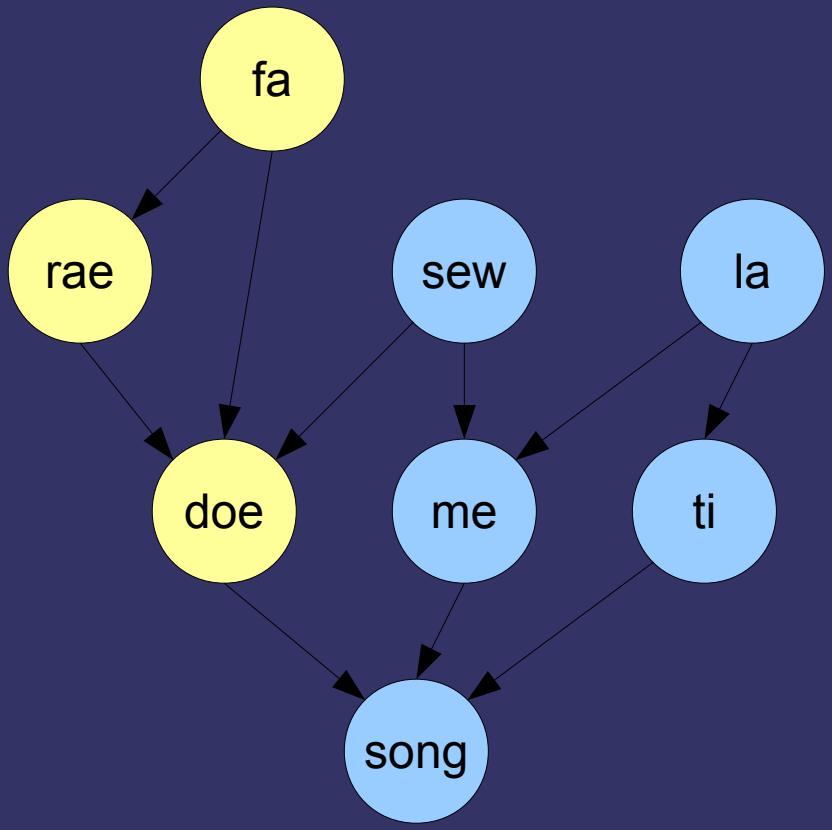
Dependency Graph



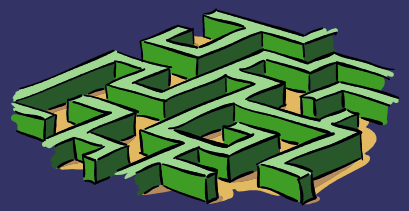
Dependency Graph

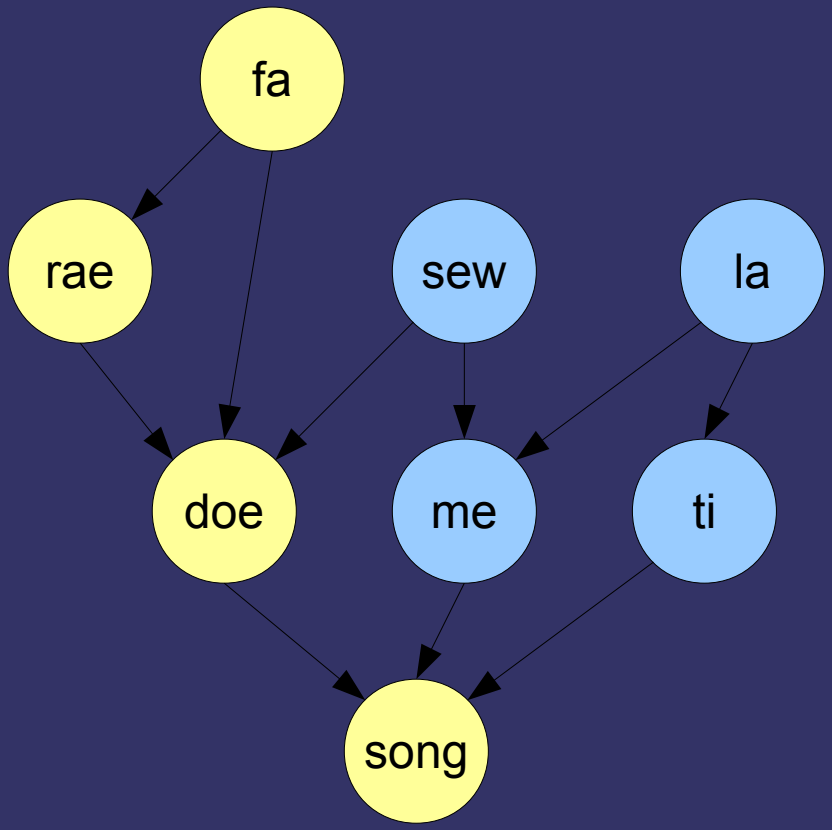
```
(defn make [a q]
    (exist [ dep ]
        (cond-e
            [(dependency a dep)
             (make dep q)]

            [(dependency a dep)
             (newer-than dep a)
             (& a q)])))
```



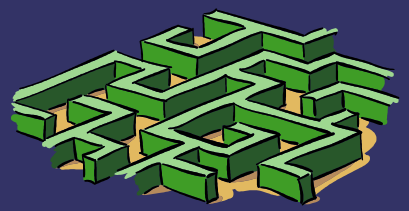


Dependency Graph



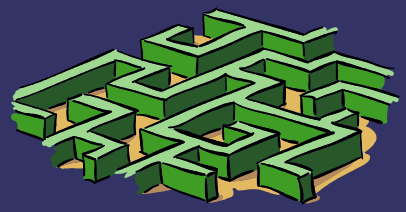


Dependency Graph



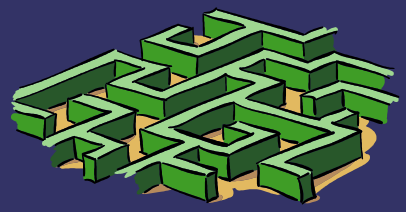
Dependency Graph

```
(defn make [a q]
    (exist [ dep ]
        (cond-e
            [(dependency a dep)
             (make dep q)]

            [(dependency a dep)
             (newer-than dep a)
             (& a q)])))
```



Logic Programming

Hides the complexity of graph searches.



Logic Programming

Hides the complexity of graph searches.

Leaving you to focus on the problem you're trying to solve.