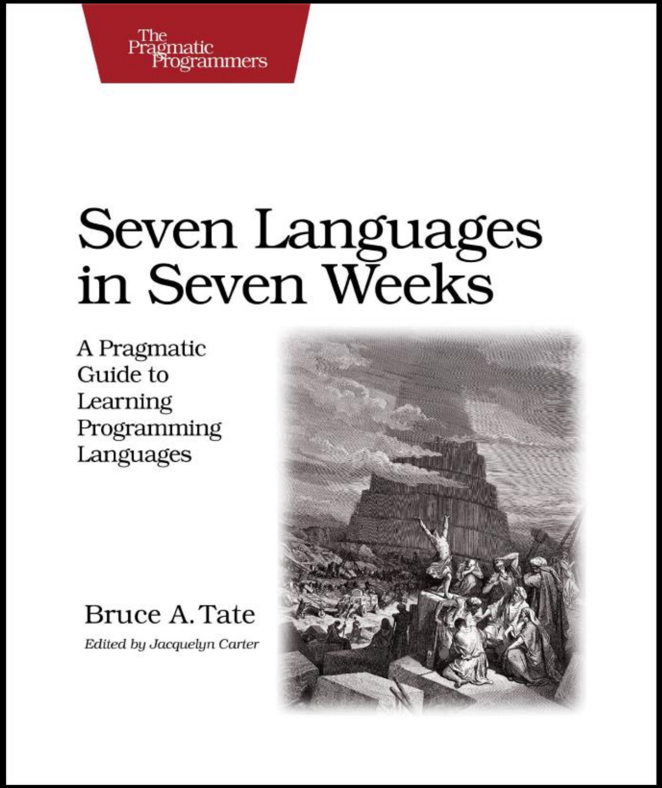
Java

Meetup

Berlin Code of Conduct

DISCORD



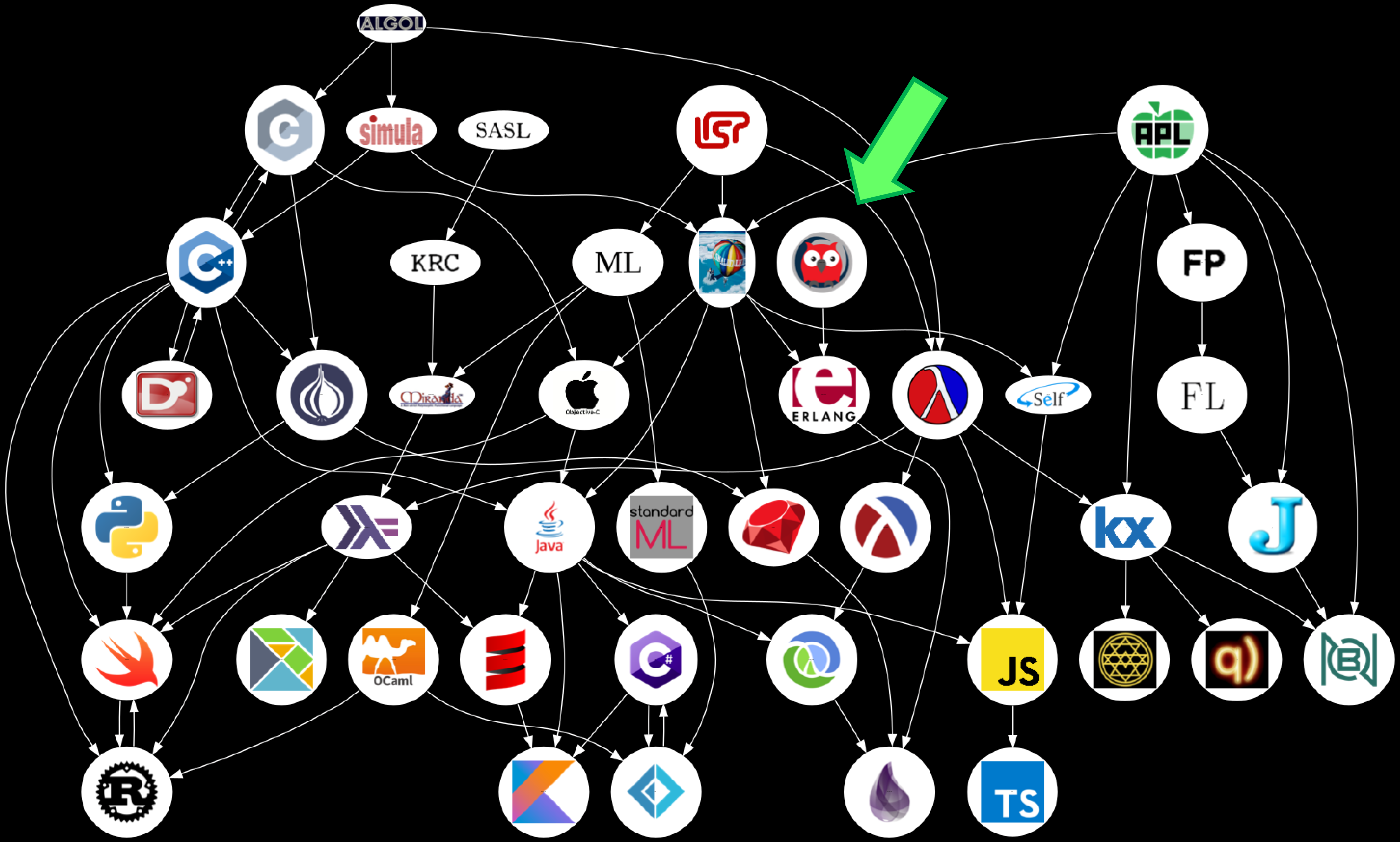
7 Languages in 7 Weeks

Prolog

4.1 About Prolog

Developed in 1972 by Alain Colmerauer and Phillipe Roussel, Prolog is a logic programming language that gained popularity in natural-language processing. Now, the venerable language provides the programming foundation for a wide variety of problems, from scheduling to expert systems. You can use this rules-based language for expressing logic and asking questions. Like SQL, Prolog works on databases, but the data will consist of logical rules and relationships. Like SQL, Prolog has two parts: one to express the data and one to query the data. In Prolog, the data is in the form of logical rules. These are the building blocks:

- *Facts.* A fact is a basic assertion about some world. (Babe is a pig; pigs like mud.)
- *Rules.* A rule is an inference about the facts in that world. (An animal likes mud if it is a pig.)
- *Query.* A query is a question about that world. (Does Babe like mud?)


ALGOL
simula
SASL
APL
KRC
ML
FP
Miranda
Objective-C
ERLANG
Self
FL
Java
standard
ML
kx
J
OCaml
JS
q)
TS

Basic Facts

In some languages, capitalization is entirely at the programmer's discretion, but in Prolog, the case of the first letter is significant. If a word begins with a lowercase character, it's an *atom*—a fixed value like a Ruby symbol. If it begins with an uppercase letter or an underscore, it's a *variable*. Variable values can change; atoms can't.

```prolog
likes(wallace, cheese).
likes(grommit, cheese).
likes(wendolene, sheep).

friend(X, Y) :- \+(X = Y), likes(X, Z), likes(Y, Z).

% | ?- likes(wallace,sheep).
% no

% | ?- likes(grommit,cheese).
% yes

% | ?- friend(wallace,wallace).
% no
```

```prolog
food_type(velveeta, cheese).
food_type(ritz, cracker).
food_type(spam, meat).
food_type(sausage, meat).
food_type(jolt, soda).
food_type(twinkie, dessert).

flavor(sweet, dessert).
flavor(savory, meat).
flavor(savory, cheese).
flavor(sweet, soda).

food_flavor(X, Y) :- food_type(X, Z), flavor(Y, Z).

% | ?- food_type(What, meat).

% What = spam ? ;
% What = sausage ? ;
% no
```

Figure 4.1: Map of some southeastern states

```prolog
different(red, green). different(red, blue).
different(green, red). different(green, blue).
different(blue, red). different(blue, green).

coloring(Alabama, Mississippi, Georgia, Tennessee, Florida) :-
  different(Mississippi, Tennessee),
  different(Mississippi, Alabama),
  different(Alabama, Tennessee),
  different(Alabama, Mississippi),
  different(Alabama, Georgia),
  different(Alabama, Florida),
  different(Georgia, Florida),
  different(Georgia, Tennessee).

% | ?- coloring(Alabama, Mississippi, Georgia, Tennessee, Florida).

% Alabama     = blue
% Florida     = green
% Georgia     = red
% Mississippi = red
% Tennessee   = green ?
```

Do:

- Make a simple knowledge base. Represent some of your favorite books and authors.
- Find all books in your knowledge base written by one author.
- Make a knowledge base representing musicians and instruments. Also represent musicians and their genre of music.
- Find all musicians who play the guitar.

1.
2.
3.
4.
5.

1.
2.
3.
4.
5.

Combinator Introductions

	Lambda Expression	Bird Name	APL	BQN	Haskell	Other	Introduced
I	`λa.a`	Identity	Same	Identity	`id`		Sch24
K	`λab.a`	Kestrel	Right	Right	`const`		Sch24
KI	`λab.b`	Kite	Left	Left			
S	`λabc.ac(bc)`	Starling		After	`<*>`	Hook (J)	Sch24
B	`λabc.a(bc)`	Bluebird	Atop	Atop	`.`		Cur29
C	`λabc.acb`	Cardinal	Commute	Swap	`flip`	`SWAP` (FORTH)	Cur29
W	`λab.abb`	Warbler	Self(ie)	Self	`join`	`DUP` (FORTH)	Cur29
B1	`λabcd.a(bcd)`	Blackbird	Atop	Atop	`.:`		Cur58
Ψ	`λabcd.a(bc)(bd)`	Psi	Over	Over	`on`		Cur58
S'	`λabcd.a(bd)(cd)`	Phoenix	Fork	Fork	`liftA2`	Infix Notation (FP)	Tur79
E	`λabcde.ab(cde)`	Eagle					Smu85
Ê	`λabcdefg.a(bcd)(efg)`	Bald Eagle					Smu85
D2	`λabcde.a(bd)(ce)`	Dovekie		Before w/ After			Smu85
D	`λabcd.ab(cd)`	Dove	Beside	After			Smu85
🚫	`λabcde.a(bde)(cde)`	*Golden Eagle*	Fork	Fork			Iv89
🚫	`λabc.a(bc)c`	*Violet Starling*		Before		backHook (I)	Loc12
🚫	`λabcd.a(bc)d`	*Zebra Dove*		Before			
🚫	`λabcde.a(bcd)e`	*Harpy Eagle*					

Combinator Introductions

	Lambda Expression	Bird Name	APL	BQN	Haskell	Other	Introduced
I	`λa.a`	Identity	Same	Identity	`id`		Sch24
K	`λab.a`	Kestrel	Right	Right	`const`		Sch24
KI	`λab.b`	Kite	Left	Left			
S	`λabc.ac(bc)`	Starling		After	`<*>`	Hook (J)	Sch24
B	`λabc.a(bc)`	Bluebird	Atop	Atop	`.`		Cur29
C	`λabc.acb`	Cardinal	Commute	Swap	`flip`	`SWAP` (FORTH)	Cur29
W	`λab.abb`	Warbler	Self(ie)	Self	`join`	`DUP` (FORTH)	Cur29
B1	`λabcd.a(bcd)`	Blackbird	Atop	Atop	`.:`		Cur58
Ψ	`λabcd.a(bc)(bd)`	Psi	Over	Over	`on`		Cur58
S'	`λabcd.a(bd)(cd)`	Phoenix	Fork	Fork	`liftA2`	Infix Notation (FP)	Tur79
E	`λabcde.ab(cde)`	Eagle					Smu85
Ê	`λabcdefg.a(bcd)(efg)`	Bald Eagle					Smu85
D2	`λabcde.a(bd)(ce)`	Dovekie		Before w/ After			Smu85
D	`λabcd.ab(cd)`	Dove	Beside	After			Smu85
🚫	`λabcde.a(bde)(cde)`	*Golden Eagle*	Fork	Fork			Iv89
🚫	`λabc.a(bc)c`	*Violet Starling*		Before		backHook (I)	Loc12
🚫	`λabcd.a(bc)d`	*Zebra Dove*		Before			
🚫	`λabcde.a(bcd)e`	*Harpy Eagle*					

```prolog
pl_paradigm(apl, array).
pl_paradigm(bqn, array).
pl_paradigm(haskell, functional).
pl_paradigm(clojure, functional).
pl_paradigm(cpp, multiparadigm).

pl_typing(apl, dynamic).
pl_typing(bqn, dynamic).
pl_typing(haskell, static).
pl_typing(clojure, dynamic).
pl_typing(cpp, static).

pl_author(apl, ken_iverson).
pl_author(bqn, marshall_lochbaum).
pl_author(haskell, simon_peyton_jones).
pl_author(clojure, rich_hickey).
pl_author(cpp, bjarne_stroustrup).
```

```prolog
pl_paradigm(apl, array).
pl_paradigm(bqn, array).
pl_paradigm(haskell, functional).
pl_paradigm(clojure, functional).
pl_paradigm(cpp, multiparadigm).

pl_typing(apl, dynamic).
pl_typing(bqn, dynamic).
pl_typing(haskell, static).
pl_typing(clojure, dynamic).
pl_typing(cpp, static).

pl_author(apl, ken_iverson).
pl_author(bqn, marshall_lochbaum).
pl_author(haskell, simon_peyton_jones).
pl_author(clojure, rich_hickey).
pl_author(cpp, bjarne_stroustrup).
```

```prolog
% | ?- pl_typing(What, dynamic).

% What = apl ? ;
% What = bqn ? ;
% What = clojure ? ;
% no


% | ?- pl_paradigm(What, array).

% What = apl ? ;
% What = bqn ? ;
% no
```

Lists and Tuples

Lists and tuples are a big part of Prolog. You can specify a list as [1, 2, 3] and a tuple as (1, 2, 3). Lists are containers of variable length, and tuples are containers with a fixed length. Both lists and tuples get much more powerful when you think of them in terms of unification.

You can deconstruct lists with [Head|Tail]. When you unify a list with this construct, Head will bind to the first element of the list, and Tail will bind to the rest, like this:

```prolog
count(0, []).
count(Count, [Head|Tail]) :-
   count(TailCount, Tail),
   Count is TailCount + 1.

sum(0, []).
sum(Total, [Head|Tail]) :-
   sum(Sum, Tail),
   Total is Head + Sum.

average(Average, List) :-
   sum(Sum, List),
   count(Count, List),
   Average is Sum/Count.
```

```prolog
count(0, []).
count(Count, [Head|Tail]) :-
   count(TailCount, Tail),
   Count is TailCount + 1.

sum(0, []).
sum(Total, [Head|Tail]) :-
   sum(Sum, Tail),
   Total is Head + Sum.

average(Average, List) :-
   sum(Sum, List),
   count(Count, List),
   Average is Sum/Count.
```

```prolog
% | ?- count(What, [1]).

% What = 1 ? ;
% no

% | ?- sum(What, [1,2,3]).

% What = 6 ? ;
% (1 ms) no

% | ?- average(What, [1,2,3,4]).

% What = 2.5 ? ;
% no
```

Do:

- Reverse the elements of a list.
- Find the smallest element of a list.
- Sort the elements of a list.

```prolog
% 1. Reverse a list

rev([], []).
rev([H|T], RevList) :- revHelper(T, [H], RevList).
revHelper([], Acc, Acc).
revHelper([H|T], Acc, RevList) :- revHelper(T, [H|Acc], RevList).

% | ?- reverse([1,2,3],What).

% What = [3,2,1]
% yes
% | ?- reverse([],What).

% What = []
% yes
```

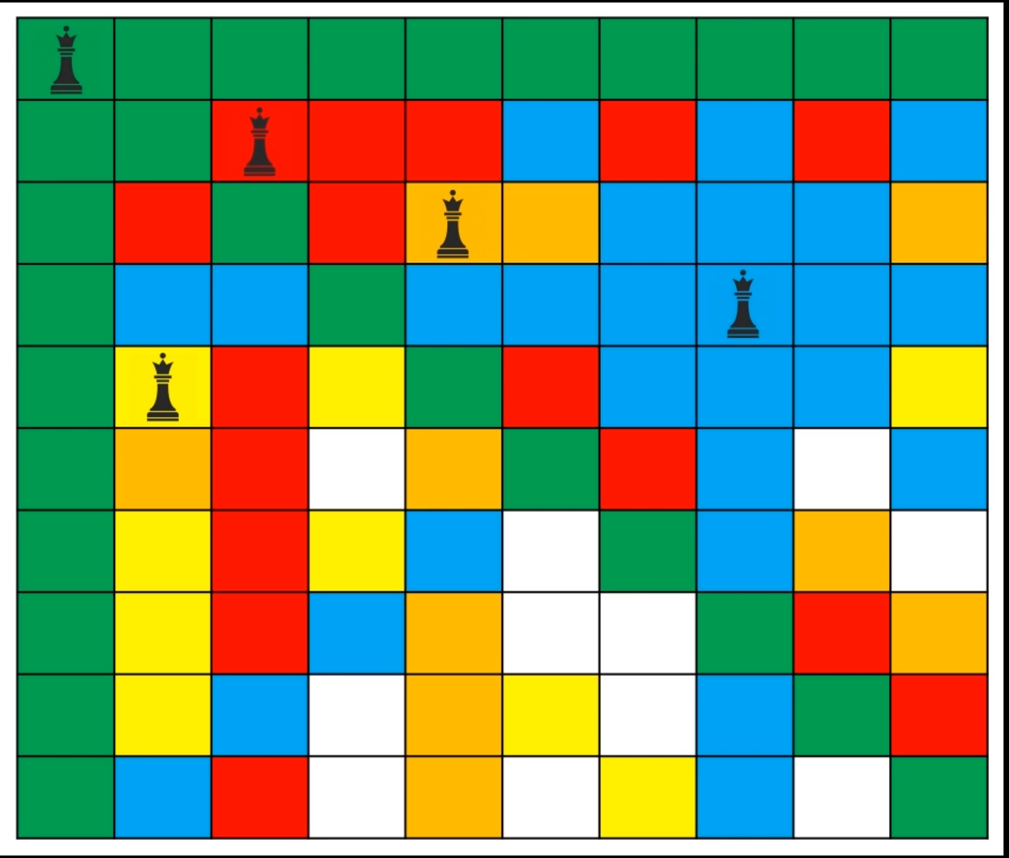
```
% 2. Find minimum

minimum(
minimumH
minimumH                                          Min);
                                                , Min)).


% | ?- m

% What =
% yes

% | ?- m

% What = 42 ? ;
% no
```

, = and

; = or

```prolog
% 3. Sort

isSorted([]).
isSorted([_]).
isSorted([X,Y|T]) :-
    X =< Y,
    isSorted([Y|T]).

mySort(List, SortedList) :-
    permutation(List, SortedList),
    isSorted(SortedList).
```

```prolog
% | ?- mySort([2,1], What).
% What = [1,2] ? ;
% no

% | ?- mySort([3,2,1], What).
% What = [1,2,3] ? ;
% no

% | ?- mySort([3,2,1,3], What).
% What = [1,2,3,3] ? ;
% What = [1,2,3,3] ? ;
% no

% | ?- mySort([5,4,7,8,10,2,4,1,3], What).
% What = [1,2,3,4,4,5,7,8,10] ? ;
% What = [1,2,3,4,4,5,7,8,10] ? ;
% (167 ms) no
```



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

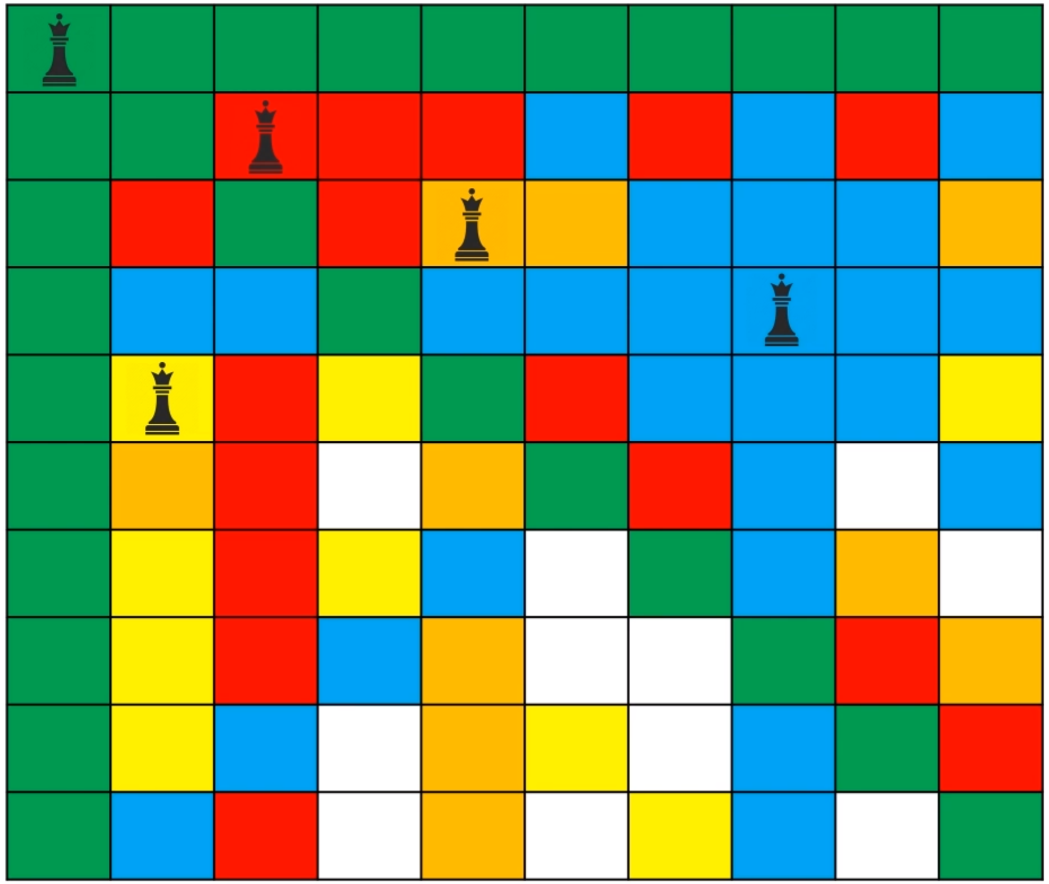
SRM 742: Problem 2 - SixteenQueens

[0, 3]

[0, 7]

3

```
ans = { 1, 2,
        2, 4
        4, 2 }
```

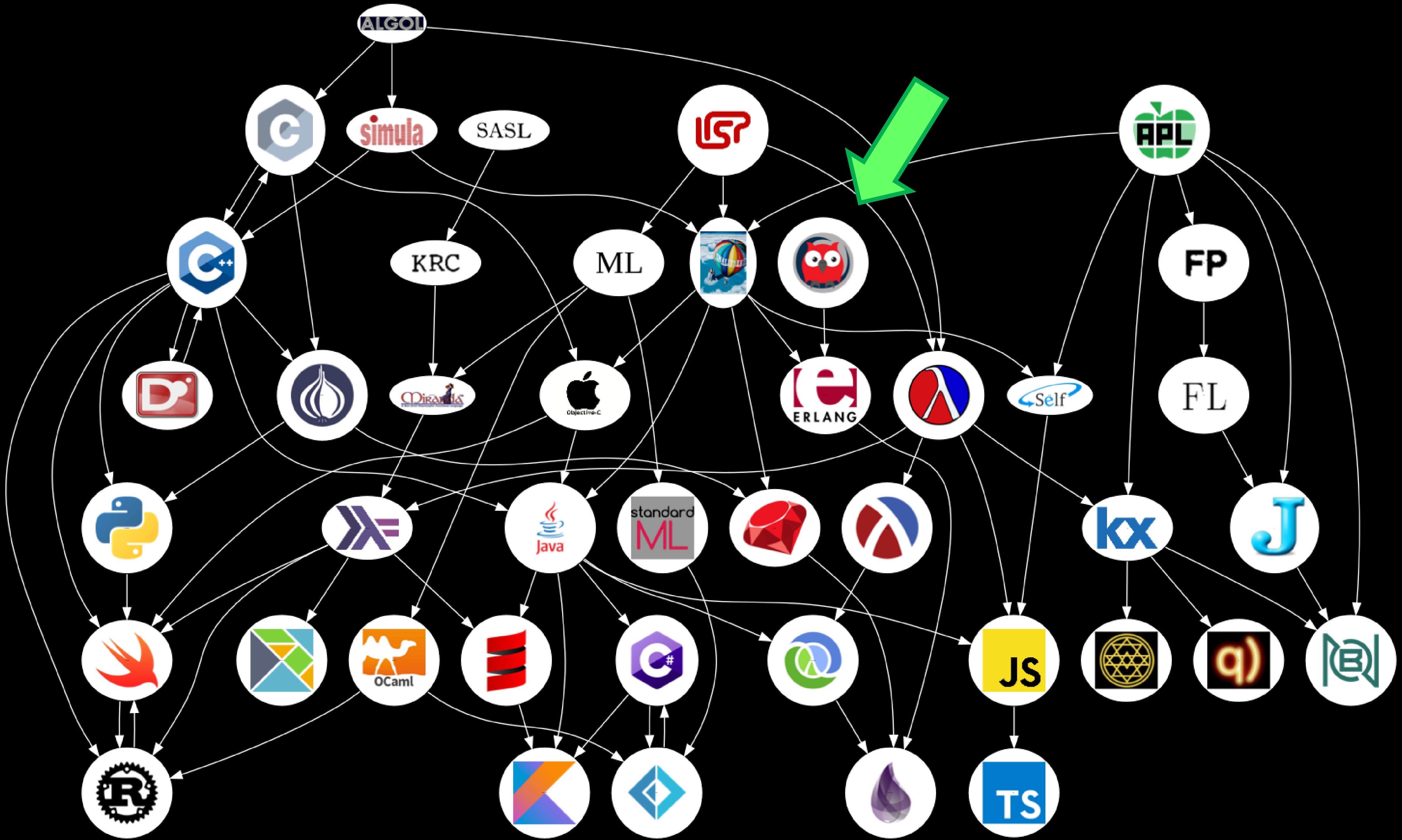


```prolog
valid_queen((Row, Col)) :-
    Range = [1,2,3,4,5,6,7,8],
    member(Row, Range), member(Col, Range).

valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).

rows([], []).
rows([(Row, _)|QueensTail], [Row|RowsTail]) :-
    rows(QueensTail, RowsTail).

cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
    cols(QueensTail, ColsTail).

diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
    Diagonal is Col - Row,
    diags1(QueensTail, DiagonalsTail).

diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
    Diagonal is Col + Row,
    diags2(QueensTail, DiagonalsTail).
```

```prolog
valid_queen((Row, Col)) :-
    Range = [1,2,3,4,5,6,7,8],
    member(Row, Range), member(Col, Range).

valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).

rows([], []).
rows([(Row, _)|QueensTail], [Row|RowsTail]) :-
    rows(QueensTail, RowsTail).

cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
    cols(QueensTail, ColsTail).

diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
    Diagonal is Col - Row,
    diags1(QueensTail, DiagonalsTail).

diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
    Diagonal is Col + Row,
    diags2(QueensTail, DiagonalsTail).
```

```prolog
eight_queens(Board) :-
    length(Board, 8),
    valid_board(Board),
    rows(Board, Rows),
    cols(Board, Cols),
    diags1(Board, Diags1),
    diags2(Board, Diags2),
    fd_all_different(Rows),
    fd_all_different(Cols),
    fd_all_different(Diags1),
    fd_all_different(Diags2).
```

```
sudoku(Puzzle, Solution) :-
    Solution = Puzzle,
    Puzzle = [S11, S12, S13, S14,
              S21, S22, S23, S24,
              S31, S32, S33, S34,
              S41, S42, S43, S44],
    fd_domain(Puzzle, 1, 4),
    Row1 = [S11, S12, S13, S14],
    Row2 = [S21, S22, S23, S24],
    Row3 = [S31, S32, S33, S34],
    Row4 = [S41, S42, S43, S44],
    Col1 = [S11, S21, S31, S41],
    Col2 = [S12, S22, S32, S42],
    Col3 = [S13, S23, S33, S43],
    Col4 = [S14, S24, S34, S44],
    Square1 = [S11, S12, S21, S22],
    Square2 = [S13, S14, S23, S24],
    Square3 = [S31, S32, S41, S42],
    Square4 = [S33, S34, S43, S44],
    valid([Row1, Row2, Row3, Row4,
           Col1, Col2, Col3, Col4,
           Square1, Square2, Square3, Square4]).

valid([]).
valid([Head | Tail]) :- fd_all_different(Head), valid(Tail).
```



ALGOL
simula
SASL
APL
KRC
ML
FP
Objective-C
ERLANG
Self
FL
standard ML
Java
kx
J
OCaml
JS
TS

Java

Meetup