# ECS 140A Discussion 4

Ty Feng, Chris Fernandez

2:10-3:50pm Thursday, Aug 31 2023

## Today’s agenda

*Homework 3*

*Haskell Exercises*

*Logic Programming & Prolog*

*Q&A*

## Midterm 2

Prof. Eiselt said:

No Java or syntax questions

Questions about functional programming in general

Read and write Haskell code

Logic programming paradigm

Maybe read simple Prolog code

Won’t be asked to write Prolog code until the final exam

## Important HW3 Instructions

For each of the following problems, you are to provide two solutions: one using the pattern-matching techniques, and one not using those pattern-matching techniques
For example, the two solutions for **myappend** might look like:

```
-- without pattern matching
 myappend list1 list2
   | list1 == [] = list2
   | otherwise   = (head list1):(myappend (tail list1) list2)
-- with pattern matching
myappend_pm [] list2     = list2
 myappend_pm (x:xs) list2 = x:(myappend_pm xs list2)
```

Note that I have ended the name of the pattern-matching solution with **"_pm"**. Please do the same for all the pattern-matching solutions that you submit for this assignment.

## Important HW3 Instructions

To implement the functions below, you may use only the following list operations -- **`':'`, `'head'`, `'tail'`, `'null'`, `and` `'elem'`.** (Note that there are pattern matching equivalents for most of these.) Do not resort to just giving a new name to an existing Haskell function that already does what we want your function to do. So, for example

```
myappend inlist1 inlist2 = inlist1 ++ inlist2
```

wouldn't get you any points. Also, do not traverse any list more times than is necessary.

## Important HW3 Instructions

Please make sure you name your functions with the names provided in the HW (with and without the _pm suffix). Also, include type declarations with your functions. It will be good practice.

Submit your solutions as a single file named "hw3.hs".

## HW3 will be partially autograded

It’s very important to follow the previous instructions, especially the naming conventions. Otherwise, the autograder won’t be able to run the test cases, and you’ll get 0.

We will look through the code manually to make sure you have used recursion for all your problems, and to make sure you don’t use an existing Haskell function.

## HW3 tip: use recursion!

To solve hw3 problems, you will need to use recursion extensively.

- Make sure you define your base cases, or recursion won’t stop
- Think about what should the function do at each recursive step
- Recurse on n-1 elements

## HW3 Problem 5

5) myreverse

There’s a simple but inefficient solution to this problem, and a much more efficient solution.  For full credit, provide the more efficient solution.

```
myreverse "" => ""
myreverse "abc" => "cba"
myreverse [1, 2, 3] => [3, 2, 1]
myreverse [] => []
```

The simple approach uses the “++” operator to append an element to the end of the list. Appending is expensive. You may use a helper function that stores the resulting list (acts as an accumulator), and recurse on the input list to add element by element to the resulting list using the constructor (:).

## HW3 Problem 5

5) myreverse

There's a simple but inefficient solution to this problem, and a much more efficient solution.  For full credit, provide the more efficient solution.

```
myreverse "" => ""
myreverse "abc" => "cba"
myreverse [1, 2, 3] => [3, 2, 1]
myreverse [] => []
```

What's the runtime of ++?

What's the runtime of (:)?

## HW3 Problem 5

5) myreverse

There's a simple but inefficient solution to this problem, and a much more efficient solution. For full credit, provide the more efficient solution.

myreverse "" => ""
myreverse "abc" => "cba"
myreverse [1, 2, 3] => [3, 2, 1]
myreverse [] => []

What's the runtime of ++? Answer: O(N)

What's the runtime of (:)? Answer: O(1)

Every time we use the `++` operator, Haskell has to walk through all the elements of the list, which takes O(n) time. The myreverse solution would recurse on the tail of the list, until the list is empty, and each time append the head to the recursion result. This process takes O(n) time as well.

To illustrate, for a list of n elements,
- reversing the first element takes 1 step,
- reversing the first two elements takes 3 steps,
- reversing the first three elements takes 6 steps,
- and so on.

The total number of steps to reverse the whole list is therefore 1 + 3 + 6 + ... + (n-1 + n) = n(n+1)/2, which simplifies to O(n^2).

## HW3 Problem 5

5) myreverse

There’s a simple but inefficient solution to this problem, and a much more efficient solution.  For full credit, provide the more efficient solution.

```
myreverse "" => ""
myreverse "abc" => "cba"
myreverse [1, 2, 3] => [3, 2, 1]
myreverse [] => []
```

What’s the overall runtime of a solution using ++? Answer: O(N^2)

What’s the overall runtime of a solution using (:)? Answer: O(N)

## Haskell

More Haskell practice:

[tyfeng.com/ecs140a/discussion4.html](tyfeng.com/ecs140a/discussion4.html)

To run any Haskell code:

```
$ ghci haskellCode.hs
ghci> functionName arg1 arg2 argN
:q to quit
```

## Haskell

Example from class: myinsert
Write a function, myinsert, that takes in a sorted list and a value and inserts the value into the list while keeping it sorted.

```
myinsert [1,3,5,7] 4 => [1,3,4,5,7]
myinsert "aceg" 'd' => "acdeg"
```

## Haskell

Example from class: myinsert
Write a function, myinsert, that takes in a sorted list and a value and inserts the value into the list while keeping it sorted.

myinsert [1,3,5,7] 4 => [1,3,4,5,7]
myinsert "aceg" 'd' => "acdeg"

Non-pattern matching:

```
myinsert :: Ord a => [a] -> a -> [a]
myinsert list1 y
      | null list1 = [y]
      | y <= head list1 = y:list1
      | otherwise = (head list1) :
(myinsert y (tail list1))
```

Pattern matching:

```
myinsert :: Ord a => [a] -> a -> [a]
myinsert [] y = [y]
myinsert (x:xs) y
  | y <= x = y : (x : xs)
  | otherwise = x : myinsert xs y
```

## Haskell

1) mycount
Write a function, mycount, that takes in a list and a value and returns the number of times the value appears in the list.

```
mycount [1,2,3,1,2,1,3,1] 1 => 4
mycount "hello" 'l' => 2
```

## Haskell

1) mycount
Write a function, mycount, that takes in a list and a value and returns the number of times the value appears in the list.

```
mycount [1,2,3,1,2,1,3,1] 1 => 4
mycount "hello" 'l' => 2
```

Solution:

```
mycount :: Eq a => [a] -> a -> Int
mycount [] _ = 0
mycount (x:xs) y
  | x == y = 1 + mycount xs y
  | otherwise = mycount xs y
```

## Haskell

2) myunion
Write a function that takes two lists and returns a new list that is a union of the two. If a value exists in both lists it only shows up once in the returned list.

```
myunion [1,2,3] [2,3,4] => [1,2,3,4]
myunion [1,2,3] [] => [1,2,3]
myunion [] [4,5,6] => [4,5,6]
```

## Haskell

2) myunion
Write a function that takes two lists and returns a new list that is a union of the two. If a value exists in both lists it only shows up once in the returned list.

```
myunion [1,2,3] [2,3,4] => [1,2,3,4]
myunion [1,2,3] [] => [1,2,3]
myunion [] [4,5,6] => [4,5,6]
```

Solution:

```
myunion :: Eq a => [a] -> [a] -> [a]
myunion list1 [] = list1
myunion list1 (y:ys)
  | elem y list1 = myunion list1 ys
  | otherwise    = myunion (y:list1) ys
```

## Haskell

3) mytake
Write a function, mytake, that takes in a positive integer n and a list and returns a new list with the first n elements of the original list.

```
mytake 3 "haskell" => "has"
mytake 2 [1,2,3,4,5] => [1,2]
```

## Haskell

3) mytake

Write a function, mytake, that takes in a positive integer n and a list and returns a new list with the first n elements of the original list.

```
mytake 3 "haskell" => "has"
mytake 2 [1,2,3,4,5] => [1,2]
```

Solution:

```
mytake :: Int -> [a] -> [a]
mytake 0 _ = []
mytake _ [] = []
mytake n (x:xs) = x : mytake (n-1) xs
```

## Haskell

4) myremove

Write a function, myremove, that takes in a value and a list and returns a new list with all occurrences of the value removed.

myremove 2 [1,2,3,2,4,2] => [1,3,4]

myremove 'a' "banana" => "bnn"

## Haskell

4) myremove
Write a function, myremove, that takes in a value and a list and returns a new list with all occurrences of the value removed.
myremove 2 [1,2,3,2,4,2] => [1,3,4]
myremove 'a' "banana" => "bnn"

Solution:

```
myremove :: Eq a => a -> [a] -> [a]
myremove _ [] = []
myremove y (x:xs)
  | x == y = myremove y xs
  | otherwise = x : myremove y xs
```

## Haskell

5) myproduct

Write a function myproduct that takes a list of numbers and returns the product of all the numbers in the list.

```
myproduct [1,2,3,4,5] => 120
myproduct [3,3,3,3] => 81
myproduct [] => 1
```

## Haskell

### 5) myproduct

Write a function myproduct that takes a list of numbers and returns the product of all the numbers in the list.

```
myproduct [1,2,3,4,5] => 120
myproduct [3,3,3,3] => 81
myproduct [] => 1
```

Solution:

```
myproduct :: Num a => [a] -> a
myproduct [] = 1
myproduct (x:xs) = x * myproduct xs
```

# Haskell

6) myRemoveFirstOccurrence
This function will remove the first occurrence of a provided element from a list.

```
myRemoveFirstOccurrence 3 [1, 3, 5, 3, 7] => [1,5,3,7]
myRemoveFirstOccurrence 'a' "haskell" => "hskell"
```

# Haskell

6) myRemoveFirstOccurrence
This function will remove the first occurrence of a provided element from a list.

```
myRemoveFirstOccurrence 3 [1, 3, 5, 3, 7] => [1,5,3,7]
myRemoveFirstOccurrence 'a' "haskell" => "hskell"
```

Solution:

```
myRemoveFirstOccurrence :: Eq a => a -> [a] -> [a]
myRemoveFirstOccurrence _ [] = []
myRemoveFirstOccurrence x (y:ys)
  | x == y    = ys
  | otherwise = y : myRemoveFirstOccurrence x ys
```

## Haskell

7) mydrop
Write a function, mydrop, that takes in a positive integer n and a list and returns a new list with the first n elements dropped.

```
mydrop 3 "haskell" => "kell"
mydrop 2 [1,2,3,4,5] => [3,4,5]
```

## Haskell

7) mydrop
Write a function, mydrop, that takes in a positive integer n and a list and returns a new list with the first n elements dropped.

```
mydrop 3 "haskell" => "kell"
mydrop 2 [1,2,3,4,5] => [3,4,5]
```

Solution:
```
mydrop :: Int -> [a] -> [a]
mydrop 0 xs = xs
mydrop _ [] = []
mydrop n (_:xs) = mydrop (n-1) xs
```

## Haskell

8) mymaximum
Write a function, mymaximum, that takes in a list of numbers and returns the maximum value in the list.

```
mymaximum [1,5,3,2,4] => 5
mymaximum [3.5, 2.2, 1.3] => 3.5
```

## Haskell

8) mymaximum
Write a function, mymaximum, that takes in a list of numbers and returns the maximum value in the list.

```
mymaximum [1,5,3,2,4] => 5
mymaximum [3.5, 2.2, 1.3] => 3.5
```

Solution:

```
mymaximum :: Ord a => [a] -> a
mymaximum [x] = x
mymaximum (x:(y:ys))
 | x > y     = mymaximum (x:ys)
 | otherwise = mymaximum (y:ys)
```

## Haskell

9) myzip
Write a function, myzip, that takes in two lists and returns a new list of pairs, where each pair contains one element from each list.

```
myzip "abc" "123" => [('a','1'),('b','2'),('c','3')]
myzip [1,2,3] ['a','b'] => [(1,'a'),(2,'b')]
```

## Haskell

9) myzip
Write a function, myzip, that takes in two lists and returns a new list of pairs, where each pair contains one element from each list.

```
myzip "abc" "123" => [('a','1'),('b','2'),('c','3')]
myzip [1,2,3] ['a','b'] => [(1,'a'),(2,'b')]
```

Solution:

```
myzip :: [a] -> [b] -> [(a,b)]
myzip [] _ = []
myzip _ [] = []
myzip (x:xs) (y:ys) = (x, y) : myzip xs ys
```

## Haskell

10) myReplaceOne:
This function will replace the first occurrence of a provided element in a list with a new one.

```
myReplaceOne 3 7 [1, 3, 5, 3, 7] => [1, 7, 5, 3, 7]
myReplaceOne 'a' 'x' "haskell" => "hxskell"
```

# Haskell

10) myReplaceOne:
This function will replace the first occurrence of a provided element in a list with a new one.

```
myReplaceOne 3 7 [1, 3, 5, 3, 7] => [1, 7, 5, 3, 7]
myReplaceOne 'a' 'x' "haskell" => "hxskell"
```

Solution:

```
myReplaceOne :: Eq a => a -> a -> [a] -> [a]
myReplaceOne _ _ [] = []
myReplaceOne x new (y:ys)
  | x == y    = new : ys
  | otherwise = y : myReplaceOne x new ys
```

## Logic Programming Paradigm

- Logic Programming is based on the declarative paradigm of computer programming.
- In this paradigm, programs are built using formal logic to represent relationships and constraints.
- Users specify the underlying data through a symbolic system of relations, in the forms of facts and rules.
- It would solve logical problems like puzzles, series etc.
- In logic programming we have a knowledge base which we know before. This knowledge base along with the question is given to machine, which produces result.
- In logical programming the main emphasis is on knowledge base and the problem. The execution of the program is very much like proof of mathematical statement, e.g., Prolog

## Logic Programming Paradigm

- Logic programming is a computer programming paradigm where program statements express facts and rules about problems within a system of formal logic.
- Rules are written as logical clauses with a head and a body.
- For instance, "H is true if B1, B2, and B3 are true."
- Facts are expressed similar to rules, but without a body; for instance, "H is true."
- Logic programming languages includes a combination of declarative and imperative statements.
- They may also include procedural statements, such as "To solve H, solve B1, B2, and B3."

## Benefits of Logic Programming

- Logical relationships can easily be transferred into facts and rules for use in a logic program.
- Users do not have to be experts in traditional programming to use it. They only have to understand the logical domain and know how to add the predicates. Logic programming syntax is straightforward.
- It can be used to represent very complicated ideas and rapidly refine an existing data model.
- It is very good at pattern matching.
- It is efficient in terms of memory management and data storage.
- It allows data to be presented in several different ways.

## Drawbacks of Logic Programming

- It can be challenging to translate knowledge into facts and rules.
- Programs can be difficult to debug and test.
- Unintended side effects are much more difficult to control in logic programming than they are in traditional languages.
- Slight changes can generate vastly different outcomes.

## Applications of Logic Programming

- Artificial Intelligence/Machine Learning: It is especially relevant because it provides a structured method of defining domain-specific knowledge. AI systems use their facts and rules to analyze new queries and statements.
- Natural Language Processing (NLP): NLP handles interactions between people and computers. It relies upon a system of rules to interpret and understand speech or text.
- Database Management: Logic programming can determine the best place in a database to store new data. It can also analyze the contents of a database and retrieve the most useful and relevant results for a query.
- Predictive Analysis: Logic programs can sort through a large amount of data, analyze results and make predictions.

## Prolog

- This is the original logic programming language, developed at a French university in 1972.
- It was designed for use in artificial intelligence and is still the most popular logic programming language today.
- Prolog mainly uses the declarative programming paradigm but also incorporates imperative programming.
- It is designed for symbolic computation and inference manipulation.
- Its logical rules are expressed in terms of relations and take the form of Horn clauses.
- Queries use these relations to generate results.
- Prolog operates by negating the original query and trying to find information proving it false.

## Prolog

- In Prolog, the Horn clause is written as:

  H :- B1,...,Bn.

- Antecedents (or left-hand side of the sentence) in the Horn Clause are called subgoals or tail.
- The consequent (or right-hand side of the sentence) in the Horn Clause is called goal or head.
- A Horn Clause with no tail is a fact. For example, rainy(seattle). does not depend on any condition.
- A Horn Clause with a tail is a rule. For example, snowy(X) :- rainy(X),cold(X)..

## Example

- The following is a simple Prolog program:

```
man(socrates).
mortal(X) :- man(X).
```

- The first line can be read, ``Socrates is a man.'' It is a base clause, which represents a simple fact.
- The second line can be read, ``X is mortal if X is a man;'' in other words, ``All men are mortal.'' This is a clause, or rule, for determining when its input X is ``mortal.''
- The symbol ``:-'', sometimes called a turnstile, is pronounced ``if''.

## Syntax

- A program Prolog consists of one or more predicates; each predicate consists of one or more clauses.
- A clause is a base clause if it is unconditionally true, that is, it has no ``if part.''Two clauses belong to the same predicate if they have the same functor (name) and the same arity (number of arguments).
- A functor is followed by zero or more arguments; the arguments are enclosed in parentheses and separated by commas.
- There must be no space between the functor and the opening parenthesis! If there are no arguments, the parentheses are omitted.
- Arguments may be any legal Prolog values or variables.

## Syntax (contd)

- A variable is written as a sequence of letters and digits, beginning with a capital letter. The underscore (_) is considered to be a capital letter.
- An atom is a fundamental data type that represents a constant. An atom is a sequence of characters that is used to represent a unique, indivisible entity within a Prolog program.
- An atom is any sequence of letters and digits, beginning with a lowercase letter. Alternatively, an atom is any sequence of characters, enclosed by single quotes (')
- Comments begin with the characters /* and end with */. Comments are not restricted to a single line, but may not be nested.

## Example

- parent(john, mary).
  In this example, the predicate parent represents a parent-child relationship. It states that "john" is a parent of "mary." The predicate name is "parent" and it has two arguments ("john" and "mary").

- ancestor(X, Y) :- parent(X, Y).
  ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
  In this case, the ancestor predicate is defined using rules. It states that "X" is an ancestor of "Y" if "X" is a parent of "Y," or if there exists a "Z" such that "X" is a parent of "Z" and "Z" is an ancestor of "Y."

## Example 1  fib1.pl

Simple recursive approach

```
fib(0, 1) :- !.
fib(1, 1) :- !.
fib(N, F) :-
        N > 1,
        N1 is N-1,
        N2 is N-2,
        fib(N1, F1),
        fib(N2, F2),
        F is F1+F2.
```

## Example 1  fib1.pl

Simple recursive approach

```
fib(0, 1) :- !.
fib(1, 1) :- !.
fib(N, F) :-
        N > 1,
        N1 is N-1,
        N2 is N-2,
        fib(N1, F1),
        fib(N2, F2),
        F is F1+F2.
```

```
[trace]  ?- fib(3, F).
   Call: (10) fib(3, _28116) ? creep
   Call: (11) 3>1 ? creep
   Exit: (11) 3>1 ? creep
   Call: (11) _30818 is 3+ -1 ? creep
   Exit: (11) 2 is 3+ -1 ? creep
   Call: (11) _32334 is 3+ -2 ? creep
   Exit: (11) 1 is 3+ -2 ? creep
   Call: (11) fib(2, _33844) ? creep
   Call: (12) 2>1 ? creep
   Exit: (12) 2>1 ? creep
   Call: (12) _36114 is 2+ -1 ? creep
   Exit: (12) 1 is 2+ -1 ? creep
   Call: (12) _37630 is 2+ -2 ? creep
   Exit: (12) 0 is 2+ -2 ? creep
   Call: (12) fib(1, _39140) ? creep
   Exit: (12) fib(1, 1) ? creep
   Call: (12) fib(0, _40650) ? creep
   Exit: (12) fib(0, 1) ? creep
   Call: (12) _33844 is 1+1 ? creep
   Exit: (12) 2 is 1+1 ? creep
   Exit: (11) fib(2, 2) ? creep
   Call: (11) fib(1, _44428) ? creep
   Exit: (11) fib(1, 1) ? creep
   Call: (11) _28116 is 2+1 ? creep
   Exit: (11) 3 is 2+1 ? creep
   Exit: (10) fib(3, 3) ? creep
F = 3.
```

## Example 2 fib2.pl

Recursive bottom-up approach

```
fib(0, 1) :- !.
fib(1, 1) :- !.
fib(N, F) :-
        fib(1, N, 1, 1, F).

fib(N, N, _, F, F) :- !.
fib(N0, N, F0, F1, F) :-
        N1 is N0 + 1,
        F2 is F0 + F1,
        fib(N1, N, F1, F2, F).
```

## Example 2 fib2.pl

Recursive bottom-up approach

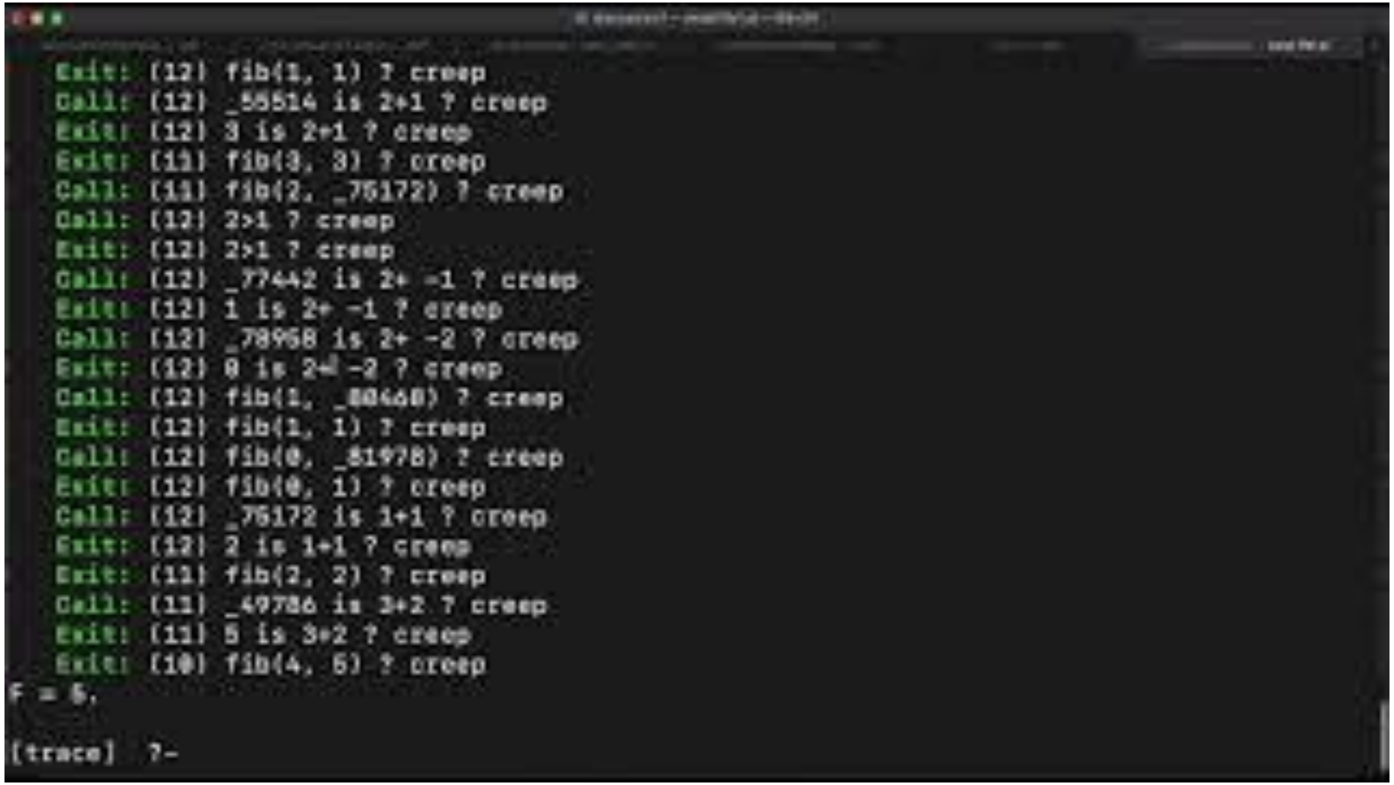
```
fib(0, 1) :- !.
fib(1, 1) :- !.
fib(N, F) :-
        fib(1, N, 1, 1, F).

fib(N, N, _, F, F) :- !.
fib(N0, N, F0, F1, F) :-
        N1 is N0 + 1,
        F2 is F0 + F1,
        fib(N1, N, F1, F2, F).
```

```
[trace]  ?- fib(3,F).
   Call: (10) fib(3, _25074) ? creep
   Call: (11) fib(1, 3, 1, 1, _25074) ? creep
   Call: (12) _27034 is 1+1 ? creep
   Exit: (12) 2 is 1+1 ? creep
   Call: (12) _28544 is 1+1 ? creep
   Exit: (12) 2 is 1+1 ? creep
   Call: (12) fib(2, 3, 1, 2, _25074) ? creep
   Call: (13) _30832 is 2+1 ? creep
   Exit: (13) 3 is 2+1 ? creep
   Call: (13) _32342 is 1+2 ? creep
   Exit: (13) 3 is 1+2 ? creep
   Call: (13) fib(3, 3, 2, 3, _25074) ? creep
   Exit: (13) fib(3, 3, 2, 3, 3) ? creep
   Exit: (12) fib(2, 3, 1, 2, 3) ? creep
   Exit: (11) fib(1, 3, 1, 1, 3) ? creep
   Exit: (10) fib(3, 3) ? creep
F = 3.
```

# Fib1.pl and Fib2.pl are explained in this video

Discussion video on Prolog from Fall 2021: [https://youtu.be/AEIwBcM848U](https://youtu.be/AEIwBcM848U)



```
   Exit: (12) fib(1, 1) ? creep
   Call: (12) _55514 is 2+1 ? creep
   Exit: (12) 3 is 2+1 ? creep
   Exit: (11) fib(3, 3) ? creep
   Call: (11) fib(2, _75172) ? creep
   Call: (12) 2>1 ? creep
   Exit: (12) 2>1 ? creep
   Call: (12) _77442 is 2+ -1 ? creep
   Exit: (12) 1 is 2+ -1 ? creep
   Call: (12) _78958 is 2+ -2 ? creep
   Exit: (12) 0 is 2+ -2 ? creep
   Call: (12) fib(1, _80468) ? creep
   Exit: (12) fib(1, 1) ? creep
   Call: (12) fib(0, _81978) ? creep
   Exit: (12) fib(0, 1) ? creep
   Call: (12) _75172 is 1+1 ? creep
   Exit: (12) 2 is 1+1 ? creep
   Exit: (11) fib(2, 2) ? creep
   Call: (11) _49786 is 3+2 ? creep
   Exit: (11) 5 is 3+2 ? creep
   Exit: (10) fib(4, 5) ? creep
F = 5.

[trace]  ?-
```

## Please read on your own: Prolog Resources

A Concise Introduction to Prolog, David Matuszek

tyfeng.com/ecs140a/prolog/concise/

Fib1.pl and fib2.pl

tyfeng.com/ecs140a/prolog