Dayananda Sagar University

School of Engineering, Hosur Main Road, Kudlu Gate, Bengaluru-560 068

ARTIFICIAL INTELLIGENCE-II
20AM3602

20AM3602

SEMESTER – VI

Course Code: 20AM3602

Prof. Pradeep Kumar K

Dept. of CS&E (AIML)

DSU, Bangalore

20AM3602

Course Objectives:

- Gain a perspective of state space search in AI
- Investigate applications of AI techniques in intelligent agents, expert systems,
- Understanding Natural language and its importance in AI

Course outcomes:

CO No.	Outcomes	Bloom's Taxonomy Level
CO1	Demonstrate awareness and a fundamental understanding of various applications of AI techniques in intelligent agents, expert systems, artificial neural networks and other machine learning models.	L5
CO2	Demonstrate proficiency in applying scientific method to models using PROLOG and LISP	L4
CO3	Apply AI techniques to real-world problems to develop intelligent systems	L4
CO4	Design, analyse and demonstrate AI applications or systems that apply to real life problems.	L5

20AM3602

Text Documents Requirement

TEXT BOOKS:

1. George F Luger, William A Stubblefield. Artificial Intelligence: Structures and Strategies for Complex Problem Solving, 3rd Edition, Addison Wesley Longman, Inc

2. Introduction to Artificial Intelligence and Expert Systems by Dan W. Patterson, Pearson Education

3. Forsyth and Ponce, “Computer Vision – A Modern Approach”, Second Edition, Prentice Hall, 2011.

REFERENCES:

1. Daniel Jurafsky, James H. Martin. Speech and Language processing: An Introduction to Natural Language processing, computational Linuistics and Speech, Pearson Publication, 2014

2. Artificial Intelligence: Concepts and Applications, Wiley (1 January 2021); ISBN-10 : 8126519932.

Content of Syllabus

20AM3602

Module-1:	Contact Hours
AI as Representation and Search: The predicate Calculus, Using Inference Rules to produce Predicate Calculus Expressions, Structures and Strategies for State Space Search: Graph Theory, Strategies for State Space Search, using state space to represent Reasoning with predicate calculus Recursion-based search, production systems, Predicate Calculus and Planning	10
Module – 2:	
Programming Languages for AI An Introduction to PROLOG: Syntax for predicate Calculus programming, ADTs in PROLOG, A production system example in PROLOG. An Introduction to LISP: LISP - A brief overview, Search in LISP, Pattern matching in LISP	08
Module – 3:	
Expert Systems: Introduction to AI agents, Overview of expert system Technology, Rule Based expert systems, Model based reasoning, Case-based Reasoning, knowledge-Representation problem, An Expert system Shell in LISP.	08

Content of Syllabus

20AM3602

Module-4:	Contact Hours
Understanding Natural Language: Role of knowledge in Language Understanding, Language Understanding: A symbolic approach, Syntax, Combining Syntax and semantic in ATN parsers, Stochastic Tool for language Analysis, Natural Language Applications: Story Understanding and Question Answering	07
Module – 5:	
Pattern Recognition: Introduction, Recognition and classification process, learning Classification patterns, Recognizing and Understanding Speech Computer Vision Overview and State-of-the-art, Fundamentals of Image Formation, Transformation: Orthogonal, Euclidean, Affine, Projective, etc; Fourier Transform, Convolution and Filtering, Image Enhancement, Restoration, Histogram Processing	08

20AM3602

Module-2 : AI as Representation and Search:

- An Introduction to PROLOG: Syntax for predicate Calculus programming, ADTs in PROLOG, A production system example in PROLOG. An Introduction to LISP: LISP - A brief overview, Search in LISP, Pattern matching in LISP

20AM3602

An Introduction to PROLOG -The language of logic

- Prolog is a logic programming language.
- It has important role in artificial intelligence. Unlike many other programming languages, Prolog is intended primarily as a declarative programming language.
- In prolog, logic is expressed as relations (called as **Facts and Rules**).
- Core heart of prolog lies at the logic being applied.
- Formulation or Computation is carried out by running a query over these relations.

- Prolog (programming in logic) is a logic-based programming language: programs correspond to sets of logical formulas and the Prolog interpreter uses logical methods to resolve queries.
- Prolog is a declarative language: you specify what problem you want to solve rather than how to solve it.
- Prolog is very useful in some problem areas, such as artificial intelligence, natural language processing, databases, . . . , but pretty useless in others, such as for instance graphics or numerical algorithms.

- sudo apt-get install swi-prolog (Installation in Ubuntu)

20AM3602

Syntax and Basic Fields

- In prolog, We declare some **facts**. These facts constitute the Knowledge Base of the system.
- We can **query** against the Knowledge Base. We get **output as affirmative** if our query is already in the knowledge Base or it is implied by Knowledge Base, otherwise we get output as negative.
- So, **Knowledge Base** can be considered similar to database, against which we can query.
- Prolog facts are expressed in definite pattern. Facts contain entities and their relation.
- Entities are written within the **parenthesis separated by comma (,).** Their relation is expressed at the start and outside the parenthesis. Every fact/rule ends with a dot (.).

Example

- Format : relation(entity1, entity2, ...k'th entity).

Example :

- friends(raju, mahesh).
- singer(sonu).
- odd_number(5).

Explanation :

- These facts can be interpreted as :
- raju and mahesh are friends.
- sonu is a singer.
- 5 is an odd number.

Running queries :
A typical prolog query can be asked as :

Query 1 : ?- singer(sonu).
Output : Yes.

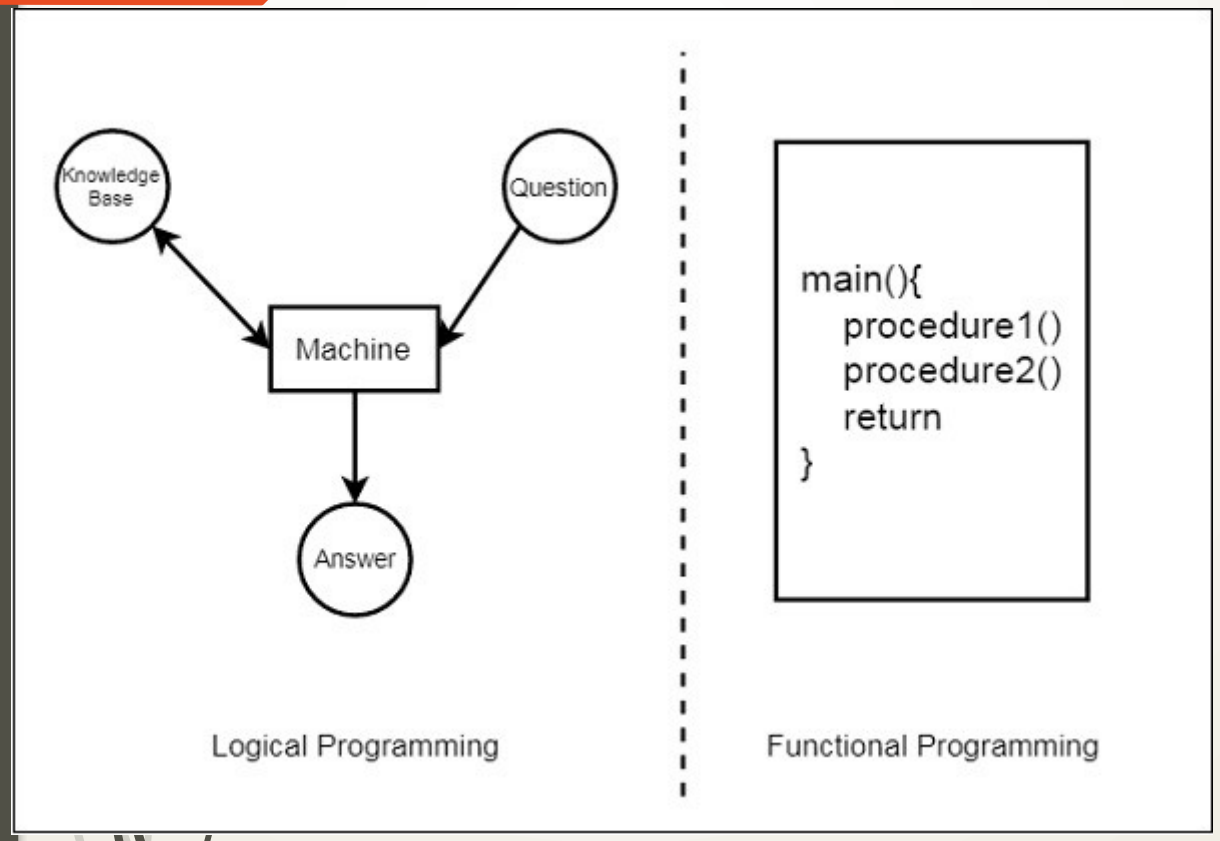
Explanation : As our knowledge base contains the above fact, so output was 'Yes', otherwise it would have been 'No'.

Query 2 : ?- odd_number(7).
Output : No.

Explanation : As our knowledge base does not contain the above fact, so output was 'No'.

20AM3602

Logic and Functional Programming



- Functional Programming: Define the procedures, and the rule how the procedures work. These procedures work step by step to solve one specific problem based on the algorithm.
- Logic Programming, we will provide knowledge base. Using this knowledge base, the machine can find answers to the given questions, which is totally different from functional programming.
- In functional programming, we have to mention how one problem can be solved, but in logic programming we have to specify for which problem we actually want the solution. Then the logic programming automatically finds a suitable solution that will help us solve that specific problem.

Functional Programming	Logic Programming
Functional Programming follows the Von-Neumann Architecture, or uses the sequential steps.	Logic Programming uses abstract model, or deals with objects and their relationships.
The syntax is actually the sequence of statements like (a, s, I).	The syntax is basically the logic formulae (Horn Clauses).
The computation takes part by executing the statements sequentially.	It computes by deducting the clauses.
Logic and controls are mixed together.	Logics and controls can be separated.

20AM3602

Represent Facts and Rules

- Facts (propositions); e.g. male(Philip).
- Rules (implications):e.g.
  - Parent(X,Y):=father(X, Y).
  - Parent(X,Y):=mother(X, Y).
  - Parent(X,Y):=father(X, Y); mother(X, Y).
- Questions(start point of execution):e.g- ?-parent(x, Y).

A Simple Rule for analysis of the given

English description:	Logic formula:	Prolog:
If X is male, F is the father of X, M is the mother of X, F is the father of Y, M is the mother of Y Then X is a brother of Y	male(X) ∧ father(F, X) ∧ mother(M, X) ∧ father(F, Y) ∧ mother(M, Y) → brother(X, Y)	brother(X, Y) :- male(X), father(F,X), mother(M,X), father(F,Y), mother(M, Y).

20AM3602

Prolog as Logic

- Horn Clause- a subset of first-order logic
- Logic formula:
- Description – A-e.g. Philip is male.
  - Logic OR - A v B: A; B.
  - Logic AND – A ^ B: A, B.
  - Logic NOT- `A : not A.
  - Logic implication: A→ B: B:-A.
- Each expression terminates with a.
- Deductive inference – Modus Ponents

A

A→ B

B

Program execution

- Start from the question
- Matching – match the question with facts and rules by substitution (try)
- Unification – looking for unified solution (consistent solution)
- Backtracking – once fails, go back to try another case
- Divide-and-conquer
  - Divide problem into sub problems
  - Solve all sub problems
  - Integrate sub solutions to build the final solution
  - Solutions to all sub problems must ne consistent.

Example

- Suppose we have some knowledge, that Priya, Tiyasha, and Jaya are three girls, among them, Priya can cook. Let's try to write these facts in a more generic way as shown below
- girl(priya).
- girl(tiyasha).
- girl(jaya).
- can_cook(priya).

Advantages :

1. Easy to build database. Doesn’t need a lot of programming effort.
2. Pattern matching is easy. Search is recursion based.
3. It has built in list handling. Makes it easier to play with any algorithm involving lists.

Disadvantages :

1. LISP (another logic programming language) dominates over prolog with respect to I/O features.
2. Sometimes input and output is not easy.

Applications :

Prolog is highly used in artificial intelligence(AI). Prolog is also used for pattern matching over natural language parse trees.

Syntax and semantics

- In Prolog, program logic is expressed in terms of relations, and a computation is initiated by running a query over these relations.
- Relations and queries are constructed using Prolog's single data type, the term.
- Relations are defined by clauses. Given a query.
- Prolog (and other logic programming languages) particularly useful for database, symbolic mathematics, and language parsing applications. Because Prolog allows impure predicates, checking the truth value of certain special predicates.

Data types

20AM3602

- Prolog's single data type is the term. Terms are either **atoms, numbers, variables or compound terms**.
- An **atom** is a general-purpose name with no inherent meaning. Examples of atoms include x, red, 'Taco', and 'some atom'.
- **Numbers** can be floats or integers.
- **Variables** are denoted by a string consisting of letters, numbers and underscore characters, and beginning with an upper-case letter or underscore. Variables closely resemble variables in logic in that they are placeholders for arbitrary terms.
- A **compound term** is composed of an atom called a "functor" and a number of "arguments", which are again terms. Compound terms are ordinarily written as a functor followed by a comma-separated list of argument terms, which is contained in parentheses.
- The number of arguments is called the term's arity. An atom can be regarded as a compound term with arity zero. An example of a compound term is person_friends(zelda,[tom,jim]).

Special cases of compound terms:

- A **List** is an ordered collection of terms. It is denoted by square brackets with the terms separated by commas, or in the case of the empty list, by []. For example, [1,2,3] or [red,green,blue].
- **Strings:** A sequence of characters surrounded by quotes is equivalent to either a list of (numeric) character codes, a list of characters (atoms of length 1), or an atom depending on the value of the Prolog flag double_quotes. For example, "to be, or not to be".

20AM3602

- Rules and facts
- Prolog programs describe relations, defined by means of clauses. Pure Prolog is restricted to Horn clauses. There are two types of clauses: facts and rules. A rule is of the form
- Head :- Body.
- and is read as "Head is true if Body is true". A rule's body consists of calls to predicates, which are called the rule's goals. The built-in logical operator ,/2 (meaning an arity 2 operator with name ,) denotes conjunction of goals, and ;/2 denotes disjunction. Conjunctions and disjunctions can only appear in the body, not in the head of a rule.
- Clauses with empty bodies are called facts. An example of a fact is:
- cat(tom).
- which is equivalent to the rule:
- cat(tom) :- true.

The built-in predicate true/0 is always true.
Given the above fact, one can ask:
is tom a cat?
?- cat(tom).
Yes
what things are cats?
?- cat(X).
X = tom
Clauses with bodies are called rules. An example of a rule is:
animal(X) :- cat(X).
If we add that rule and ask what things are animals?
?- animal(X).
X = tom

Example

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

Check for the answers

?- sibling(sally, erica).

Answer….?

?- father_child(Father, Child).

Answer….?

20AM3602

Key Features of prolog

1. Unification : The basic idea is, can the given terms be made to represent the same structure.
2. Backtracking : When a task fails, prolog traces backwards and tries to satisfy previous task.
3. Recursion : Recursion is the basis for any search in program.

20AM3602

Unification in Prolog

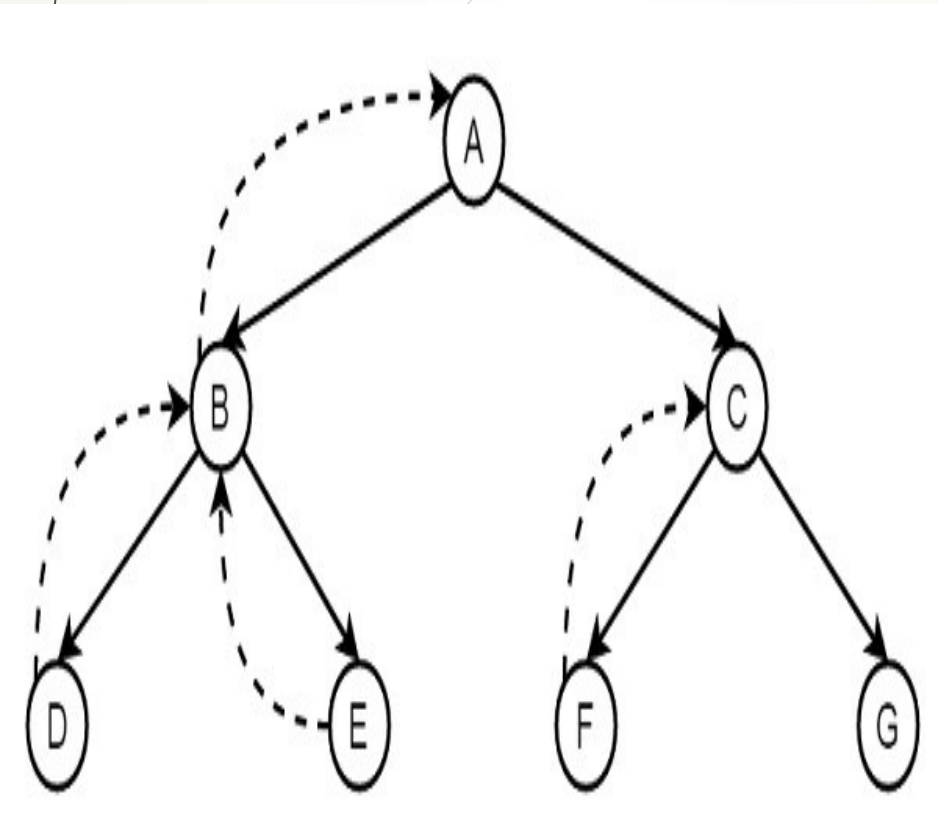
- We will give a goal to evaluate and Prolog will work through the clauses in the database. In this, Prolog attempts to match the goal with each clause. The matching process works from left to right. The goal will fail if no match is found. If a match is found, the action will take.
- Prolog uses the unification technique, and it is a very general form of matching technique. In unification, one or more variables being given value to make the two call terms identical. This process is called binding the variables to values.
- For example: Prolog can unify the terms cat(A), and cat(mary) by binding variable A to atom mary that means we are giving the value mary to variable A. Prolog can unify person(Kevin, dane) and person(L, S) by binding L and S to atom kevin and dane, respectively.

- A successful unification is shown by the following example, and it involves repeated variables.
- pred3(A, A, male)
- pred3(canada, canada, X)
- The variable A bound to atom canada and variable X bound to atom male, so it will succeed.
- The following example describes a repeated variable in one of the arguments in the compound term.
- pred(male, female, mypred(A, A, B))
- pred(L, S, mypred(no, yes, maybe))
- It will fail.

20AM3602

Backtracking in Prolog

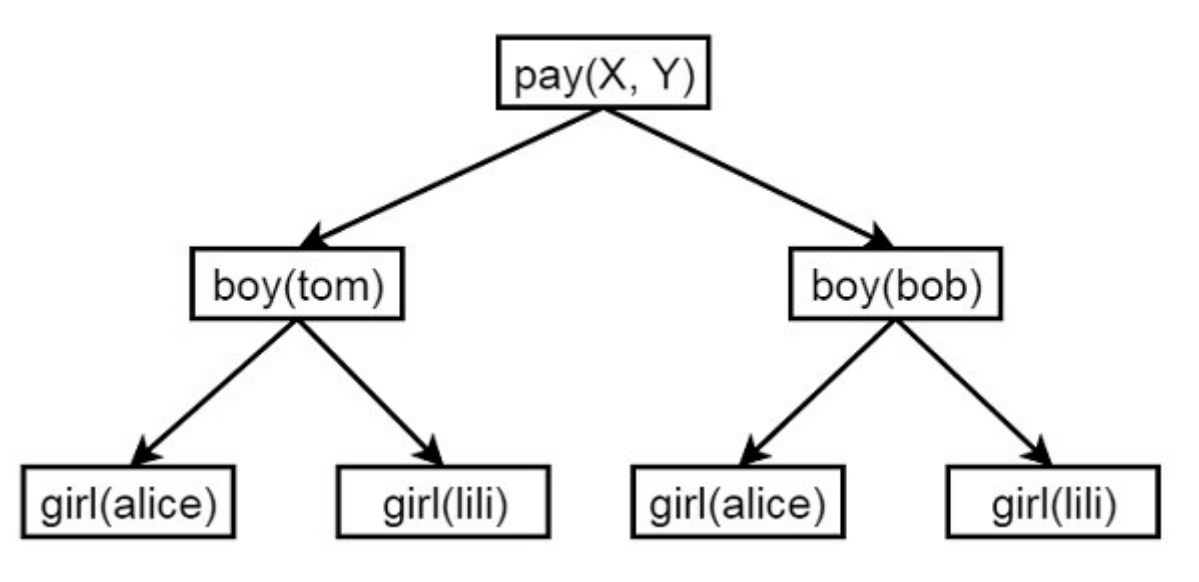
- Backtracking is a procedure, in which prolog searches the truth value of different predicates by checking whether they are correct or not. The backtracking term is quite common in algorithm designing, and in different programming environments. In Prolog, until it reaches proper destination, it tries to backtrack. When the destination is found, it stops.
- Let us see how backtracking takes place using one tree like structure −



- Suppose A to G are some rules and facts. We start from A and want to reach G. The proper path will be A-C-G, but at first, it will go from A to B, then B to D. When it finds that D is not the destination, it backtracks to B, then go to E, and backtracks again to B, as there is no other child of B, then it backtracks to A, thus it searches for G, and finally found G in the path A-C-G. (Dashed lines are indicating the backtracking.) So when it finds G, it stops.

How Backtracking works?

- Now we know, what is the backtracking in Prolog. Let us see one example,
- Now, consider a situation, where two people X and Y can pay each other, but the condition is that a boy can pay to a girl, so X will be a boy, and Y will be a girl. So for these we have defined some facts and rules −



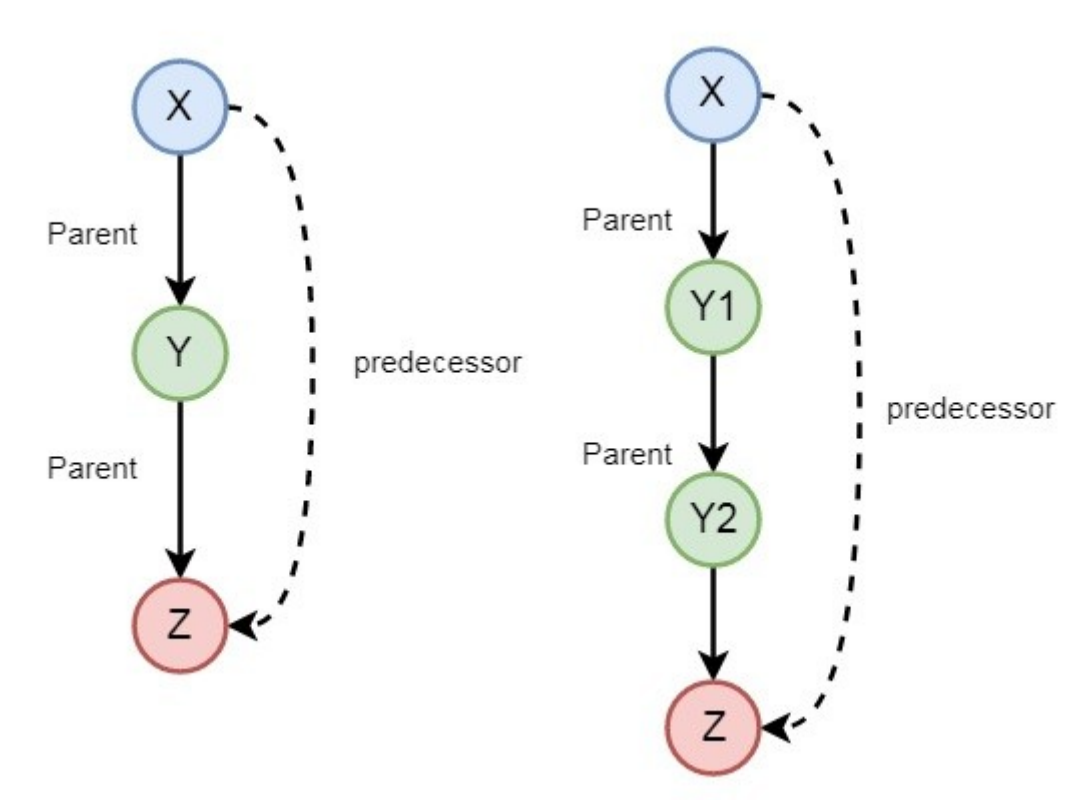

```
boy(tom).
boy(bob).
girl(alice).
girl(lili).

pay(X,Y) :- boy(X), girl(Y).


?- pay (X,Y).
```

Check the output (use next & check)

20AM3602

Prolog - Recursion and Structures

- Recursion
- Recursion is a technique in which one predicate uses itself (may be with some other predicates) to find the truth value.
- Let us understand this definition with the help of an example −
- is_digesting(X,Y) :- just_ate(X,Y).
- is_digesting(X,Y) :-just_ate(X,Z),is_digesting(Z,Y).
- So this predicate is recursive in nature. Suppose we say that just_ate(deer, grass), it means is_digesting(deer, grass) is true. Now if we say is_digesting(tiger, grass), this will be true if is_digesting(tiger, grass) :- just_ate(tiger, deer), is_digesting(deer, grass), then the statement is_digesting(tiger, grass) is also true.



- So we can understand the predecessor relationship is recursive. We can express this relationship using the following syntax −
- predecessor(X, Z) :- parent(X, Z).
- predecessor(X, Z) :- parent(X, Y),predecessor(Y, Z).

20AM3602

Example for Recursion in Prolog

- is_digesting(X,Y) :- just_ate(X,Y).
- is_digesting(X,Y) :-
- just_ate(X,Z),
- is_digesting(Z,Y).

- just_ate(mosquito,blood(john)).
- just_ate(frog,mosquito).
- just_ate(stork,frog).

Query:

?- is_digesting(stork,mosquito).

?- just_ate(stork,Z), *is_digesting*(Z,mosquito).

?-

*Hint https://lpn.swi-prolog.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse9

Find some more examples.

Abstract Data Types in Prolog

- ADT
  - A set of operations
  - No data structure specified
- ADT building components
  - Recursion, list, and pattern matching
  - List handing and recursive processing are hidden in ADTs
- Typical ADTs
  - Stack, Queue, Set, Priority Queue

The ADT Stack

- Characteristics
  - LIFO (Last-in-first-out)
- Operations
  - Empty test
  - Push an element onto the stack
  - Pop the top element from the stack
  - Peek to see the top element
  - Member_stack to test members
  - Add_list to add a list of elements to the Stack
- Implementation
  - empty_stack([]).
  - stack(Top, Stack, [Top|Stack]).
    - Push – bind first two elements and free the third element
    - Pop – bind the third element and free the first two
    - Peek – same as Pop but keep the third element
  - member_stack(Element, Stack) :- member(Element, Stack).
  - add_list_to_stack(List, Stack, Result):-append(List, Stack, Result).

Example program in ADT

```
empty_stack([]).
stack(Top, Stack, [Top|Stack]).
member_stack(E, Stack) :-
                                    member(E, Stack).
add_list_to_stack(L, S, R) :-
                                    append(L, S, R).
reverse_print_stack(S) :-
                                    empty_stack(S), !.
reverse_print_stack(S) :-
                                    stack(H, T, S),
                                    reverse_print_stack(T),
                                    write(H), nl.
```

- ?- reverse_print_stack([1, 2, 3, 4, 5]).

20AM3602

The ADT Queue

- Examine queue processing
- Define a queue abstract data type
- Demonstrate how a queue can be usedto solve problems
- Examine various queueimplementations
- Compare queue implementations

Operations on a Queue

Operation	Description
dequeue	Removes an element from the front of the queue
enqueue	Adds an element to the rear of the queue
first	Examines the element at the front of the queue
isEmpty	Determines whether the queue is empty
Size	Determines the number of elements in the queue
toString	Returns a string representation of the queue

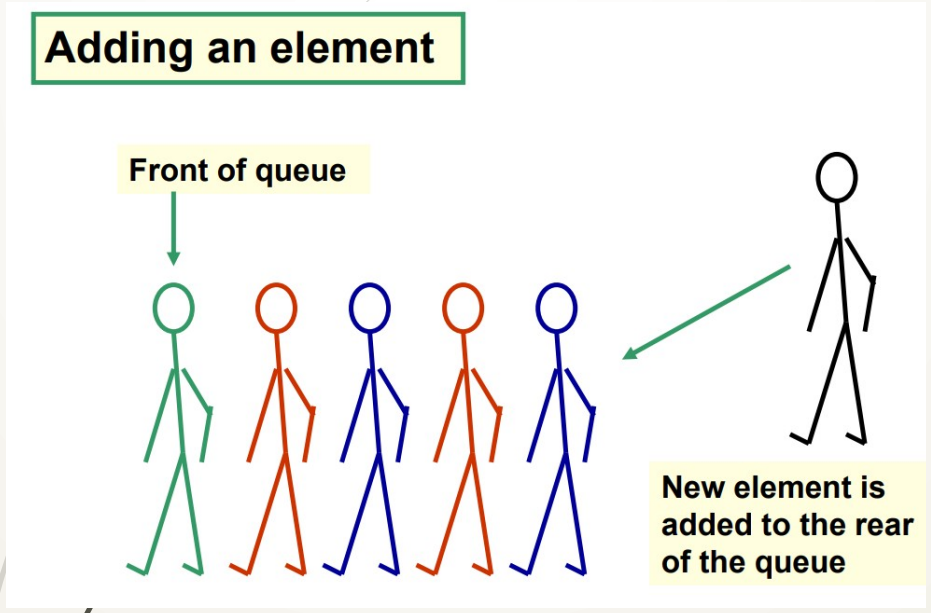
Queues

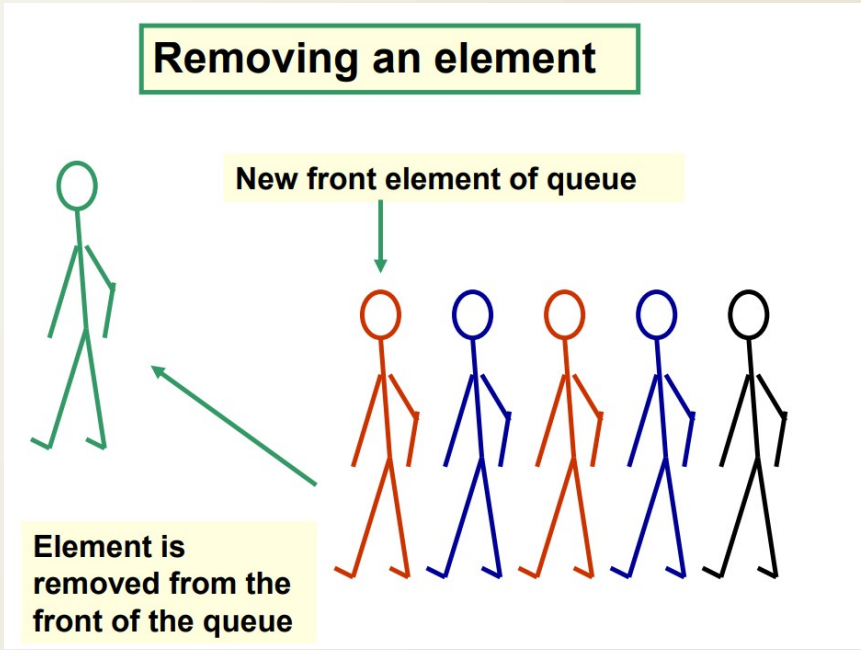
- Queue: a collection whose elements are added at one end (the rear or tail of the queue) and removed from the other end (the front or head of the queue)
- A queue is a FIFO (first in, first out) data structure

Any waiting line is a queue:

- The check-out line at a grocery store
- The cars at a stop light
- An assembly line

Conceptual View of a Queue





```
empty_queue([]).
enqueue(E, [], [E]).
enqueue(E, [H | T], [H | Tnew]) :-
enqueue(E, T, Tnew).
dequeue(E, [E | T], T).
dequeue(E, [E | _], _).
member_queue(Element, Queue) :-
member(Element, Queue).
add_list_to_queue(List, Queue, Newqueue) :-
append(Queue, List, Newqueue).
```

Check the output

```
?- append(X, Y, Z).
?- append(Queue, List, Newqueue).
?- enqueue(X, Y, Z).
```

20AM3602

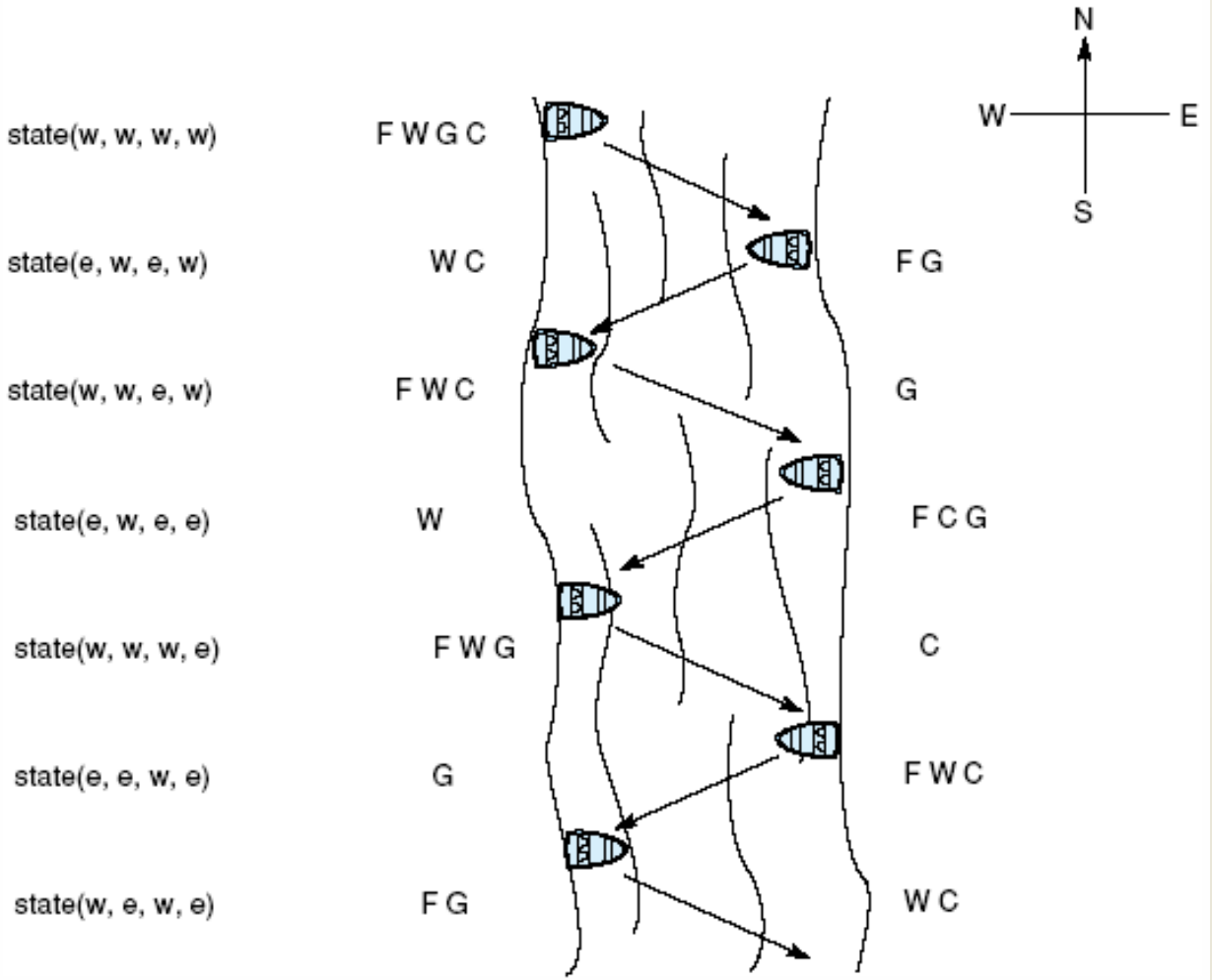
The ADT Priority Queue

- Assignment

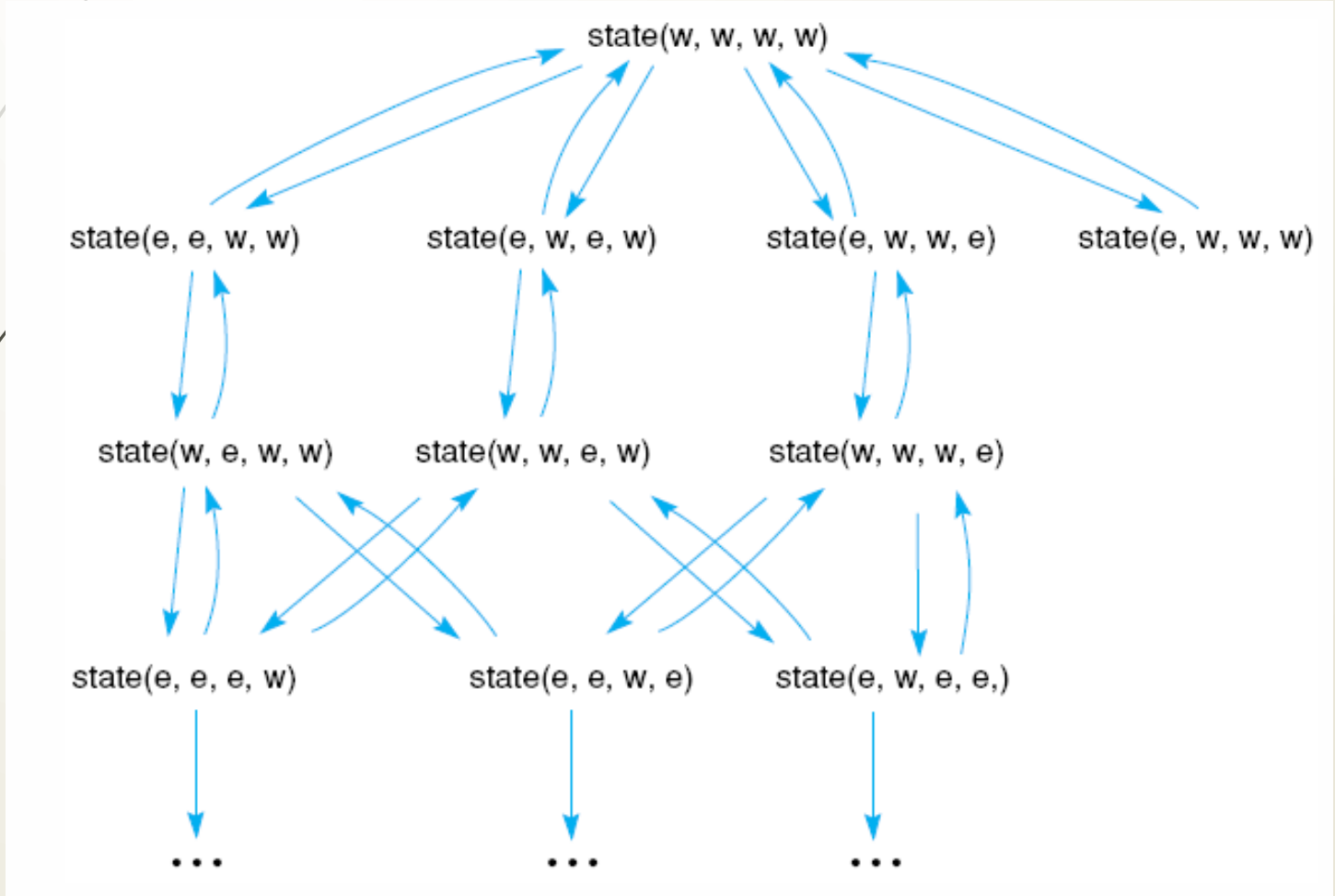
A Production System in Prolog

- Farmer, wolf, goat, and cabbage problem
- A farmer with his wolf, goat, and cabbage come to the edge of a river they wish to cross. There is a boat at the river's edge, but, of course, only the farmer can row. The boat also can carry only two things, including the rower, at a time. If the wolf is ever left alone with the goat, the wolf will eat the goat; similarly if the goat is left alone with the cabbage, the goat will eat the cabbage. Devise a sequence of crossings of the river so that all four characters arrives safely on the other side of the river.
- Representation
- state(F, W, G, C) describes the location of Farmer, Wolf, Goat, and Cabbage
- Possible locations are e for east bank, w for west bank
- Initial state is state(w, w, w, w)
- Goal state is state(e, e, e, e)
- Predicates opp(X, Y) indicates that X and y are opposite sides of the river
- Facts:
- opp(e, w).
- opp(w, e).

Sample crossings for the farmer, wolf, goat, and cabbage problem.



Portion of the state space graph of the farmer, wolf, goat, and cabbage problem, including unsafe states.



Production Rules in Prolog

20AM3602

```
cross(w, e).
cross(e, w).
execmove([F, W, G, C], 0, [NF, NW, NG, NC]) :- F = W,
                                cross(F, NF),
                                cross(W, NW),
                                NG = G,
                                NC = C.
execmove([F, W, G, C], 1, [NF, NW, NG, NC]) :- F = G,
                                cross(F, NF),
                                NW = W,
                                cross(G, NG),
                                NC = C.
execmove([F, W, G, C], 2, [NF, NW, NG, NC]) :- F = C,
                                cross(F, NF),
                                NW = W,
                                NG = G,
                                cross(C, NC).
execmove([F, W, G, C], 3, [NF, NW, NG, NC]) :- cross(F, NF),
                                NW = W,
                                NG = G,
                                NC = C.
answer([e,e,e,e], []).
answer(State, [Move|Moves]) :- execmove(State, Move, [F, W, G, C]),
                                (W \= G; G = F),
                                (G \= C; C = F),
                                answer([F, W, G, C], Moves).

anshelp(S, L, X) :- length(X, L), answer(S, X);
                L2 is L + 1, anshelp(S, L2, X).
getans(S, X) :- anshelp(S, 0, X).
printans([]).
printans([0|T]) :- write('The farmer takes the wolf across.\n'), printans(T).
printans([1|T]) :- write('The farmer takes the goat across.\n'), printans(T).
printans([2|T]) :- write('The farmer takes the cabbage across.\n'), printans(T).
printans([3|T]) :- write('The farmer crosses alone.\n'), printans(T).
```

Question

?- getans([w,w,w,w], X), printans(X).

20AM3602

An Introduction to LISP:

- Lisp is a programming language that has an overall style that is **organized around expressions and functions**.
- Every Lisp procedure is a function, and when called, it returns a data object as its value. It is also commonly referred to as **“functions”** even though they may have side effects.
- Lisp is the second-oldest high-level programming language in the world which is invented by John McCarthy in the year 1958 at the Massachusetts Institute of Technology.

- **https://onecompiler.com/commonlisp**

Features of LISP Programming Language:

- It is a machine-independent language
- It uses iterative design methodology and is easily extensible
- It allows us to create and update the programs and applications dynamically.
- It provides high-level debugging.
- It supports object-oriented programming.
- It supports all kinds of data types like objects, structures, lists, vectors, adjustable arrays, set, trees,hash-tables, and symbols.
- It is an expression-based language
- It can support different decision-making statements like if, when,case, and cond
- It will also support different iterating statements like do, loop,loopfor, dotimes and dolist.
- It will support input and output functions
- By using lisp we can also create our own functions

Benefits of Lisp

- While there are several reasons why the Lisp programming language is still popular after all these years, perhaps the most important is that it's considered to be a relatively simple language to learn. This is probably why it's still popular in academia. Other benefits include the following:

- access to powerful and easy-to-integrate macros;
- the language itself is programmable to meet nearly any need;
- operates on most platforms; and
- many find programming in Lisp to be faster with smaller code footprints.

- Lisp is used within academia for a variety of functions, ranging from basic programming and AI to machine learning and quantum computing. Outside the university walls, Lisp dialects are used by the following:
- Symbolic AI programmers;
- Quantum computing professionals;
- Embedded systems programmers; those seeking a quick scripting language; and small or understaffed programming teams.
- Today, Lisp dialects are used to create code in a variety of use-case scenarios from basic HyperText Markup Language and web-based apps to software that operates and controls mass transit systems, including the London Tube.
- Commercial applications of Common Lisp include Grammarly, which uses AI to analyze text and suggest improvements, and Boeing, which uses a server written in the Lisp variant. Lisp Clojure or ClojureScript users include Amazon, Capital One and Walmart.

20AM3602

Applications Built in LISP

- Large successful applications built in Lisp.
- Emacs
- G2
- AutoCad
- Igor Engraver
- Yahoo Store

- Software download link https://sourceforge.net/projects/clisp/files/latest/download

LISP - Program Structure

- LISP expressions are called symbolic expressions or s-expressions. The s-expressions are composed of three valid objects, atoms, lists and strings.
- Any s-expression is a valid program.
- LISP programs run either on an interpreter or as compiled code.
- The interpreter checks the source code in a repeated loop, which is also called the read-evaluate-print loop (REPL). It reads the program code, evaluates it, and prints the values returned by the program.

- Example 1: (write (+ 7 9 11))
- Example 2: (60 * 9 / 5) + 32 -→ (write(+ (* (/ 9 5) 60) 32))

20AM3602

Evaluation of LISP Programs

Evaluation of LISP programs has two parts −

- Translation of program text into Lisp objects by a reader program
- Implementation of the semantics of the language in terms of these objects by an evaluator program

The evaluation process takes the following steps −

- The reader translates the strings of characters to LISP objects or s-expressions.
- The evaluator defines syntax of Lisp forms that are built from s-expressions. This second level of evaluation defines a syntax that determines which s-expressions are LISP forms.
- The evaluator works as a function that takes a valid LISP form as an argument and returns a value. This is the reason why we put the LISP expression in parenthesis, because we are sending the entire expression/form to the evaluator as arguments.

20AM3602

Basic Building Blocks in LISP

LISP programs are made up of three basic building blocks −

- Atom : An atom is a number or string of contiguous characters. It includes numbers and special characters.
- List : A list is a sequence of atoms and/or other lists enclosed in parentheses.
- String: A string is a group of characters enclosed in double quotation marks.

20AM3602

Search in LISP

- Function SEARCH
- Syntax:

search sequence-1 sequence-2 &key from-end test test-not key start1 start2 end1 end2

- => position

- In LISP, you can perform search operations on lists using various built-in functions and constructs. Here are some examples:

1. member: This function takes an item and a list as arguments, and returns the sublist of the list that starts with the first occurrence of the item, or nil if the item is not found in the list. For example:

- (member 'b '(a b c d)) ; Returns (b c d)
- (member 'e '(a b c d)) ; Returns nil

2. position: This function takes an item and a list as arguments, and returns the index of the first occurrence of the item in the list, or nil if the item is not found in the list. For example:

- (position 'b '(a b c d)) ; Returns 1
- (position 'e '(a b c d)) ; Returns nil

3. find: This function takes a predicate function and a list as arguments, and returns the first element of the list for which the predicate function returns true, or nil if no such element is found. For example:

- (find (lambda (x) (> x 5)) '(2 4 6 8)) ; Returns 6
- (find (lambda (x) (> x 10)) '(2 4 6 8)) ; Returns nil

4. remove-if-not: This function takes a predicate function and a list as arguments, and returns a new list that contains only the elements of the original list for which the predicate function returns true. For example:

- (remove-if-not (lambda (x) (stringp x)) '(a "b" c "d")) ; Returns ("b" "d")
- In addition to these built-in functions, you can also use constructs such as mapcar, reduce, and loop to perform more complex search operations on lists in LISP. By combining these functions and constructs, you can write powerful and flexible search algorithms in LISP.

Pattern matching in LISP

- In LISP, pattern matching is performed using the match macro. The match macro is used to compare a pattern to an expression and bind variables based on the pattern.

- The basic syntax of the match macro is as follows:
- (match expression
- pattern1 expression1
- pattern2 expression2
- ...
- patternN expressionN)

- The match macro will evaluate the expression and then attempt to match it to each pattern in turn. The first pattern that matches the expression will be used to bind the variables in the pattern and execute the associated expression.

- A pattern can consist of a single value or a combination of values. Some examples of pattern matching in LISP include:

```
(match '(1 2 3)
  ((list a b c) (print a) (print b) (print c)))
```

In this example, the pattern (list a b c) matches the expression '(1 2 3)'. The variables a, b, and c are bound to the values 1, 2, and 3 respectively.

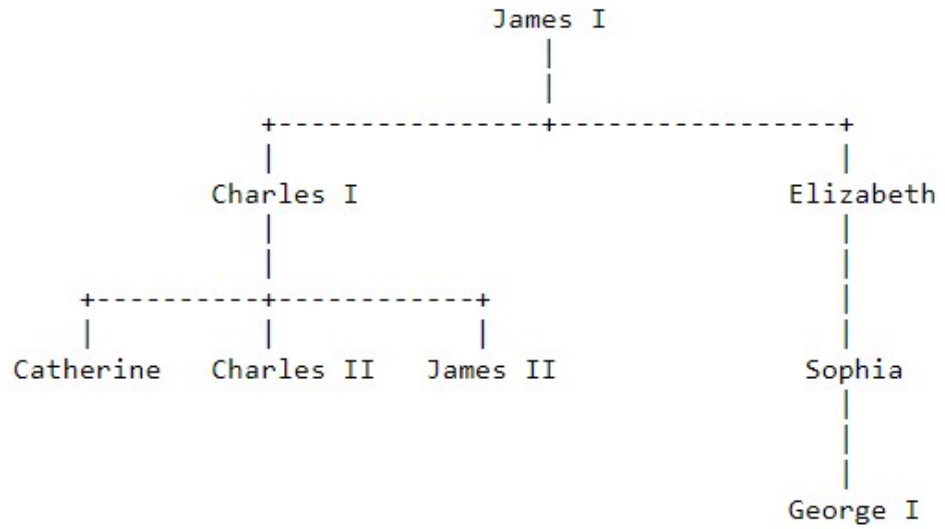
Online software used to execute Lisp

- **REPL.it:** A popular online code editor that supports several programming languages, including LISP. It has a built-in LISP interpreter and allows you to write and execute LISP code with pattern matching.
- **Ideone.com:** A free online compiler and debugging tool that supports several programming languages, including LISP. You can write and execute LISP programs with pattern matching using the online editor.
- **Online LISP Compiler:** A web-based LISP compiler that allows you to write and execute LISP programs with pattern matching. It has a simple and intuitive interface, and supports multiple LISP dialects, including Common Lisp and Scheme.
- **LISPBox:** An online LISP interpreter that allows you to write and execute LISP programs with pattern matching. It has a simple and easy-to-use interface, and supports multiple LISP dialects.

20AM3602

Define macro match

```
(defmacro match (expr &rest clauses)
  `(let ((it ,expr)))
     (catch 'failed
       ,@(mapcar (lambda (clause)
                   (let ((pattern (car clause))
                         (expr (cdr clause)))
                     `(when (and ,(compile pattern 'env) (not (eq it '*failed*)))
                        (return-from match ,@(compile expr 'env))))))
                 clauses)
       '*failed*)))
```



Here are the resultant clauses:

```
male(james1).
male(charles1).
male(charles2).
male(james2).
male(george1).

female(catherine).
female(elizabeth).
female(sophia).

parent(charles1, james1).
parent(elizabeth, james1).
parent(charles2, charles1).
parent(catherine, charles1).
parent(james2, charles1).
parent(sophia, elizabeth).
parent(george1, sophia).
```

- Here is how you would formulate the following queries:

- Was George I the parent of Charles I?
- Query: parent(charles1, george1).
- Who was Charles I's parent?
- Query: parent(charles1,X).
- Who were the children of Charles I?
- Query: parent(X,charles1).
- Now try expressing the following rules:

- M is the mother of X if she is a parent of X and is female
- F is the father of X if he is a parent of X and is male
- X is a sibling of Y if they both have the same parent.

- Furthermore add rules defining:

- "sister", "brother",
- "aunt", "uncle",
- "grandparent", "cousin"

20AM3602

Assignment

- https://staff.fnwi.uva.nl/u.endriss/teaching/prolog/prolog.pdf
- https://en.wikipedia.org/wiki/Prolog
- https://www.javatpoint.com/unification-in-prolog
- https://www.tutorialspoint.com/prolog/prolog_backtracking.htm#:~:text=Backtracking%20is%20a%20procedure%2C%20in,destination%2C%20it%20tries%20to%20backtrack

  .
- http://bkraabel.free.fr/pages/prolog.html
- https://gist.github.com/cflems/0308adef98ad26807ffe9a3bf35310b9 (FCWG Problem analysis)


Thank
You