**Module 4**

**Functional and Logic Languages:- Lambda Calculus, Overview of Scheme, Strictness and Lazy Evaluation, Streams and Monads, Higher-Order Functions, Logic Programming in Prolog, Limitations of Logic Programming.**

**Functional Languages**

The imperative and functional models grew out of work undertaken by mathematicians Alan Turing, Alonzo Church, Stephen Kleene, Emil Post etc.
These individuals developed several very different formalizations of the notion of an algorithm, or *effective procedure*, based on automata, symbolic manipulation, recursive function definitions, and combinatorics.
Turing's model of computing was the *Turing machine*, an automaton reminiscent of a finite or pushdown automaton, but with the ability to access arbitrary cells of an unbounded storage "tape."
Church's model of computing is called the *lambda calculus*.
It is based on the notion of parameterized expressions ;with each parameter introduced by an occurrence of the letter $\lambda$—hence the notation's name.
Lambda calculus was the inspiration for functional programming: one uses it to compute by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions.

A *constructive* proof is the one that shows how to obtain a mathematical object with some desired property, and a *nonconstructive* proof is one that merely shows that such an object must exist, perhaps by contradiction, or counting arguments, or reduction to some other theorem whose proof is nonconstructive.
The logic programmer writes a set of *axioms* that allow the *computer* to discover a constructive proof for each particular set of inputs.

Functional Programming Concepts

*Functional programming* defines the outputs of a program as a mathematical function of the inputs, with no notion of internal state, and thus no side effects.
Miranda, Haskell, pH, Sisal, and Single Assignment C are purely functional.
Features are:
First-class function values and higher-order functions
Extensive polymorphism
List types and operators
Structured function returns
Constructors (aggregates) for structured objects
Garbage collection

A first-class value as one that can be passed as a parameter, returned from a subroutine, or in a language with side effects; assigned into a variable.
A *higher order function* takes a function as an argument, or returns a function as a result.
Polymorphism is important in functional languages because it allows a function to be used on as general a class of arguments as possible.
Lists are important in functional languages because they have a natural recursive definition, and are easily manipulated by operating on their first element and recursively the remainder of the list.
Python and Ruby provide aggregates capable of representing an unnamed functional value ;a *lambda expression*, but few imperative languages are so expressive.
Functional languages tend to employ a garbage-collected heap for *all* dynamically allocated data.

**Lambda Calculus**
Lambda calculus is a *constructive* notation for function definitions.
Any computable function can be written as a lambda expression.
Computation amounts to macro substitution of arguments into the function definition, followed by reduction to simplest form via simple and mechanical rewrite rules.
The order in which these rules are applied captures the distinction between applicative and normal-order evaluation.
Conventions on the use of certain simple functions e.g., the identity function allow selection, structures, and even arithmetic to be captured as lambda expressions.

**An Overview of Scheme**

Scheme implementations employ an interpreter that runs a"read-eval-print" loop.
The interpreter repeatedly reads an expression from standard input –generally typed by the user**Module 4**
**Functional and Logic Languages:- Lambda Calculus, Overview of Scheme, Strictness and Lazy Evaluation, Streams and Monads, Higher-Order Functions, Logic Programming in Prolog, Limitations of Logic Programming.**

**Functional Languages**
The imperative and functional models grew out of work undertaken by mathematicians Alan Turing, Alonzo Church, Stephen Kleene, Emil Post etc.
These individuals developed several very different formalizations of the notion of an algorithm, or *effective procedure*, based on automata, symbolic manipulation, recursive function definitions, and combinatorics.
Turing's model of computing was the *Turing machine*, an automaton reminiscent of a finite or pushdown automaton, but with the ability to access arbitrary cells of an unbounded storage "tape."
Church's model of computing is called the *lambda calculus*.
It is based on the notion of parameterized expressions ;with each parameter introduced by an occurrence of the letter $\lambda$ —hence the notation's name.
Lambda calculus was the inspiration for functional programming: one uses it to compute by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions.

A *constructive* proof is the one that shows how to obtain a mathematical object with some desired property, and a *nonconstructive* proof is one that merely shows that such an object must exist, perhaps by contradiction, or counting arguments, or reduction to some other theorem whose proof is nonconstructive.
The logic programmer writes a set of *axioms* that allow the *computer* to discover a constructive proof for each particular set of inputs.

Functional Programming Concepts
*Functional programming* defines the outputs of a program as a mathematical function of the inputs, with no notion of internal state, and thus no side effects.
Miranda, Haskell, pH, Sisal, and Single Assignment C are purely functional.
Features are:
First-class function values and higher-order functions
Extensive polymorphism
List types and operators
Structured function returns

Constructors (aggregates) for structured objects
Garbage collection

A first-class value as one that can be passed as a parameter, returned from a subroutine, or in a language with side effects; assigned into a variable.
A *higher order function* takes a function as an argument, or returns a function as a result.
Polymorphism is important in functional languages because it allows a function to be used on as general a class of arguments as possible.
Lists are important in functional languages because they have a natural recursive definition, and are easily manipulated by operating on their first element and recursively the remainder of the list.
Python and Ruby provide aggregates capable of representing an unnamed functional value ;a *lambda expression*, but few imperative languages are so expressive.
Functional languages tend to employ a garbage-collected heap for *all* dynamically allocated data.
**Lambda Calculus**
Lambda calculus is a *constructive* notation for function definitions.
Any computable function can be written as a lambda expression.
Computation amounts to macro substitution of arguments into the function definition, followed by reduction to simplest form via simple and mechanical rewrite rules.
The order in which these rules are applied captures the distinction between applicative and normal-order evaluation.
Conventions on the use of certain simple functions e.g., the identity function allow selection, structures, and even arithmetic to be captured as lambda expressions.

**An Overview of Scheme**

Scheme implementations employ an interpreter that runs a"read-eval-print" loop.
The interpreter repeatedly reads an expression from standard input –generally typed by the user, evaluates that expression, and prints the resulting value.
If the user types
(+ 3 4)
the interpreter will print
7
To save the programmer the need to type an entire program verbatim at the keyboard, most Scheme implementations provide a load function that reads and evaluates input from a file:
(load "my_Scheme_program")
Scheme (like all Lisp dialects) uses *Cambridge Polish* notation for expressions.
Parentheses indicate a function application.
Extra parentheses change the semantics of Lisp/Scheme programs.
(+ 3 4) ⇒ 7
((+ 3 4)) ⇒ *error*
Here the ⇒ means "evaluates to."
This symbol is not a part of the syntax of Scheme itself. _
One can prevent the Scheme interpreter from evaluating a parenthesized expression by *quoting* it:
(quote (+ 3 4)) ⇒ (+ 3 4)
Though every expression has a type in Scheme, that type is generally not determined until run time.
The expression,
(if (> a 0) (+ 2 3) (+ 2 "foo"))
will evaluate to 5 if a is positive, but will produce a run-time type clash error if a is negative or zero.

(define min (lambda (a b) (if (< a b) a b)))
The expression (min 123 456) will evaluate to 123;
User-defined functions can implement their own type checks using predefined *type predicate* functions:
(boolean? x) ; is x a Boolean?
(char? x) ; is x a character?

A *symbol* in Scheme is comparable to what other languages call an identifier.
Identifiers are permitted to contain a wide variety of punctuation marks:
(symbol? 'x$_%:&=*!) =⇒ #t

To create a function in Scheme one evaluates a *lambda expression*:
(lambda (x) (* x x)) ⇒ *function*
Scheme differentiates between functions and so-called *special forms* -lambda among them, which resemble functions but have special evaluation rules.
The value of the last expression -most often there *is* only one, becomes the value returned by the function:
((lambda (x) (* x x)) 3) ⇒ 9
Simple conditional expressions can be written using if:
If expressions
(if (< 2 3) 4 5) ⇒ 4
(if #f 2 3) ⇒ 3

Bindings
Names can be bound to values by introducing a nested scope:

```
(let ((a 3)
(b 4)
(square (lambda (x) (* x x)))
(plus +))
(sqrt (plus (square a) (square b))))      ⇒ 5.0
```

The special form let takes two or more arguments.
Scheme provides a special form called define that has the side effect of creating a global binding for a name:

```
(define hypot
(lambda (a b)
(sqrt (+ (* a a) (* b b)))))
(hypot 3 4)                ⇒ 5
```

Lists and Numbers
The three most important are car, which returns the head of a list, cdr ("coulder"), which returns the rest of the list (everything after the head), and cons, which joins a head to the rest of a list:
(car '(2 3 4)) ⇒ 2
(cdr '(2 3 4)) ⇒ (3 4)
(cons 2 '(3 4)) = (2 3 4)
The notation '(2 3 4) indicates a *proper* list, in which the final element is the empty list:
(cdr '(2)) ⇒ ()
(cons 2 3) ⇒ (2 . 3) ; an improper list

EqualityTesting and Searching

eqv? Performs a shallow comparison, while equal? performs a deep (recursive) comparison, using eqv? at the leaves.
The functions memq, memv, and member take an element and a list as argument, and return the longest suffix of the list (if any) beginning with the element:

```
(memq ’z ’(x y z w)) ⇒ (z w)
(memv ’(z) ’(x y (z) w)) ⇒ #f          ; (eq? ’(z) ’(z)) ⇒ #f
(member ’(z) ’(x y (z) w)) ⇒ ((z) w)       ; (equal? ’(z) ’(z)) ⇒ #t
```

The memq, memv, and member functions perform their comparisons using eq?, eqv?, and equal?, respectively.
The functions assq, assv, and assoc search for values in association lists -otherwise known as A-lists.
An A-list is a dictionary implemented as a list of pairs.

Control Flow and Assignment
Cond resembles a more general if. . . elsif. . . else:

```
(cond
((< 3 2) 1)
((< 4 3) 2)
(else 3)) ⇒ 3
```

The arguments to cond are pairs.
They are considered in order from first to last.
Assignment employs the special form set! and the functions set-car! and set-cdr!:

```
(let ((x 2) ; initialize x to 2
(l ’(a b))) ; initialize l to (a b)
(set! x 3) ; assign x the value 3
(set-car! l ’(c d)) ; assign head of l the value (c d)
(set-cdr! l ’(e)) ; assign rest of l the value (e)
```

Delay and force permit the lazy evaluation of expressions.
Call-with-current-continuation (call/cc;) allows the current program counter and referencing environment to be saved in the form of a closure, and passed to a specified subroutine.
Programs as Lists
Lisp and Scheme are *homoiconic*—self-representing.
A parenthesized string of symbols -in which parentheses are balanced is called an *S-expression* regardless of whether we think of it as a program or as a list.
Scheme provides an eval function that can be used to evaluate a list that has been created as a data structure:

```
(define compose
(lambda (f g)
(lambda (x) (f (g x)))))
((compose car cdr) ’(1 2 3)) ⇒ 2
```

Compose takes as arguments a pair of functions f and g.
It returns as result a function that takes as parameter a value x, applies g to it, then applies f, and finally returns the result.
There is an *unevaluated* expression (lambda (x) (f (g x))).
When passed to eval, this list evaluates to the desired function.

```
(eval (list ’lambda ’(x) (list f (list g ’x)))
```

Eval ***and*** Apply

Lisp included a *self-definition* of the language:code for a Lisp interpreter.
The code is based on the functions eval and apply.
Apply, takes two arguments: a function and a list.
It achieves the effect of calling the function, with the elements of the list as arguments.
For *Primitive* special forms, built into the language implementation- lambda, if, define, set!, quote, etc. eval provides a direct implementation.
Formalizing Self-Definition: Self-definition" is that for all expressions *E*, we get the same result by evaluating *E* under the interpreter *I* that we get by evaluating *E* directly.
Consider,
*M*(*I*) =*M*
Suppose let *H*(*F*) = *F*(*I* ) where *F* can be any function that takes a Scheme expression as its argument.
Clearly
*H*(*M*) =*M*
Function *M* is said to be a *fixed point* of *H*.
*H* is well defined we can use it to obtain a rigorous definition of *M.*

Extended Example: DFA Simulation
Consider the simulation of the execution of a DFA (deterministic finite automaton).
We represent a DFA as a list of three items: the start state, the transition function, and a list of final states.
To simulate this machine, we pass it to the function simulate along with an input string.
As it runs, the automaton accumulates as a list a trace of the states through which it has traveled, ending with the symbol accept or reject.
For example, if we type

```
(simulate
zero-one-even-dfa       ; machine description
'(0 1 1 0 1))           ; input string
```

then the Scheme interpreter will print

```
(q0 q2 q3 q2 q0 q1 reject)
```

If we change the input string to 010010, the interpreter will print

```
(q0 q2 q3 q1 q3 q2 q0 accept)
```

Evaluation Order Revisited
One can choose to evaluate function arguments before passing them to a function, or to pass them unevaluated.
The former option is called *applicative-order* evaluation; the latter is called *normal-order* evaluation.
Consider, the following function:

```
(define double (lambda (x) (+ x x)))
```

Under normal-order evaluation we would have

```
(double (* 3 4))
⇒ (+ (* 3 4) (* 3 4))
⇒ (+ 12 (* 3 4))
⇒ (+ 12 12)
⇒ 24
```

Normal order causes us to evaluate (* 3 4) twice.
Special forms and functions are known as *expression types* in Scheme.
Some expression types are *primitive*, in the sense that they must be built into the language implementation.
Others are *derived*; they can be defined in terms of primitive expression types.

In an eval/apply–based interpreter, primitive special forms are built into eval; primitive functions are recognized by apply.
Syntax-rules, can be used to create derived special forms.
These can then be bound to names with define-syntax and let-syntax.
Derived special forms are known as *macro*s in Scheme, but unlike most other macros, they are *hygienic*—lexically scoped, integrated into the language's semantics.

**Strictness and Lazy Evaluation**

A sideeffect-free function is said to be *strict* if it is undefined -fails to terminate, or encounters an error; when any of its arguments is undefined.
Such a function can safely evaluate all its arguments, so its result will not depend on evaluation order.
A function is said to be *nonstrict* if it does not impose this requirement—that is, if it is sometimes defined even when one of its arguments is not.
A *language* is said to be strict if it is defined in such a way that functions are always strict.
ML and Scheme are strict.
Miranda and Haskell are nonstrict.
*Lazy evaluation* gives us the advantage of normal-order evaluation -not evaluating unneeded subexpressions while running within a constant factor of the speed of applicative-order evaluation for expressions in which everything is needed.
The trick is to tag every argument internally with a"memo"that indicates its value, if known.
(double (* 3 4)) will be compiled as (double (f)), where f is a hidden closure with an internal side effect:

```
(define f
(lambda ()
(let ((done #f)       ; memo initially unset
(memo '())
(code (lambda () (* 3 4))))
(if done memo            ; if memo is set, return it
(begin
(set! memo (code))        ; remember value
memo)))))                ; and return it
```

Here (* 3 4) will be evaluated only once.
Lazy evaluation is used for all arguments in Miranda and Haskell.
It is available in Scheme through explicit use of delay and force.
Lazy evaluation is sometimes said to employ "call-by-need."
The principal problem with lazy evaluation is its behavior in the presence of side effects.
If an argument contains a reference to a variable that may be modified by an assignment, then the value of the argument will depend on whether it is evaluated before or after the assignment.
Scheme requires that every use of a delay-ed expression be enclosed in force, making it relatively easy to identify the places where side effects are an issue.

**I/O: Streams and Monads**

Side effects can be found in traditional I/O, including the builtin functions read and display of Scheme: read will generally return a different value every time it is called, and multiple calls to display.
Though they never return a value, they must occur in the proper order if the program is to be considered correct.
One way to avoid these side effects is to model input and output as *streams*—unbounded-length lists whose elements are generated lazily.
If we model input and output as streams, then a program takes the form

```
(define output (my_prog input))
```

When it needs an input value, function my_prog forces evaluation of the car of input, and passes the cdr on to the rest of the program.
The language implementation repeatedly forces evaluation of the car of output, prints it, and repeats:

```
(define driver (lambda (s)
(if (null? s) '() ; nothing left
(display (car s))
(driver (cdr s)))))
(driver output)
```

Lazy evaluation would *force* things to happen in the proper order.
The car of output is the first prompt.
The cadr of output is the first square, a value that requires evaluation of the car of input.
The caddr of output is the second prompt.
Recent versions of Haskell employ a more general concept known as *monads*.
Monads are drawn from a branch of mathematics known as *category theory*, but one doesn't need to understand the theory to appreciate their usefulness in practice.
In Haskell, monads are essentially a clever use of higher-order functions, coupled with a bit of syntactic sugar, that allow the programmer to chain together a sequence of *actions* -function calls that have to happen in order.
The following code calls random twice to illustrate its interface;

```
twoRandomInts gen = let
(rand1, gen2) = random gen
(rand2, gen3) = random gen2
in ([rand1, rand2], gen3)
```

gen2, one of the return values from the first call to random, has been passed as an argument to the second call.
Then gen3, one of the return values from the second call, is returned to main, where it could, if we wished, be passed to another function.
This is particularly complicated for deeply nested functions.
Monads provide a more general solution to the problem of threading mutable state through a functional program.
We use Haskell's standard IO monad, which includes a random number generator:

```
twoMoreRandomInts = do
rand1 <- randomIO
rand2 <- randomIO
return [rand1, rand2]
```

The type of the twoMoreRandomInts function has become IO [Integer].
This identifies it as an *IO action*—a function that -in addition to returning an explicit list of integers invisibly accepts and returns the state of the IO monad -including the standard RNG.
The type of randomIO is IO Integer.
The return operator in twoMoreRandomInts packages an explicit return value -in our case, a two-element list together with the hidden state, to be returned to the caller.
The IO monad, however, is *abstract* : only part of its state is defined in library header files; the rest is implemented by the language run-time system.
This is unavoidable, because, in effect, *the hidden state of the* IO *monad encompasses the real world*.
IO state hiding means that a value of type IO T is permanently tainted: it can never be extracted from the monad to produce a "pure T."
Given putChar, we can define putStr:

```
putStr :: String -> IO ()
putStr s = sequence_ (map putChar s)
```

The result of map putChar s is a list of actions, each of which prints a character: it has type [IO ()].

The bulk of the program—both the computation of values *and the determination of the order in which I/O actions should occur*—is purely functional.

For a program whose I/O can be expressed in terms of streams, the top-level structure may consist of a single line:

```
main = interact my_program
```

The library function interact is of type (String -> String) -> IO ().

**Higher-Order Functions**

A function is said to be a *higher-order function* -also called a *functional form;* if it takes a function as an argument, or returns a function as a result.

Examples of higher-order functions: call/cc, for-each, compose and apply.

Map takes  as argument a function and a *sequence* of lists.

Map calls its function argument on corresponding sets of elements from the lists:

```
(map * '(2 4 6) '(3 5 7)) ⇒ (6 20 42)
```

Where for-each is executed for its side effects, and has an implementation dependent return value, map is purely functional: it returns a list composed of the values returned by its function argument.

We would be able to "fold" the elements of a list together, using an associative binary operator:

```
(define fold (lambda (f i l)
(if (null? l) i             ; i is commonly the identity element for f
(f (car l) (fold f i (cdr l))))))
```

Now (fold + 0 '(1 2 3 4 5)) gives us the sum of the first five natural numbers.

One of the most common uses of higher-order functions is to build new functions from existing ones:

```
(define total (lambda (l) (fold + 0 l)))
(total '(1 2 3 4 5)) ⇒ 15
(define total-all (lambda (l)
(map total l)))
(total-all '((1 2 3 4 5)
(2 4 6 8 10)
(3 6 9 12 15))) ⇒ (15 30 45)
```

Currying

A common operation, named for logician Haskell Curry, is to replace a multiargument function with a function that takes a single argument and returns a function that expects the remaining arguments:

```
(define curried-plus (lambda (a) (lambda (b) (+ a b))))
((curried-plus 3) 4) ⇒ 7
(define plus-3 (curried-plus 3))
(plus-3 4) ⇒ 7
```

Among other things, currying gives us the ability to pass a "partially applied" function to a higher-order function.

We can write a general-purpose function that "curries" its (binary) function argument:

```
(define curry (lambda (f) (lambda (a) (lambda (b) (f a b)))))
(((curry +) 3) 4)               ⇒ 7
(define curried-plus (curry +))
```

Consider the following function in ML:

```
fun plus (a, b) : int = a + b;
==> val plus = fn : int * int -> int
```

The ML definition says that all functions take a *single* argument.
What we have declared is a function that takes a two-element *tuple* as argument.
To call plus, we juxtapose its name and the tuple that is its argument:

```
plus (3, 4);
==> val it = 7 : int
```

Now consider the definition of a curried function in ML:

```
fun curried_plus a = fn b : int => a + b;
==> val curried_plus = fn : int -> int -> int
```

Note the type of curried_plus: int -> int -> int groups implicitly as int ->(int -> int).
ML's syntax for function calls—juxtaposition of function and argument—makes the use of a curried function more intuitive and convenient than it is in Scheme:

```
curried_fold plus 0 [1, 2, 3, 4, 5]; (* ML *)
(((curried_fold +) 0) '(1 2 3 4 5)) ; Scheme
```

**Logic Languages**

Prolog and other logic languages are based on *first-order predicate calculus*

Logic Programming Concepts

Logic programming systems allow the programmer to state a collection of *axioms* from which theorems can be proven.
The user of a logic program states a theorem, or *goal*, and the language implementation attempts to find a collection of axioms and inference steps- including choices of values for variables that together imply the goal.
In almost all logic languages, axioms are written in a standard form known as a *Horn clause*.
A Horn clause consists of a *head*, or *consequent* term *H*, and a *body* consisting of terms $B_i$ :

$H \leftarrow B_1, B_2, \ldots, B_n$

The semantics of this statement are that when the *Bi* are all true,we can deduce that *H* is true as well.
We say "*H*, if $B_1$, $B_2$, . . . , and $B_n$."
A logic programming system combines existing statements, canceling like terms, through a process known as *resolution*.
If we know that *A* and *B* imply *C*, for example, and that *C* implies *D*, we can deduce that *A* and *B* imply *D*:

$C \leftarrow A, B$
$D \leftarrow C$
$D \leftarrow A, B$

In general, terms like *A*, *B*, *C*, and *D* may consist not only of constants -"Rochester is rainy", but also of *predicates* applied to *atoms* or to *variables*:

rainy(Rochester), rainy(Seattle), rainy(*X*).

During resolution, free variables may acquire values through *unification* with expressions in matching terms, much as variables acquire types in ML

flowery(*X*) ← rainy(*X*)
rainy(Rochester)
flowery(Rochester)

**Prolog**

A Prolog interpreter runs in the context of a *database* of *clauses* (Horn clauses) that are assumed to be true.
Each clause is composed of *terms*, which may be constants, variables, or *structures*.

Atoms in Prolog are similar to symbols in Lisp.
Lexically, an atom looks like an identifier beginning with a lowercase letter, a sequence of “punctuation”characters, or a quoted character string:

```
foo         my_Const    +      ’Hi, Mom’
```

A variable looks like an identifier beginning with an uppercase letter:

```
Foo     My_var     X
```

Variables can be *instantiated* to (i.e., can take on) arbitrary values at run time as a result of unification.
Structures consist of an atom called the *functor* and a list of arguments:

```
rainy(rochester)
teaches(scott, cs254)
bin_tree(foo, bin_tree(bar, glarch))
```

We use the term“predicate” to refer to the combination of a functor and an “arity” -number of arguments.
The predicate rainy has arity 1.
The predicate teaches has arity 2. _
The clauses in a Prolog database can be classified as *facts* or *rules*, each of which ends with a period.
A fact is a Horn clause without a right-hand side.
It looks like a single term -the implication symbol is implicit:

```
rainy(rochester).
```

A rule has a right-hand side:

```
snowy(X) :- rainy(X), cold(X).
```

The token :- is the implication symbol; the comma indicates “and.”
Variables that appear in the head of a Horn clause are universally quantified: for all X, X is snowy if X is rainy and X is cold. _
It is also possible to write a clause with an empty left-hand side.
Such a clause is called a *query*, or a *goal*.
Queries are entered with a special ?- version of the implication symbol.
If we were to type the following:

```
rainy(seattle).
rainy(rochester).
?- rainy(C).
```

the Prolog interpreter would respond with

```
C = seattle
```

If we want to find all possible solutions, we can ask the interpreter to continue by typing a semicolon:

```
C = seattle ;
C = rochester
```

If we type another semicolon, the interpreter will indicate that no further solutions are possible:

```
C = seattle ;
C = rochester ;
No
```

<u>Resolution and Unification</u>
The *resolution principle*, says that if *C*1 and *C*2 are Horn clauses and the head of *C*1 matches one of the terms in the body of *C*2, then we can replace the term in *C*2 with the body of *C*1.
Consider the following example:

```
takes(jane_doe, his201).
takes(jane_doe, cs254).
takes(ajit_chandra, art302).
takes(ajit_chandra, cs254).
```

classmates(X, Y) :- takes(X, Z), takes(Y, Z).
If we let X be jane_doe and Z be cs254, we can replace the first term on the right-hand side of the last clause with the (empty) body of the second clause, yielding the new rule
classmates(jane_doe, Y) :- takes(Y, cs254).
In other words, Y is a classmate of jane_doe if Y takes cs254.
The pattern-matching process used to associate X with jane_doe and Z with cs254 is known as *unification*.
Variables that are given values as a result of unification are said to be *instantiated*.
The unification rules for Prolog state that:
A constant unifies only with itself.
Two structures unify if and only if they have the same functor and the same arity, and the corresponding arguments unify recursively.
A variable unifies with anything. If the other thing has a value, then the variable is instantiated.
Formal parameter of type int * ’b list, for example, will unify with an actual parameter of type ’a * real list in ML by instantiating ’a to int and ’b to real. _
Equality in Prolog is defined in terms of “unifiability.”
The goal =(A, B) succeeds if and only if A and B can be unified.

```
?- a = a.
Yes % constant unifies with itself
?- a = b.
No % but not with another constant
?- foo(a, b) = foo(a, b).
Yes % structures are recursively identical
```
It is possible for two variables to be unified without instantiating them.
If we type
?- A = B.
the interpreter will simply respond
A = B
Suppose we are given the following rules:
```
takes_lab(S) :- takes(S, C), has_lab(C).
has_lab(D) :- meets_in(D, R), is_lab(R).
```
S takes a lab class if S takes C and C is a lab class.
Moreover D is a lab class if D meets in room R and R is a lab.

Lists

List manipulation is a sufficiently common operation in Prolog to warrant its own notation.
The construct [a, b, c] is syntactic sugar for the structure .(a, .(b, .(c, []))), where [] is the empty list and . is a built-in cons-like predicate.
[a, b, c] could be expressed as [a | [b,c]], [a, b | [c]], or [a, b, c | []].
The vertical-bar notation is particularly handy when the tail of the list is a variable:
```
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).
sorted([]). % empty list is sorted
sorted([A, B | T]) :- A =< B, sorted([B | T]).
```

% compound list is sorted if first two elements are in order and % remainder of list -after first element is sorted
Here =< is a built-in predicate that operates on numbers.
The underscore is a placeholder for a variable that is not needed anywhere else in the clause.
Note that [a, b | c] is the *improper* list .(a, .(b, c)).
Given,

```
append([], A, A).
append([H | T], A, [H | L]) :- append(T, A, L).
```
we can type
```
?- append([a, b, c], [d, e], L).
L = [a, b, c, d, e]
?- append(X, [d, e], [a, b, c, d, e]).
```
<u>Arithmetic</u>

The usual arithmetic operators are available in Prolog, but they play the role of predicates, not of functions.

Thus +(2, 3), which may also be written 2 + 3, is a two argument structure.

It will not unify with 5:

?- (2 + 3) = 5.

No

Prolog provides a built-in predicate, is, that unifies its first argument with the arithmetic value of its second argument:
```
?- is(X, 1+2).
X = 3
?- X is 1+2.
X = 3 % infix is also ok
?- 1+2 is 4-1.
No % first argument (1+2) is already instantiated.
```

<u>Search/Execution Order</u>

We can imagine two principal search strategies:

Start with existing clauses and work forward, attempting to derive the goal.

This strategy is known as *forward chaining*.

Start with the goal and work backward, attempting to "unresolve" it into a set of preexisting clauses. This strategy is known as *backward chaining*.

The Prolog interpreter (or program) explores this tree depth first, from left to right.

It starts at the beginning of the database, searching for a rule *R* whose head can be unified with the top-level goal. It then considers the terms in the body of *R* as subgoals, and attempts to satisfy them, recursively, left to right.

The process of returning to previous goals is known as *backtracking*.

It strongly resembles the control flow of generators in Icon.
```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```

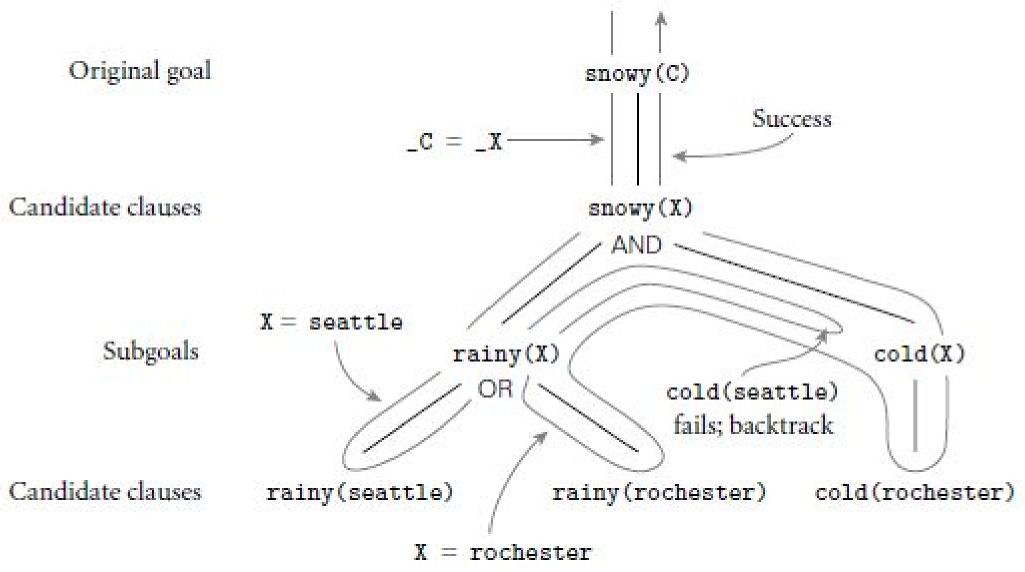

Fig: Backtracking search in Prolog- An OR level consists of alternative database clauses whose head will unify with the subgoal above; one of these must be satisfied.
The notation _C = _X is meant to indicate that while both C and X are uninstantiated, they have been associated with one another in such a way that if either receives a value in the future it will be shared by both.
The binding of X to seattle is broken when we backtrack to the rainy(X) subgoal.
The effect is similar to the breaking of bindings between actual and formal parameters in an imperative programming language.
The interpreter pushes a frame onto its stack every time it begins to pursue a new subgoal *G*.
If *G* succeeds, control returns to the "caller" (the parent in the search tree), but *G*'s frame remains on the stack.
Later subgoals will be given space *above* this dormant frame.
Suppose for example that we have a database describing a directed acyclic graph:

```
edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, X).
path(X, Y) :- edge(Z, Y), path(X, Z).
```

The last two clauses tell us how to determine whether there is a path from node X to node Y.
If we were to reverse the order of the terms on the right-hand side of the final clause, then the Prolog interpreter would search for a node Z that is reachable from X before checking to see whether there is an edge from Z to Y.

```
path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).
```

The interpreter first unifies path(a, a) with the left-hand side of path(X, Y):- path(X, Z), edge(Z, Y).
It then considers the goals on the right-hand side, the first of which (path(X, Z)), unifies with the left-hand side of the very same rule, leading to an infinite regression.

Extended Example: Tic-Tac-Toe
Ordering also allows the Prolog programmer to indicate that certain resolutions are *preferred*, and should be considered before other, "fallback" options.
Consider, for example, the problem of making a move in tic-tac-toe.
Tic-tac-toe is a game played on a 3 × 3 grid of squares.
Two players, X and O, take turns placing markers in empty squares.
Let us number the squares from 1 to 9 in row-major order.

Let us use the Prolog fact x(n) to indicate that player X has placed a marker in square *n*, and o(m) to indicate that player O has placed a marker in square *m*.
Issue a query ?- move(A) that will cause the Prolog interpreter to choose a good square A for the computer to occupy next.
Clearly we need to be able to tell whether three given squares lie in a row.
One way to express this is:

```
ordered_line(1, 2, 3). ordered_line(4, 5, 6).
ordered_line(7, 8, 9). ordered_line(1, 4, 7).

line(A, B, C) :- ordered_line(A, B, C).
line(A, B, C) :- ordered_line(A, C, B).
```

The following rules work well.

```
move(A) :- good(A), empty(A).
full(A) :- x(A).
full(A) :- o(A).
empty(A) :- \+(full(A)).
% strategy:
good(A) :- win(A). good(A) :- block_win(A).
good(A) :- split(A). good(A) :- strong_build(A).
good(A) :- weak_build(A).
```

The initial rule indicates that we can satisfy the goal move(A) by choosing a good, empty square.
The \+ is a built-in predicate that succeeds if its argument -a goal cannot be proven;
If none of these goals can be satisfied, our final, default choice is to pick an unoccupied square, giving priority to the center, the corners, and the sides in that order:

```
good(5).
good(1). good(3). good(7). good(9).
good(2). good(4). good(6). good(8).
```

Imperative Control Flow

Prolog provides the programmer with several explicit control-flow features.
The most important of these features is known as the *cut*.
The cut is a zero-argument predicate written as an exclamation point: !.
As a subgoal it always succeeds, but with a crucial side effect: it commits the interpreter to whatever choices have been made since unifying the parent goal with the lefthand side of the current rule, including the choice of that unification itself.
Definition of list membership:

```
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).
```

If a given atom a appears in list L *n* times, then the goal ?- member(a, L) can succeed *n* times.
This can lead to wasted computation, particularly for long lists, when member is followed by a goal that may fail:

```
prime_candidate(X) :- member(X, candidates), prime(X).
```

Suppose that prime(X) is expensive to compute.
If prime(a) fails, Prolog will backtrack and attempt to satisfy member(a, candidates) again.
We can save substantial time by cutting off all further searches for a after the first is found:

```
member(X, [X | _]) :- !.
member(X, [_ | T]) :- member(X, T).
```

The cut on the right-hand side of the first rule says that if X is the head of L, we should not attempt to unify member(X, L) with the left-hand side of the second rule; the cut commits us to the first rule.

```
member(X, [X | _]).
member(X, [H | T]) :- X \= H, member(X, T).
```
Here X \= H means X and H will not unify; that is, \+(X = H).
It turns out that \+ is actually implemented by a combination of the cut and two other built-in predicates, call and fail:
```
\+(P) :- call(P), !, fail.
\+(P).
```
The call predicate takes a term as argument and attempts to satisfy it as a goal -terms are first-class values in Prolog.
The fail predicate always fails.
Explicit use of the cut may actually make a program *easier* to read.
We can cut off consideration of the others by using the cut:
```
move(A) :- good(A), empty(A), !.
```
Definition of append:
```
append([], A, A).
append([H | T], A, [H | L]) :- append(T, A, L).
```
If we use write append(A, B, L), where L is instantiated but A and B are not, the interpreter will find an A and B for which the predicate is true.
If backtracking forces it to return, the interpreter will look for *another* A and B; append will *generate* pairs on demand.
Idiom—an unbounded generator with a test-cut terminator— is known as *generate-and-test*.
Like the iterative constructs of Scheme, it is generally used in conjunction with side effects.
The built-in predicates consult and reconsult can be used to read database clauses from a file, so they don't have to be typed into the interpreter by hand.
The predicate get attempts to unify its argument with the next *printable* character of input, skipping over ASCII characters with codes below 32.
```
repeat.
repeat :- repeat.
```
Within the above definition of get, backtracking will return to repeat as often as needed to produce a printable character.
Database Manipulation
Clauses in Prolog are simply collections of terms, connected by the built-in predicates :- and ,, both of which can be written in either infix or prefix form.
The structural nature of clauses and database contents implies that Prolog, like Scheme, is *homoiconic*: it can represent itself. It can also modify itself.
A running Prolog program can add clauses to its database with the built-in predicate assert, or remove them with retract:

```
?- assert(rainy(syracuse)).
Yes
?- rainy(X).
X = seattle ;
X = rochester ;
X = syracuse ;
No
?- retract(rainy(rochester)).
Yes
```
There is also a retractall predicate that removes all matching clauses from the database.
Individual terms in Prolog can be created, or their contents extracted, using the built-in predicates functor, arg, and =...
The goal functor(T, F, N) succeeds if and only if T is a term with functor F and arity N:

```
?- functor(foo(a, b, c), foo, 3).
Yes
?- functor(foo(a, b, c), F, N).
F = foo
N = 3
?- functor(T, foo, 3).
T = foo(_10, _37, _24)
```

The goal arg(N, T, A) succeeds if and only if its first two arguments (N and T) are instantiated, N is a natural number, T is a term, and A is the Nth argument of T:

```
?- arg(3, foo(a, b, c), A).
A = c
```

Using =.. and call, the programmer can arrange to pursue (attempt to satisfy) a goal created at run-time:

```
param_loop(L, H, F) :- natural(I), I >= L,
G =.. [F, I], call(G),
I = H, !.
```

The goal param_loop(5, 10, write) will produce the following output:

```
5678910
Yes
```

The only mechanism we have for *perusing* the database i.e., to determine its contents is the built-in search mechanism.

To allow programs to "reason" in more general ways, Prolog provides a clause predicate that attempts to match its two arguments against the head and body of some existing clause in the database:

```
?- clause(snowy(X), B).
B = rainy(X), cold(X) ;
No
```

A clause with no body (a fact) matches the body true:

```
?- clause(rainy(rochester), true).
Yes
```

Note that clause is quite different from call

Various other built-in predicates can also be used to "deconstruct" the contents of a clause.

The var predicate takes a single argument; it succeeds as a goal if and only if its argument is an uninstantiated variable.

The atom and integer predicates succeed as goals if and only if their arguments are atoms and integers, respectively.

The name predicate takes two arguments.

It succeeds as a goal if and only if its first argument is an atom.

## Parts of the logic not covered

Horn clauses do not capture all of first-order predicate calculus.

They cannot be used to express statements whose clausal form includes a disjunction with more than one non-negated term.

In Prolog we use the \+ predicate, but the semantics are not the same.

## Execution Order

While logic is inherently declarative, most logic languages explore the tree of possible resolutions in deterministic order.

Prolog provides a variety of predicates, including the cut, fail, and repeat, to control that execution order.

One must often consider execution order to ensure that a Prolog search will terminate.

Even for searches that terminate, naive code can be *very* inefficient.

## Negation and the "Closed World" Assumption

A collection of Horn clauses, such as the facts and rules of a Prolog database, constitutes a list of things assumed to be true.
It does not include any things assumed to be false.
This reliance on purely “positive” logic implies that Prolog’s \+ predicate is different from logical negation.
Unless the database is assumed to contain *everything* that is true -this is the *closed world assumption*), the goal \+(T) can succeed simply because our current knowledge is insufficient to prove T.
Negation in Prolog occurs *outside* any implicit existential quantifiers on the right-hand side of a rule. Thus ,

```
?- \+(takes(X, his201)).
```

**Logical limitations of Prolog**

Prolog can do many things. But it has four fundamental logical weaknesses:

Prolog doesn't allow "or"d (disjunctive) facts or conclusions--that is, statements that one of several things is true, but you don't know which.
For instance, if a light does not come on when we turn on its switch, we can conclude that either the bulb is burned out or the power is off or the light is disconnected. Prolog doesn't allow "not" (negative) facts or conclusions--that is, direct statements that something is false.

For instance, if a light does not come on when we turn on its switch, but another light in the same room comes on when we turn on *its* switch, we can conclude that it is false that there is a power failure.

Prolog doesn't allow most facts or conclusions having existential quantification--that is, statements that there exists some value of a variable, though we don't know what, such that a predicate expression containing it is true.