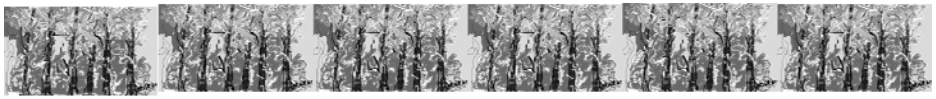


Chapter 10 – Logic Programming
Introduction to Programming Languages
First Edition, 2013

Author: Arvind Bansal

ISBN: 978-146-6565142

Topics Covered

- Logic Programming Fundamentals
- Unification – Bidirectional Information Flow
- Representing Logic Programs
- Abstract Implementation Model
  - AND-OR Tree
  - Warren Abstract Machine
- Programming using Prolog
- Nondeterministic Programming
- Extending Logic Programming Paradigm
- Integration with Other Paradigms
- Summary

Introduction

- Declarative programming paradigm
  - Popular logic programming is based upon first order predicate calculus
  - Applied in expert systems, nondeterministic programming, game playing, intelligent system, genome comparison, natural language processing etc.
- Components of Predicate Calculus
  - Propositional calculus – variableless axioms and complex facts derived by combining axioms using logical operators such as AND, OR, negation
  - Quantification: universal or existential
  - Universal quantification associates a property with elements of a set
  - Existential quantification identifies at least one member in a set with desired properties
- Classification of logic programming
  - Constraint logic programming – handles constraints
  - Deductive logic programming - new axioms using existing axioms
  - Temporal logic programming – incorporates time based reasoning
  - Inductive logic programming – generalizes examples into rules
  - Higher order logic programming – treats relations as arguments

Logic Programming Fundamentals

- Two types of information: Rules and facts
  - Facts are axioms that are treated as true.
  - Rules are used to split a complex query into multiple simpler queries connected through logical AND
  - Rule is of the form Clausehead ← $\text{sungoal}_1 \wedge \ \ldots \wedge \text{subgoal}_N$
  - There can be more than one rules connected through logical-OR used to reduce the same query:
- Example of logic program

  $((\forall X\ \forall Y\ \text{sibling}(X, Y) \leftarrow \exists Z\ \text{parent}(X, Z) \wedge \text{parent}(Y, Z) \ \wedge \ \neg (X == Y)\ ) \vee$
  $(\forall X\ \forall Y\ \text{sibling}(X, Y) \leftarrow \exists Z\ \text{fraternity}(X, Z) \wedge \text{fraternity}(Y, Z) \wedge \neg (X == Y)\ )$
  $\text{parent}(\text{tom}, \text{mary}) \vee \ \text{parent}(\text{neena}, \text{mary}) \vee \ \text{parent}(\text{tom}, \text{john})$
- Corresponding Prolog program

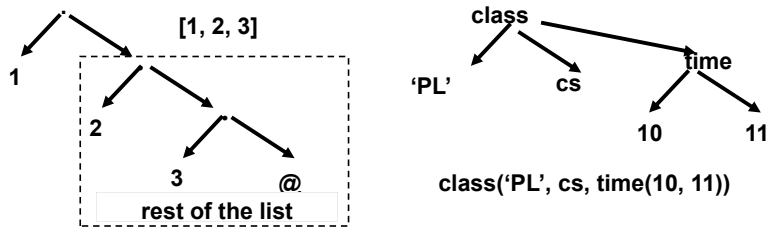
```
sibling(X, Y) :- parent(X, Z), parent(Y, Z), not(X = Y) .
sibling(X, Y) :- fraternity(X, Z), fraternity(Y, Z), not(X = Y).
parent(tom, mary).      parent(neena, mary).      parent(tom, john).
```

Forward vs. Backward Reasoning

- Forward Reasoning system
  - Takes known facts, and uses all the applicable rules eagerly to derive new derived facts.
  - Lot of redundant computations
  - Useful for monitoring system, logical databases, prediction systems
  - Example of forward reasoning system is OPS5
- Backward Reasoning system
  - Starts from a query, and splits query into simpler subqueries using rules and logical operators logical-AND, logical-OR and implication
  - More focused and efficient, uses depth first search for implementation
  - All subqueries must be satisfied by given facts to satisfy top level query
  - Useful in expert systems,
  - Example of backward reasoning system is Prolog

Data Representation

- Traditional data structures are sequences, tuples, tree, and facts
  - List is included within square brackets; tuples within
  - *[a, b, "Arvind Bansal"]* ; *(4, 5, 6)* is a tuple
  - n-ary tree is represented as *<functor-name>*$(Arg_1, \ldots Arg_N)$
  - Constant starts with small letters; variable starts with a capital letter
  - Ground term has all constants; nonground term has atleast one variable
  - [1, 2, 3] can be represented in many ways: [1| [2, 3]], [1, 2 | [3]]
- Modern languages also support rich data structures such as
  - Dynamic arrays, blackboards, associative maps, graphs, sets



a. representation of list **b. representation of an n-ary** tree

Unification

- A means to equate two logical terms
  - Finds out a set of substitutions when applied makes terms equal
  - Does not evaluate any expression, and information flow is two way
  - Used to pass parameters between a query and the LHS of a rule
- Pattern matching and binding in unification
  - Matching is done position by position for
  - Unification performs both pattern matching of constants and binding of variable to a logical subterm
  - Two unbound variables become aliases of each other
- Example
  - b(1, X) = b(Y, b) gives substitution {X/b, Y/1}
  - (a, X, X) = (a, 3, Y) will yield the set of bindings (X/3, and Y/3)
  - b(X, Y, [3, a(d, e)]) = b(N, M, [M, N]) would give the binding set {X/a(d, e), Y/3, M/3, N/a(d, e) }.
  - b(1, 1) = b(X, 4) will fail since constants 1 and 4 do not match

Unification Algorithm

Algorithm unify
Input: Two logical terms: T_1 and T_2
Output: Binding set S;

⊙ means apply substitution
+ means insert in a set
∪ union of two sets
¬ negation

```
{   I1 = T1; I2 = T2; S = { }; unified = true;
    for each position p in I1 and I2
        {  if (is_variable(I1(p) ) ∧ is_variable(I2(p)) then
                S = S + I1(p) ‖ I2(p);
           elseif (is_variable(I1(p) ) ∧ non-ground(I2(p))) then
                S = S + I1(p)/ I2(p);
           elseif (non-ground(I1(p)) ∧ is_variable(I2(p))) then
                S = S + I2(p) / I1(p);
           elseif (ground(I1(p)) ∧ ground(I2(p)) ∧ ¬ (matches(I1(p), I2(p))))
                {unified = false; exit}
           else {Sp = unify(I1(p), I2(p)); S = S ∪ Sp;} % unify non-ground terms
                I1 = I1 ⊙ S;   I2 = I2 ⊙ S;
        }
    if unified then return S;
}
```

Representing Logic Programs

- Logic program is a set of procedures.
- Procedure is a set of rules / facts having the same name and arity.
- A rule is of the form Clausehead :- $Subgoal_1$, …, $Subgoal_N$
- A fact has RHS as trivially true.
- Example

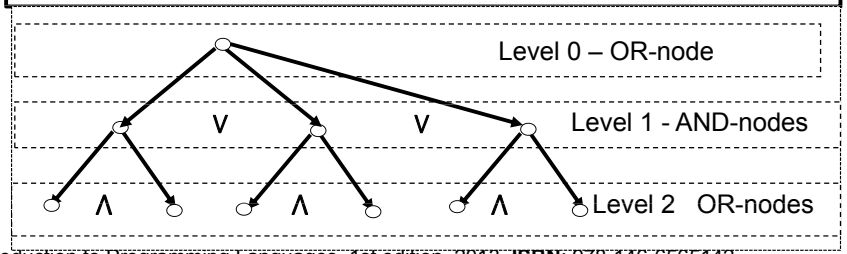
```
factorial(0, 1).  % mode is factorial(+, -)
factorial(M, N) :- M > 0,  M1 is M – 1, factorial(M1, N1), N is M * N1.

fibonacci(0, 1).          fibonacci(1, 1).  % mode is fibonacci(+, -)
fibonacci(M, N) :-
        M > N, M1 is M – 1, M2 is M – 2,
        fibonacci(M1, N1), fibonacci(M2, N2),
        N is N1 + N2.

append([ ], Ys, Ys).  % works in two different modes
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```


Abstract Implementation Model

- Implementation process
  - Map logic program on AND-OR tree
  - map AND-OR tree on a low level abstract machine
  - Popular abstract machine is Warren Abstract Machine (WAM)
- AND-OR tree: a logical tree
  - Has two types of nodes: AND node and OR node that alternate
  - An AND-node is true if all its children are true
  - An OR-node is true if atleast one of its children are true



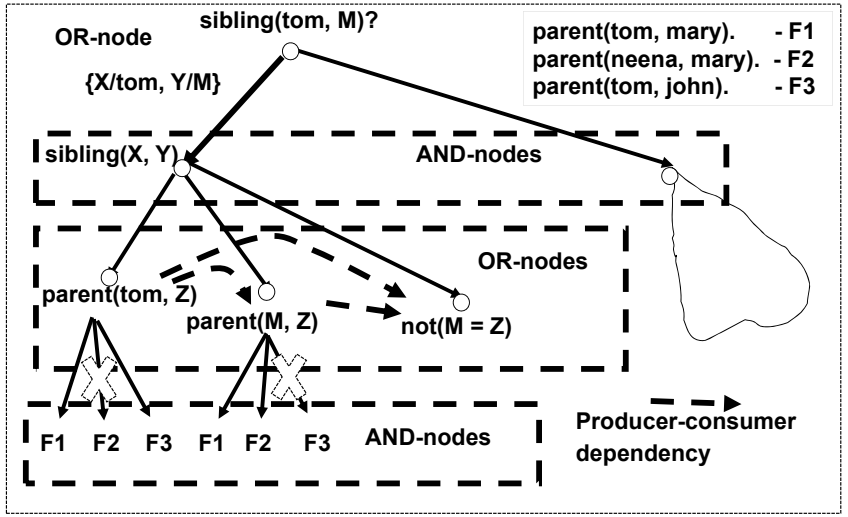
Query Reduction

- A query is an OR-node
- LHS of a rule is AND-node; and RHS subgoals are OR nodes
- LHS of a fact is AND-node; RHS is trivially true.
- Process
  - Rules with the same name and arity are identified
  - OR-node is unified with AND-node to generate a set of substitution
  - Substitution is applied on the RHS of the rule to generate a conjunction of subqueries. Each subquery is recursively solved for solution
  - If the AND-node is LHS of a fact then successful unification returns true
  - In case of facts substitution returns the value of unbound variables
  - If all subqueries return true then AND-node is true
  - If one of the rule is successful then OR-node is true
- **Example**

```
sibling(X, Y) :-  parent(X, Z), parent(Y, Z), not(X = Y) .
sibling(X, Y) :-  fraternity(X, Z),  fraternity(Y, Z), not(X = Y).
parent(tom, mary).   parent(neena, mary).        parent(tom, john).
```

Query: *sibling(tom, M)?*

AND-OR Tree Illustration



Implementing AND-OR Tree

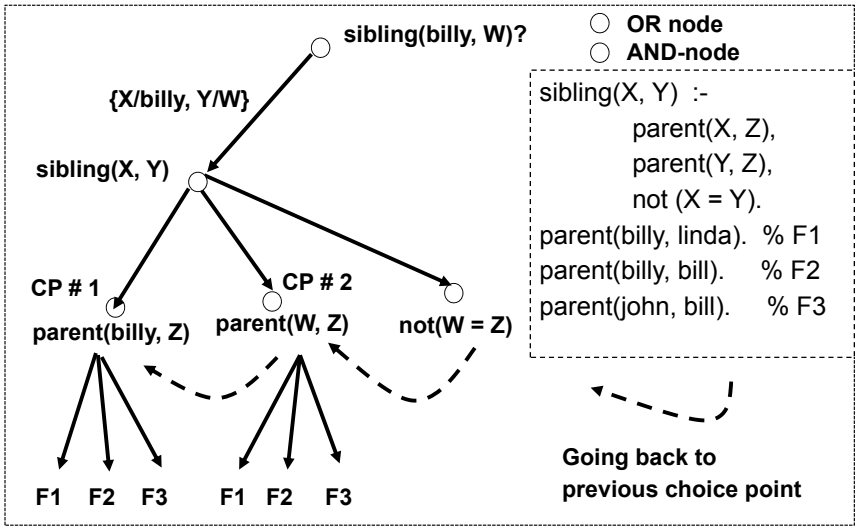
- Sequential implementations use Depth first search
- Concurrent implementations also use breadth-first search
- Depth first search implementation
  - AND-OR tree is expanded in a focused way
  - First clause is tried first followed by next clauses
  - Leftmost subgoal in a clause is tried first followed by the second subgoal.
- Searching the options in Depth-first search
  - OR-nodes are points of exploring different rules. If one rule does not yield solution, next rule is tried.
  - OR-nodes are called **choice-points**
  - Trail of choice points are stored in a special stack to explore next rule. The stack is called **trail stack**
  - **Trail stack** stores the choice points in the LIFO order

Backtracking

- Backtracking pops the last choice-point to explore alternate rules
  - Pop the last choice point from the trail-stack
  - Unbind all the variables that were bound between the last choice point and the failed subgoal
  - Pick up next unexplored clause (AND-node)
  - Unify the OR-node corresponding to the choice point and the next AND-node to pick up new set of bindings
  - Solve the subqueries (OR-nodes) in the subgoals of the new rule
- Termination condition for the search
  - All the choice-points have been consumed, and trail-stack is empty
- Example

```
sibling(X, Y)  :- parent(X, Z),  parent(Y, Z), not (X = Y).
parent(billy, linda).          /* fact F1 */
parent(billy, bill).           /* fact F2  */
parent(john, bill).            /* fact F3 */
```


Backtracking Illustration



Warren Abstract Machine

- WAM is a low level abstract machine for compiling logic programs
  - Uses a hash table to jump to specific procedure
  - Uses a variation of if-then-else and exit to try different rules
  - Uses registers to pass the arguments to the called procedure
  - Uses a special stack called trail-stack for backtracking
  - Local variables in control stack; and complex logical terms in heap
  - Nested structure represented as multicells connected through pointers
  - A structure *f(a, b(M, N), L)* maps to $register_1 \rightsquigarrow f(a, X1, L)$, $X1 \rightsquigarrow b(M, N)$.
- WAM Architecture: Registers, control stack, heap, trail-stack, code area
- WAM Instructions:
  - *hashing on procedure-name, arity and first argument*,
  - *set_structure, set_value,*
  - *get_structure, get_value, put_value,*
  - *conditional jump, arithmetic operations,*
  - *unify_variable, and unify_value,*
  - *try_me_else <Label> ; retry_me_else <Label>,*
  - *trust_me_else_fail*
  - *proceed*

Compiling to WAM Instructions

- Translation

  clausehead($Args_0$) :- $subgoal_1$($Args_1$), … $subgoal_N$($Args_N$)

  When translated in WAM will have following pattern:

  Allocate N % for at least N variables

  get_arguments of Clausehead in the registers

  put_arguments of $subgoal_1$;

  call $subgoal_1$;

  …

  put arguments of $subgoal_N$;

  call $subgoal_N$;

- Instructions after compiling clauses
  - *try_me_else* <*label*> stores the label <label> in the control stack, and executes the first clause. If the first clause fails, then it pops the label <label> from the stack and jumps to the next clause
  - *try_me_else_fail* is used for the last clause, and states that if the clause does not succeed then go back to the previous choice point

Program Analysis

- Program analysis is used to
  - Derive new properties of the program in some abstract domain
  - Abstract domain could be type domain, modes etc.
- Applications of Program Analysis
  - Derive types and modes of the arguments
  - Derive concurrency in the program
  - Derive producer-consumer relationship for automatic parallelization of the program
  - Used for compiler optimization such as tail recursive optimization

Programming in Prolog

- Prolog programs
  - Horn clause has one term on LHS and multiple subgoals on RHS
  - Prolog uses **negation as failure** that means failure of a subgoal is equivalent to negation.
  - Prolog rules are not mutually exclusive.
  - Prolog programs support nondeterministic programming due to unification allowing bidirectional information flow
  - Supports meta programming to reason about programs
  - Prolog can build predicates as data
- Prolog uses backtracking and depth first search
  - Backtracking can be enforced to generate multiple solutions to a query
  - Clauses are tried top to bottom and subgoals are tried left to right
- Archiving partial computations
  - Blackboards or assert/retract to substitute for global variables
  - A fact can be asserted (inserted in the database) or retracted (deleted) from the database providing capability of destructive update
  - Blackboard is a global data structure that is destructively updated

Prolog Programming II

- Blackboard based programming
  - Improves efficiency by storing partial computation
  - Can be used a global variable

```
factorial_bb(N, V) :- bb_get(fact:N, V), !.
factorial_bb(N, V) :- factorial(N, V),  bb_put(fact:N, V).

factorial(N, M) :-  N > 0, !, N1 is N – 1, factorial_bb(N1, M1), M is N * M1.
factorial(0, 1).
```

- Use of cuts
  - Cuts of the choice-points in that rule
  - Improves efficiency if no more solutions are needed from rule
  - Used to simulate control abstractions such as if-then-else and repeat-loop

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

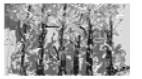
  - Unsafe use of cut may cause incorrectness by not deriving a solution

Prolog Programming III

- Programming with sets: the predicate setof/3
  - Arguments: 1) the variable to hold mutiple values; 2) predicate to be executed; and 3) a variable to hold the set of solutions

```
cs_majors(Students) :-
    setof(C, cs_majors(C), CS_Scientists),
    setof(B, biology_majors(B), Biologists),
    setof(B, math_majors(B), Mathematicians),
    union([Biologists, Mathematicians], Set1),
    subtract(CS_Scientists, Set1, Students).
compbio_majors(Students) :-
   setof(C, computer_science_majors(C), CS_Scientists),
   setof(B, biology_majors(B), Biologists),
   intersection(CS_Scientists, Biologists, Students).
cs_majors('Ahmad').        cs_majors('Kevin').        cs_majors('Shivani').
biology_majors('Ahmad').   biology_majors('Julie').
math_majors('Tom').        math_majors('Kevin').
```


Nondeterministic Programming



- Nondeterminism is supported due to
  - Logical OR between the rules since OR is commutative
  - Logical AND between the subgoals since AND is commutative
  - Bidirectional information flow during unification between a goal and the corresponding clausehead
- Example I

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

  - Member/2 can be executed in two modes: member(-, +) or member(+, +)
- Example II

```
append([ ], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).
```

  - append/3 can be executed in 2 modes: append(+, +, -); append(-, -, +);
- Advantages of unification
  - Multiple problems with alternative strategies can be programmed easily
  - Examples: missionary and cannibal problem; various games

Data Abstractions

- Prolog supports tuples, linked-lists, dynamic arrays, sets, facts
  - Tuples are used to simulate dynamic creation of structures
  - Linked-lists can be traditional or they can be extended indefinitely
  - Logic programs also support difference lists in the form [a|X] – X to describe an efficient extensible list.
  - Arrays are implemented as dynamically extensible trees
  - Trees are also used to handle association-lists and dictionaries for looking up a value given the key
  - Dynamic insertion of facts is used to build new information for reasoning
- Example of dynamic structure creation

```
student_new(Name, Age, Course, Id) :- student_template(Template),
Template = student(name(Name)-age(Age)-course(Course)-id(Id)),
assert(Template), !.
% mode student_info(+, +, ?).
student_info(age, Name, Age) :-   student(name(Name) -age( Age) - _ - _).
student_info(course, Name, Course) :- student(name(Name) - _ -
course(Course) - _).
student_info(id, Record, Id) :-  student(name(Name)- _ - _ - id(Id).
student_template(student(name( _ )-age( _ )-course( _ ) – id( _ )).
```


Abstractions II

- Logic programs supports
  - if-then-else, repeat-loop and recursive programming
  - Control abstractions can be simulated using cuts, blackboards, and tail-recursive programming
- Example of Dictionary and if-then-else in Sicstus Prolog

```
% mode lookup(+-?, ?).
lookup(Key-Value, Dictionary) :-
        var(Dictionary), format("Do you want to insert in dictionary?", [ ]),
        read(Answer),
        ( Answer == 'y' → Dictionary  = (Key-Value, _, _)
        ; otherwise → true
        ).
lookup(Key-Value, Dictionary) :-
        Dictionary = (Key1-Value1, _, _), Key == Key1, Value = Value1, !.
lookup(Key-Value, Dictionary) :- Dictionary = (Key1-_, Left, _) :-
                Key < Key1, !, lookup(Key-Value, Left).
lookup(Key-Value, Dictionary) :- Dictionary = (Key1-_, _,  Right) :-
                Key >= Key1,!, lookup(Key-Value, Right).
```

Simulating Control Abstractions

- Simulating if-then-else

```
If-then-else(Predicate,  ThenGoal, ElseGoal) :-
   call(Predicate), !, call(ThenGoal).
If-then-else(Predicate, _, ElseGoal) :- call(ElseGoal).
```

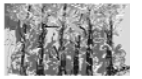
- Simulating negation of a goal

```
\+(Goal) :-  call(Goal), !, fail.
\+(Goal) :- !.
```

- Simulating iteration using tail-recursion

```
print_hello :-  read(N),  write_hello(N, 1).
write_hello(Max, Index) :-
   (        Index =< Max →   write("Hello"), nl,
            NewIndex is Index + 1,
            write_hello(Max, NewIndex)
   ;otherwise → true
   ).
```


Limitations of Prolog



- Unification does not evaluate an expression
  - Needs additional predicates to flatten expressions
  - Parameters can not evaluate complex expressions

Not Allowed in Standard Prolog	Corresponding Prolog Program
factorial(0, 1). factorial(N, M) :- factorial(N-1, M1), M is N*M1.	factorial(0, 1). factorial(N, M) :- N1 is N – 1, factorial(N1, M1), M is N*M1.

- Occurs check: X = f(X) will not terminate
- overhead of storing the choice points even for the deterministic programs with multiple clauses
  - Needs program analysis to differentiate deterministic and nondeterministic procedures to remove unnecessary choice-points

Temporal Logic Programming

- Concept of truthfulness of a predicate in a time interval
  - Predicate may be true in one time interval and false in another
  - Temporal logic programming languages: Templog and Tokio
- Operators in temporal logic programs
  - Predicate p is true at the next time instant.
  - Predicate p is always true
  - Predicate p is never true
  - Predicate p eventually becomes true;
  - Predicate p precedes predicate q
  - Predicate q is true until predicate p becomes true.
- Some application of temporal logic programming
  - Scheduling queues – to execute requests in a chronological order
  - Message processing – sending messages and acknowledging in chronological order

Constraint Logic Programming

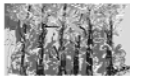
- Puts restrictions to prune the search space in deriving the solution
- Applications
  - Space optimization problems, time scheduling problem, planning, resource allocation, message routing to balance the traffic load, load balancing, profit maximization problems
- Format of clause in constraint logic programs
  - Clausehead(X, Y) :- constraint(X), predicate1(X, Z), predicate2(Z, Y).
- Implementation model
  - WAM augmented with a constraint store and constraint solver
  - Constraint is stored in constraint store and solved using constraint solver
  - Predicates go through standard unification process
- Constraint programming languages: CHIP and Ecl^{i}ps^e

```
% handling interval logic
:-  lib(ic). % use interval constraint library
:-  use_module(library(ic)).  % load interval constraints
year(M) :-  M ::  1..12.  % assign interval 1 .. 12 to the variable M
winter(4).                  fall(9).
summer_months(M) :- year(M), winter(W), fall(F), M #> W, M  #<  F.    %  5..8
```

Inductive Logic Programming

- Deductive logic is used reasoning from existing database
  - Not good for learning new rules from examples
- Inductive logic programming is about learning new rules
  - Uses positive and negative examples and background knowledge
  - Background knowledge includes background facts and rules
- Technique to form generalized rule
  - The process is of forming generalization followed by specialization
  - Find out the patterns relating the arguments in positive examples
  - Form a generalized rule using the pattern
  - Specialize using the negative examples
- Example
  - Facts: parent(tom, mary). parent(joe, mary).
    parent(cathy, john). parent(nina, john).
  - second arguments of fact3 and fact 4 are the same
  - Generalized rule: my_relationship(X, Y) :-
    parent(X, Z), parent(Y, Z), not (X == Y).

Higher Order Logic Programming



- Operator '=..' transforms
  - [predicate_name, Arg1, …, ArgN] to predicate_name(Arg1, …, ArgN)
  - The new predicate can be called dynamically
  - The operator can simulate apply_all, construction, and insertion functional forms.

Construction	apply_all
construct(PredList, Data, ResList) :- (PredList == [ ] → Result = [ ] **; otherwise** → PredList = [PredName | Fs], ResList = [Result | Rs], append(Data, [Result], Args), Goal **=..** [PredName | Args], **call**(Goal), construct(Fs, Data, Rs)).	apply_all(Pred, Data, ResList) :- (Data == [ ] → Result = [ ] **; otherwise** → Data = [Data | Ds], ResList = [Result | Rs], Goal **=..** [Pred, Data, Result], **call**(Goal), apply_all(Pred, Ds, Rs)).

Logic + Functional Programming

- The integration has been achieved by
  - Combining bidirectional informational flow present in unification with unidirectional information flow in expression reduction
  - Interfacing functional and logic programming languages
  - Modifying unification to handle λ-expressions
- Techniques for integration
  - **Narrowing** - minimal substitution between two terms using unification so that a term can be reduced using term-reduction techniques
    sum_list(nil) **<=** 0.
    sum_list(X**::**Xs) **<=** X + sum_list(Xs).
  - **Residuing** - delay the evaluation of expressions containing uninstantiated variables until the logical variables get instantiated. It is sufficient for functional programming style. However, fails for logic programming style due to the presence of don't care variables
  - **Semantic unification -** finds out the semantically equivalent expressions and performs minimum reduction until the two terms can be unified
    (X, X) = (5, 2 + 3) succeeds giving binding as {X/5}

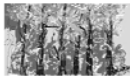
Integration with OOP

- An object in OOLP has method like Prolog procedures
  - OOLP methods are invoked like <object>**.**<method>
- Preprocessor translate OOLP to logic programs compiled to WAM
- Two approaches to incorporate OOLP
  - Preprocessor based: *Intermission, Vulcan, CPU, OOPP, and OLPSC.*
  - Library of Object-oriented programming: Sicstus Prolog
- Languages integrating Logic, Functional and OOP
  - *G, FLOOPS, UNIFORM, PARADISE* and *LIFE*
- Oz/Mozart is a multiparadigm language integrating
  - Logic programming, object-oriented programming, and concurrent programming
  - Supports both deterministic and nondeterministic programming

Concurrent Logic Programming

- Types of concurrency
  - OR-parallelism: clauses are spawned separately on different processors
  - AND-parallelism : subgoals are spawned concurrently on different processors
  - Stream parallelism : producer-consumer relationship between subgoals through shared variables having multiple solutions
- Popular concurrent programming languages
  - Concurrent Prolog and variations: exploiting stream and AND parallelism
  - Parlog and variations: exploiting OR parallelism
- Logic programs have been implemented on distributed computers and massive parallel computers
  - Unification level parallelism on SIMD computers
  - Multiple sequential execution of different clauses to exploit coarse-grain concurrency
  - Coordinator based model where a coordinator facilitates data transfer on multiple concurrently executing chunks of sequential programs

Summary I



- Logic programming uses first order predicate calculus
- First order predicate calculus integrates propositional calculus, universal quantification and existential quantification
- Logic program is a set of procedures each having multiple rules
- There are two types of reasoning systems: forward reasoning and backward reasoning
- Logic programming uses unification for equating two logical terms
  - Unification has two directional information flow, does not evaluate expressions, and performs matching position by position
- The implementation model of logic programs is AND-OR tree
  - OR nodes correspond to subgoals and AND-nodes to clauseheads
  - Low level instruction machine for compilation is Warren Abstract Machine
- **WAM** stores choice-points on trail-stack to support backtracking
- Prolog is implemented using depth-first search

Summary II

- Prolog uses blackboards and assert/retract for destructive update and global variables. Some implementations explicitly support global variables or destructive updates as in GNU Prolog.
- Logic programs support multiple data abstractions through the use of tuples, lists, sets, and facts.
- Control abstractions can be supported through the use of cuts, tail-recursive programming, and use of '=..' that allow predicates to be built as data
- Logic programming can be extended using the notion of temporal ness, constraints, induction, and integration with functions
  - Temporal logic programming allows time interval based truthfulness of axioms.
  - Constraint logic programming uses a combination of constraint store, constraint solver and Prolog engine to solve constraint logic programs.
  - Inductive logic programming uses background information generalization and specialization to learn new rules.
  - Integration with functions uses three techniques: narrowing, residuation and semantic unification to integrate unification with term reduction.