Logic Programming

- A radically different paradigm for programming: declarative programming
  - rather than writing control constructs (loops, selection statements, subroutines)
  - you specify knowledge and how that knowledge is to be applied through a series of rules
  - the programming language environment uses one or more built-in methods to reason over the knowledge and prove things (or answer questions)
    - in logic programming, the common approach is to apply the methods of resolution and unification
- While these languages have numerous flaws, they can build powerful problem solving systems with little programming expertise
  - they have been used extensively in AI research

Terminology

- Logic Programming is a specific type of a more general class: production systems (also called **rule-based systems**)
  - a production system is a collection of facts (knowledge), rules (which are another form of knowledge) and control strategies
  - we often refer to the collection of facts (what we know) as working memory
  - the rules are simple if-then statements where the condition tests values stored in working memory and the action (then clause) manipulates working memory (adds new facts, deletes old facts, modifies facts)
  - the control strategies help select among a set of rules that match – that is, if multiple rules have matching conditions, the control strategies can help decide which rule we select this time through
  - there are other control strategies as well – whether we work from conditions to conclusions or from conclusions to conditions (forward, backward chaining respectively)

Logic Programming

- Also known as declarative programming
- Mostly synonymous with the Prolog language because it is the only widely used language for logic programming
- A declarative program does not have code, instead it defines two pieces of knowledge
  - facts – statements that are true
  - rules – if-then statements that are truth preserving
- We use the program to prove if a statement is true and/or answer questions
- The reason this works is that Prolog has built-in problem solving processes called resolution and unification
  - in fact Prolog is not so much a programming language as it is a tool, but it does have *some* programming language features that make it mildly programmable

Background for Logic

- A proposition is a logical statement that is only made if it is true
  - Today is Tuesday
  - The Earth is round
- *Symbolic logic* uses propositions to express ideas, relationships between ideas and to generate new ideas based on the given propositions
- Two forms of propositions are
  - Atomic propositions
  - Compound terms (multiple propositions connected through the logical operators of and, or, not, and implies)
- Propositions will either be true (if stated) or something to prove or disprove (determine if it is true) – we do not include statements which are false
- For symbolic logic, we use 1st order predicate calculus
  - statements include predicates like round(x) where this is true if we can find an x that makes it true such as round(Earth) or round(x) when x = Earth
  - you might think of a predicate as a Boolean function except that rather than returning T or F, it finds an x that makes it true

Logic Operators

Name	Symbol	Example	Meaning
negation	¬	¬ a	not a
conjunction	∩	a ∩ b	a and b
disjunction	∪	a ∪ b	a or b
equivalence	≡	a ≡ b	a is equivalent to b
implication	⊃ ⊂	a ⊃ b a ⊂ b	a implies b b implies a
universal	∀ X.P		For all X, P is true
existential	∃X.P		There exists a value of X such that P is true

- Equivalence means that both expressions have identical truth tables
- Implication is like an if-then statement
  - if a is true then b is true
  - note that this does not necessarily mean that if a is false that b must also be false
- Universal quantifier says that this is true no matter what x is
- Existential quantifier says that there is an X that fulfills the statement

∀X.(woman(X) ⊃ human(X))
– if X is a woman, then X is a human

∃X.(mother(mary, X) ∩ male(X))
– Mary has a son (X)

Clausal Form

- To use resolution, all statements must be in clausal form
  - $B_1 \cup B_2 \cup \ldots \cup B_n \subset A_1 \cap A_2 \cap \ldots \cap A_m$
    - B_1 or B_2 or … or B_n is true if $A_1$and A_2 and … and A_m are all true
      - the left hand side is the *consequent* (what we are trying to prove) and the right hand side is the *antecedent* (what conditions must hold true)
      - notice we have turned the if-then statement around,
  - We must modify our knowledge so that:
    - existential quantifiers are not required (eliminate all of them)
    - universal quantifiers are implied (by replacing each with a specific variable)
    - no negations (all negations must be removed)
  - We break down our statements to have a single item on the left
    - the above statement is broken into multiple statements such as
      - $B_1 \subset A_1 \cap A_2 \cap \ldots \cap A_m$
      - $B_2 \subset A_1 \cap A_2 \cap \ldots \cap A_m \ldots$
      - $B_n \subset A_1 \cap A_2 \cap \ldots \cap A_m$
    - note that propositions and predicates by themselves are already in clausal form, such as round(Earth) and Sunny

Example Statements

Consider the following knowledge:

Bob is Fred's father → father(Bob, Fred)
Sue is Fred's mother → mother(Sue, Fred)
Barbara is Fred's sister → sister(Barbara, Fred)
Jerry is Bob's father → father(Jerry, Bob)

And the following rules:

A person's father's father is the person's grandfather
A person's father or mother is that person's parent
A person's sister or bother is that person's sibling
If a person has a parent and a sibling, then the sibling has the same parent

These might be captured in first-order predicate calculus as:

$\forall$x, y, z : if father(x, y) and father(y, z) then grandfather(x, z)
$\forall$x, y : if father(x, y) or mother(x, y) then parent(x, y)
$\forall$x, y : if sister(x, y) or brother(x, y) then sibling(x, y) and sibling(y, x)
$\forall$x, y, z : if parent(x, y) and sibling(y, z) then parent(x, z)

We would rewrite these as

grandfather(x, z) $\subset$ father(x, y) and father(y, z)
parent(x, y) $\subset$ father(x, y)
parent(x, y) $\subset$ mother(x, y) etc

Resolution and Unification

- Given a collection of knowledge
  - we will want to prove certain statements are true or answer questions
- For instance, from the previous example, we might ask
  - who is Bob’s grandfather?
  - is Sue Barbara’s parent?
- How can this be done? Through backward chaining through rules
- Here is how backward chaining works
  - we want to prove that A is true
  - find a rule with A on its LHS and whatever is on the rule’s RHS must now be proven to be true, so we add the items on the RHS to a list of things we are trying to prove
  - repeat until
    - we match something that we know is true to our list, then remove the item
    - we run out of items on our list, then we are done, we have proven A is true
- To complicate matters, predicates (e.g., round(X)) need to be *unified*, that is, to prove round(X) is true, we have to find some X where we know it is true, for instance, round(Earth)

Complete Logic Example

Assume that we know the following about pets:

poodle(COCOA)
setter(BIG)
terrier(SCOTTY)

dog(X) ⊂ poodle(X) (poodles are dogs)
dog(X) ⊂ setter(X) (setters are dogs)
dog(X) ⊂ terrier(X) (terriers are dogs)
small(X) ⊂ poodle(X) (poodles are small)
small(X) ⊂ terrier(X) (terriers are small)
big(X) ⊂ setter(X) (setters are big)
pet(X) ⊂ dog(X) (dogs are pets)
indoorpet(X) ⊂ pet(X) and small(X)
(small pets are indoor pets)
outdoorpet(X) ⊂ pet(X) and big(X)
(big pets are outdoor pets)

If we want to find out what would make a good indoor pet, we ask indoorpet(?)

This requires finding pet(X) and small(X)
(find an X to make both predicates true)
pet(X) is implied by dog(X),
dog(X) is implied by terrier(X),
SCOTTY is a terrier so SCOTTY is a dog so SCOTTY is a pet

Can we find if SCOTTY is small?
small(SCOTTY) is implied by terrier(SCOTTY) which we already know is true, therefore, since terrier(SCOTTY) is true, small(SCOTTY) and pet(SCOTTY) are true, so indoorpet(SCOTTY) is True

Continuing with this process will also prove that indoorpet(COCOA) is true.

PROLOG

- PROLOG is a programming language that allows the programmer to specify declarative statements only
  - declarative statements (things you are declaring) fall into 2 categories
    - predicates/propositions that are true
    - clauses (*truth preserving* rules in clausal form)
  - once specified, the programmer then introduces questions to be answered
    - PROLOG uses resolution (backward chaining) and unification to perform the problem solving automatically
- PROLOG was developed in France and England in the late 70s
  - the intent was to provide a language that could accommodate logic statements and has largely been used in AI but also to a lesser extent as a database language or to solve database related problems

Elements of Prolog

- Terms – constant, variable, structure
  - constants are atoms or integers (atoms are like those symbols found in Lisp)
  - variables are not bound to types, but are bound to values when instantiated (via unification)
  - an instantiation will last as long as it takes to complete a goal
    - proving something is true, or reaching a dead end with the current instantiation
  - structures are predicates and are represented as
    - functor(parameter list) where functor is the name of the predicate
- All statements in Prolog consist of clauses
  - headed clauses are rules
  - headless clauses are statements that are always true
    - in reality, a headless clause is a rule whose condition is always true
  - all clauses must end with a period

Rules

- All rules are stated in Horn clause form
  - the consequence comes first
    - the symbol :- is used to separate the consequence from the antecedent
  - And is denoted by , and Or is denoted by ; or separating the rule into two separately rules
  - variables in rules are indicated with upper-case letters
  - rules end with a .
  - examples:
    - parent(X, Y) :- mother(X, Y).
    - parent(X, Y) :- father(X, Y).
    - grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
    - sibling(X, Y) :- mother(M, X), mother(M, Y), father(F, X), father(F, Y).
  - we can use _ as a wildcard meaning this is true if we can find any clause that fits
    - father(X) :- father(X, _), male(X).
      - X is a father if X is male and is someone's father

Other Language Features

- Assignment statements are available using the *is* operator
  - A is B / 17 + C.
    - this works if B and C are instantiated and A is not
    - however, is does not work like a true assignment statement
      - you can not do x is x + y – this can never be true!
  - we might use the assignment operator in a rule such as
    - distance(X,Y) :- speed(X,Speed), time(X,Time), Y is Speed * Time
- List structures are also available using [ ] marks
  - as in new_list([apple, prune, grape, kumquat]).
  - this is not a binding of new_list to the values, but instead new_list is a predicate with a true instance of the predicate being the parameter [apple, prune, grape, kumquat]
    - lists can also be represented as a head and tail using | to separate the two parts similar to how Lisp uses CAR and CDR

More Prolog Examples

```
predecessor(X,Y) :- parent(X,Y); parent(X,Z), predecessor(Z,Y).
                    // X is a predecessor of Y if X is Y's parent or
                    // if X is parent of someone else who is a predecessor of Y
```

Using Not:

```
        dog(X) :- poodle(X).
        dog(X) :- terrier(X).
        likes(X,Y) :- dog(X), dog(Y), not (X=Y).
                    // can also be written as X =\= Y
```

Notice the use of "not" here – in Prolog, x != y is available but ~foo(x) is not

That is, we only declare statements that are true, we cannot declare the negation of statements that are false

Database example: imagine we have a collection of terms:

```
        record(name, yearborn, salary)
```

Successful person is someone who either makes > $50000 in salary or was born after 1980 and is making more than $40000.

```
        success(X) :- record(X, Y, Z), Z > 50000;
                    record(X, Y, Z), Y > 1980, Z > 40000.
```

Additional Prolog Examples

Defining Max:

```
max(X,Y,M) :- X > Y, M is X.
max(X,Y,M) :- Y >= X, M is Y.
```

Defining GCD:

```
gcd(X,Y,D) :- X=Y, D is X.
gcd(X,Y,D) :- X<Y, Y1 is Y - X, gcd(X, Y1, D).
gcd(X,Y,D) :- X>Y, gcd(Y, X, D).
```

Two List examples

Defining Length:

```
length([ ], 0).                    // empty list has a length of 0
length([ _ | Tail, N) :- length(Tail, N1), N is 1 + N1.  // a list that has an
// item _ and a Tail is length N if the length of Tail is N1 where N = 1 + N1
```

Sum of the items in a list:

```
sum([ ], 0).        // sum of an empty list is 0
sum([X | Tail], S) :- sum(Tail, S1), S is X + S1.
```

Advantages of Prolog

- There are several advantages to using Prolog
  - ability to create automated problem solvers merely by listing knowledge
  - a shortcut way to build and query a database
  - solving significantly difficult problems with minimal code:

Deriving the permutations of a list List:

```
perm(List,[H|Perm]):-delete(H,List,Rest),perm(Rest,Perm).
perm([ ],[ ]).
delete(X,[X|T],T).
delete(X,[H|T],[H|NT]):-delete(X,T,NT).
```

Sorting a list of values stored in List:

```
insert_sort(List,Sorted):-i_sort(List,[],Sorted).
i_sort([ ],Acc,Acc).
i_sort([H|T],Acc,Sorted):-insert(H,Acc,NAcc),i_sort(T,NAcc,Sorted).
insert(X,[Y|T],[Y|NT]):-X>Y,insert(X,T,NT).
insert(X,[Y|T],[X,Y|T]):-X=<Y.
insert(X,[],[X]).
```

A naïve sort (inefficient, but simple):

```
naive_sort(List,Sorted):-perm(List,Sorted),is_sorted(Sorted).
is_sorted([ ]).
is_sorted([ _ ]).
is_sorted([X,Y|T]):-X=<Y,is_sorted([Y|T]).
```

Deficiencies of Prolog

- Lack of control structures
  - Prolog offers built-in control of resolution and unification
    - you often have to force a problem into the resolution mold to solve it with Prolog (most problems cannot or should not be solved in this way)
- Inefficiencies of resolution
  - resolution, as a process, is intractable (O(2^n) for n clauses)
    - useful heuristics could be applied to reduce the complexity, but there is no way to apply heuristics in Prolog
      - they would just be additional rules that increases the size of n!
- Closed world assumption
  - in any form of logic reasoning, if something is not known, it is assumed to be false and *everything is* either true or false
- Negation is difficult to represent
  - since there is no NOT in Prolog, how do we represent NOT?
    - recall that anything explicitly stated must be true so we cannot specify NOT something as something would then be false

Rule-based Approaches

- Three of Prolog’s deficiencies can be eliminated (or lessened)
  - heuristics can be applied to improve efficiency
    - not necessarily reduce the complexity below $O(2^n)$ but improve it
  - uncertainty can be expressed by adding certainty factors or probabilities to data, rules and conclusions
  - use both forward and backward chaining
- Rule-based systems are less restrictive than the strictly logic-based approach in Prolog
  - by moving away from the formal logic approach however, doubts can arise from any results generated by such a system
    - that is, we can not be sure of the truth of something proven when the system contains non-truth-preserving rules and uncertain data
  - is it useful to move away from the strict logic-based approach given this uncertainty?
    - since nearly everything in the world has uncertainty, my answer is YES
- The rule-based approach is largely the same as in Prolog:
  - declare knowledge, provide rules, and ask questions to be answered, but most rule-based languages provide mechanisms for control strategies

Working Memory

- Rule-based systems divide memory into two sections
  - production memory – the collection of rules available
  - working memory –partial results and tentative conclusions
- The rule-based system works like this
  - compare the LHS conditions of every rule to working memory
  - select a rule whose left side matches (is found to be true or applicable)
    - this requires conflict resolution to pick a matching rule to select when multiple rules match
  - fire the rule (execute its RHS)
  - repeat until
    - a halt is performed (which means a conclusion has been reached)
    - the cycle count has been reached (max number of iterations), a breakpoint has been reached, or there are no matching rules (these are all failures)
- Rules are written in an if-then sort of format
  - if(square(E, 2) = whitepawn && square(E, 3) = empty && square(E, 4) = empty) → (square(E, 2) = empty && square(E, 4) = whitepawn)
  - if(context = "gram positive" && morphology = "coccus" && conformation = "clumps") → (assert identity = "staphylococcus" certainty = 0.7)

The Ops 5 Language

- Official Production System
  - other versions are Ops4 and Ops83
- Formally, a production system language
  - called rule-based language later
- Forward chaining system (or data driven)
  - starts with data, uses rules to infer conclusions (that is, maps LHS to RHS)
  - conclusions are often only partial or intermediate conclusions, other rules are applied to work towards final conclusions
    - this provides an ability to reason abstractly and then concretely
- Originally implemented in Lisp, later in BLISS
  - followed by a number of other production languages, notably SOAR which combined the rule-based approach with another method called "chunking"
  - used to implement numerous expert systems, most notably R1 (later called XCON) to design VAX computer configurations
- Data defined as tuples in a Lisp-like way:
  - (make student ^name Fox ^major CS ^gpa 1.931)
  - (make student ^name Zappa ^major MUS)
  - (make student ^name Bulger ^activity football)

Ops 5 LHS Conditions

- Conditions are specified by stating what is expected in working memory
  - does this item exist? (student ^major CS ^activity football)
    - is there a student whose major is CS and activity is football?
  - variables are available by enclosing the name of the variable in < > symbols
  - (student ^name <name> ^major CS ^gpa 4.0)
    - find student with major = CS, gpa = 4.0, store this student’s name in <name>
  - values can be tested against other values using <, >, =, <>, <= and >=, also arithmetic operations are available
  - compound conditions are available
    - where conjuctions are placed in { } as in (student ^gpa {>= 3.0 <= 4.0})
    - disjuctions are placed in << >> marks as in (student ^name <<anderson bruford wakeman howe>>)
  - testing to see if something does not exist: place – before the entry
    - - (student ^major MUS ^gpa 0.0)
      - are there no music majors with gpa of 0.0?

RHS Actions

- Actions are what will happen if a rule is fired (executed)
  - multiple actions can be listed, each in a separate list
  - actions usually revolve around manipulating working memory
    - add to working memory, for instance
      - (make student ^name <name> ^hours <oldhours> + 3)
    - remove from working memory
    - alter some value(s) of a piece of working memory
      - (modify 1 ^rank senior)
      - (modify 2 ^major) which removes the major value from that student’s entry
  - other actions include
    - compute – use values from the conditions and return a new value (do not update working memory)
    - I/O actions – open a file, close a file, input, output, append
    - input from keyboard
    - output to monitor
- A brief OPS5 example is shown in the notes section of this slide

Conflict Resolution

- In Prolog, if two or more rules have matching (true) clauses, they are tried one at a time until an answer is reached
  - but because of recursion and the depth-first approach taken, the second clause would only be tried after the first clause led to a failure
    - so if clauses 1 and 2 matched, we try clause 1, assume it leads to clauses 3, 4, 5, and 6 matching, each of which is tried and fails, we only try clause 2 after all that
- In Ops5, the decision of which rule to fire is based on a conflict resolution strategy, some options include:
  - refraction: a given rule cannot fire twice in a row
  - recency: select the rule that previously matched most recently
  - specificity: select the rule that has the most number of conditions on the LHS (this should be the most specific rule)

Handling Uncertainty

- There is no built-in way to handle uncertainty in Ops5
  - some possibilities used in AI:
    - Bayesian probabilities
    - fuzzy logic
    - certainty factors
  - how do you handle a conjunction or disjunction of items?
    - if a rules says (A B C → D) how do you compute the probability of A AND B AND C?
    - if a rule says (<<A B C>> → D) how do you compute the probability of A OR B OR C?
    - if the LHS matches, what is the probability of the RHS conclusion?
  - these questions have different answers depending on the form of uncertainty used
    - Ops5 does not directly support any of these
    - you can add probabilities/certainty factors to each piece of knowledge and add rules to handle the probabilities/certainty factors
- See the example code in the notes section of this slide

CLIPS

- There are a few problems with Ops5:
  - forward chaining is only appropriate in data-driven problems and yet people may not want to use Prolog for backward chaining problems
  - the conflict resolution strategies are built-in, programmers cannot implement their own
    - Ops5, like Prolog, is not so much a programming language as a tool, what about including other features?
- Clips – written in C (although it looks like lisp) offers solutions to these problems
  - Clips is a production system language but one that includes
    - class constructs including multiple inheritance, class daemons, polymorphism and dynamic binding
    - functions and function calls to provide a different form of control, including a progn statement, return, break
    - switch statement
    - forall construct used to test all possible combinations of a group of conditions
  - Clips can perform forward and/or backward chaining based on the way you specify your rules

CLIPS Rule Strategies

- In Ops5, all rules were considered during each iteration, and the only way to alter the search strategy (from forward chaining) was by using conflict resolution strategies
- In Clips, there are several added strategies:
  - focus: on a set of rules pertaining to a current context
  - agendas: rules whose conditions have matched but have not yet been evaluated can be stored in a list and consulted before searching through unmatched rules
  - deactivated rules: eliminate rules from consideration which are no longer relevant (deactivated rules may be later activated)
  - explicit conflict resolution strategies beyond those in Ops5
    - salience – each rule can be given a salience value, how important is it? select rules based on the highest salience
    - depth-first search or breadth-first search
    - simplicity – newly activated rules come before older activated rules
    - complexity – opposite of simplicity
    - random

Jess

- Java Expert System Shell
- Java-based implementation of Clips (to produce expert system applets)
  - Jess simplifies/restricts several elements of Clips
    - fewer resolution strategies, much of Clips OO facilities (these are replaced largely through Java Beans)
    - Jess is thought to be about 3 times slower than Clips
  - but Jess is built on top of Java so it contains all of Java’s GUI classes, is much more suitable for use over the Internet as a platform-independent tool and can use threads
- Basically the choice between the two comes down to
  - whether you want the full range of features in Clips or can live without some of it
  - want run-time efficiency
  - want platform-independence and GUI features