Logic Programming

And Prolog

5^{th}-Generation Languages

- Declarative (nonprocedural)
  - Functional Programming
  - Logic Programming
- Imperative
  - Object Oriented Programming

Nonprocedural Programming

Sorting procedurally:

1. Find the min in the remained numbers.
2. Swap it with the first number.
3. Repeat steps 1,2 until no number remains.

Sorting nonprocedurally:

1. B is a sorting of A $\leftrightarrow$ B is a permutation of A and B is ordered.
2. B is ordered $\leftrightarrow$ for each $i<j$: $B[i] \leq B[j]$

Which is *higher level*?

Automated Theorem Proving

- **A.T.P:** Developing programs that can construct formal proofs of propositions stated in a symbolic language.

- *Construct* the desired result to prove its existence (most A.T.P.'s).

- In *Logic Programming*, programs are expressed in the form of propositions and the theorem prover constructs the result(s).

- J. A. Robinson: A program is a theory (in some logic) and computation is deduction from the theory.

Programming In Logic (Prolog)

- Developed in *Groupe d'Intelligence Artificielle (GIA)* of the University of Marseilles (early 70s) to process a natural language (French).

- Interpreters: Algol-W (72), FORTRAN (73), Pascal (76), Implemented on many platforms (Now)

- Application in AI since mid-70s
- Successor to LISP for AI apps

- Not standardized (but has ISO standard now)

Structural Organization

13.2

```prolog
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Y) :- parent(X,Y), ancestor(Y,Z).
sibling(X,Y) :- mother(M,X), mother(M,Y),
                father(F,X), father(F,Y), X \= Y.
cousin(X,Y) :- parent(U,X), parent(V,Y), sibling(U,V).

father(albert, jeffrey).
mother(alice, jeffrey).
father(albert, george).
mother(alice, george).
father(john, mary).
mother(sue, mary).
father(george, cindy).
mother(mary, cindy).
father(george, victor).
mother(mary, victor).
```

SWI Prolog

```
?- [kinship].
% kinship compiled 0.00 sec, 3,016 bytes
Yes

?- ancestor(X, cindy), sibling(X, jeffrey).
X = george ↵
Yes

?- grandparent(albert, victor).
Yes

?- cousin(alice, john).
No

?- sibling(A,B).
A = jeffrey, B = george ; ↵
A = george, B = jeffrey ; ↵
A = cindy, B = victor ; ↵
A = victor, B = cindy ; ↵
No
```

Clauses

- Programs are constructed from A number of ***clauses***: <head> :- <body>
- Clauses have three forms:
  - *hypotheses* (facts)
  - *conditions* (rules)
  - *goals*

→ *assertions* (database)

→ *questions*

- Both <head> and <body> are composed of ***relationships*** (also called *predications* or *literals*)

Relationships

- Represent properties of and relations among the individuals
- A relationship is application of a *predicate* to one or more *terms*
- Terms:
  - ***atoms*** (or constants): `john, 25, …`
  - ***variables*** (begin with *uppercase letters*): `X, …`
  - ***compounds***
- *Horn clause form:* At most one relationship in <head>

Compound Terms

- It is *more* convenient to describe individuals without giving them names (*expressions* or *compounds* as terms).
- using *functors* (tags):
  d(X, **plus**(U,V), **plus**(DU,DV)) :- **d**(X,U,DU), **d**(X,V,DV).
- or using *infix functors*:
  d(X, U+V, DU+DV) :- **d**(X,U,DU), **d**(X,V,DV).
- instead of
  d(X,W,Z) :- **sum**(U,V,W), **d**(X,U,DU), **d**(X,V,DV), **sum**(DU,DV,Z).
- with less readability and some other things…

Data Structures

13.3

Primitives and Constructors

- *Few* primitives and *No* constructors.

- Data types and data structures are defined *implicitly* by their *properties*.

Example (datatype)

- Natural number arithmetic

```
sum(succ(X), Y, succ(Z)) :- sum(X,Y,Z).
sum(0,X,X).
dif(X,Y,Z) :- sum(Z,Y,X).

:-sum(succ(succ(0)),succ(succ(succ(0))),A).
A = succ(succ(succ(succ(succ(0)))))
```

- Very inefficient! (Why such a decision?)
- Use of **'is'** operator (unidirectional)

Principles

- Simplicity
  - Small number of built-in data types and operations

- Regularity
  - Uniform treatment of all data types as predicates and terms

Data Structures

- *Compound terms* can represent data structures

- Example: *Lists* in LISP

```
(car (cons X L)) = X
(cdr (cons X L)) = L
(cons (car L) (cdr L)) = L, for nonnull L
```

Lists in Prolog

- Using compound terms:

```
car( cons(X,L), X).
cdr( cons(X,L), L).
list(nil).
list(cons(X,L)) :- list(L).
null(nil).
```

- What about null(L)?

- How to accomplish (car (cons ‘(a b) ‘(c d)))?

Some Syntactic Sugar

- Using '.' infix functor (in some systems) instead of cons:
  - Clauses?

- Most Prolog systems allow the abbreviation:
  - $[X_1, X_2, \ldots, X_n] = X_1 . X_2 . \ldots . X_n.nil$
  - [ ] = nil
  - '.' is right associative!

Component Selection

- Implicitly done by **pattern matching** (*unification*).

```
append( [ ], L, L).
append( X.P, L, X.Q) :- append(P,L,Q).
```

- Compare with LISP append:

```
(defun append (M L)
   (if (null M)
     L
     (cons (car M) (append (cdr M) L)) ))
```

- *Taking apart* in terms of *putting together*!
  - **What X and P are cons'd to create M?**
  - **What number do I add to 3 to get 5 (instead of 5-3)**

- Efficient!?

Complex Structures

- A tree using lists (in LISP):
  - (times (plus x y) (plus y 1))

- Using compound terms directly (as records):
  - times(plus(x, y), plus(y, 1))

- Using predicates directly:
  - sum(x, y, t1).
  - sum(y, 1, t2).
  - prod(t1, t2, t3).

- Which is *better*?

Why Not Predicates?

Symbolic differentiation using predicate structured expressions:

d(X,W,Z) :- **sum**(U,V,W), **d**(X,Y,DU), **d**(X,V,DV), **sum**(DU,DV,Z).
d(X,W,Z) :- **prod**(U,V,W), **d**(X,U,DU), **d**(X,V,DV), **prod**(DU,V,A), **prod**(U,DV,B), **sum**(A,B,Z).
d(X,X,1).
d(X,C,0) :- **atomic**(C), C \= X.

Why Not Predicates? (cont.)

- Waste use of intermediate (temporary) variables
- Less readability
- Unexpected answers!

```
sum(x,1,z).
:- d(x,z,D).
```

No

- Why? What did *you* expect?
- How to correct it?

Closed World Model

- *All* that is true is what can be *proved* on the basis of the facts and rules in the database.
- Very reasonable in *object-oriented* apps (modeling a real or imagined world)
  - All existing objects are defined.
  - No object have a given property which cannot be found in db.
- Not suitable for *mathematical problems* (Why?)
  - An object is generally take to exist if its existance doesn't contradict the axioms.
- Predicates are better for OO-relationships, Compounds for mathematical ones (Why?)
  - We cannot assume existance of 1+0 whenever needed.

An Argument!

- What's the answer?

```
equal(X,X).
:- equal(f(Y),Y).
?
```

- What's the *logical* meaning? (*occurs check*)
- Any *other* meaning?
- Can it be represented in a *finite amount* of memory?
- Should we *detect* it?

Control Structures

13.4

Algorithm = Logic + Control

- N. Wirth: Program = data structure + algorithm
- R. Kowalski: Algorithm = logic + control

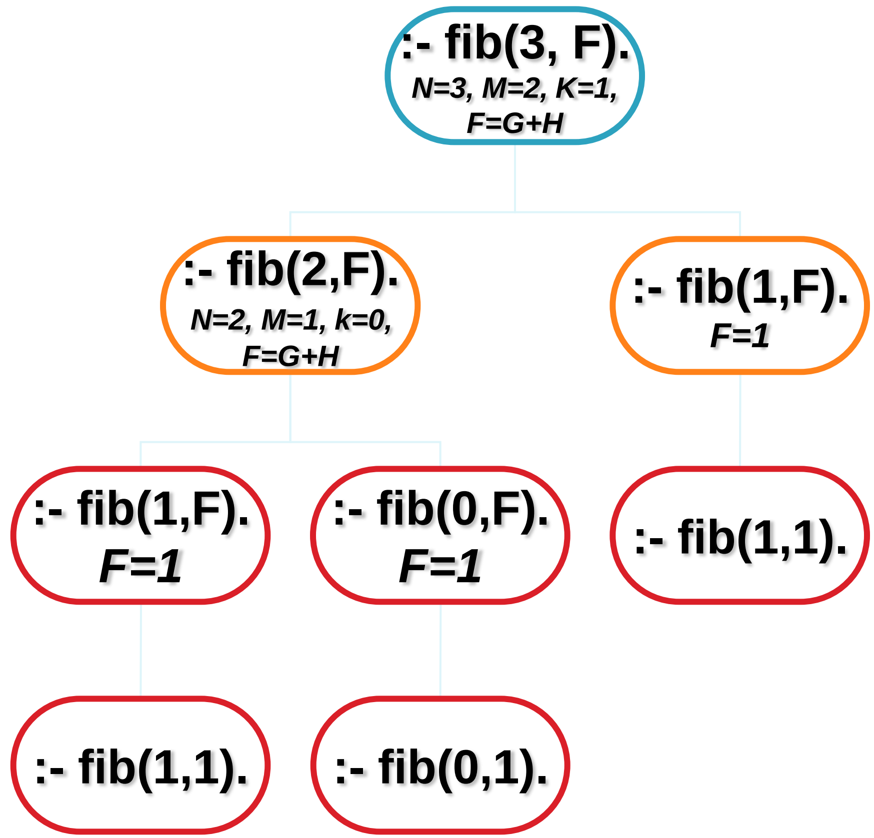
- In conventional programming:
  - *Logic* of a program is closely related to its *control*
  - A change in order of statements alters the meaning of program

- In (pure) logic programming:
  - *Logic* (logic phase) is determined by logical *interrelationships* of the clauses not their *order*.
  - *Control* (control phase) affects the *order* in which actions occur in time and only affects the *efficiency* of programs.

- Orthogonality Principle

Top-Down vs. Bottom-Up Control

- Top-down ≈ **Recursion**:
  - Try to reach the hypotheses from the goal.
- Bottom-up ≈ **Iteration**:
  - Try to reach the goal from the hypotheses.
- Hybrid:
  - Work from both the goals and the hypotheses and try to meet in the middle.
- Which one is better?




```
fib(0,1). fib(1,1).
fib(N,F) :- N=M+1, M=K+1, fib(M,G),
            fib(K,H), F=G+H, N>1.
```

Procedural Interpretation

- We have seen *logical* and *record* (data structure) interpretations.
- Clauses can also be viewed as ***procedure invocations***:
  - <head>: proc. definition
  - <body>: proc. body (a series of proc. calls)
  - Multiple definitions: branches of a conditional (case)
  - fib() example…
- Procedure calls can be executed in any order or even concurrently! (pure logic)
- Input/Output params are not distinguished!
  - fib(3,3) ↔ true. fib(3,F) ↔ F=3. fib(N,3) ↔ N=3. fib(N,F) ↔ ?

Unify, Fail, Redo…

- Heavy use of *unification*, *backtracking* and *recursion*.
- Unification (Prolog pattern matching – from *Wikipedia*):
  - One-time assignment (binding)
  - uninst. var with atom/term/another uninst. var (aliasing) (occurs check)
  - atom with the same atom
  - compound with compound if top predicates and arities of the terms are identical and if the parameters can be unified ***simultaneously***
  - We can use '=' operator to explicitly unify two terms
- Backtracking:
  - Make another choice if a choice (unif./match) ***failes*** or want to find ***other*** answers.
  - In logic prog. It is the ***rule*** rather than the ***exception***.
  - Very expensive!
- Example: **len**([ ], 0). **len**(X.T, L+1) :- **len**(T,L).

Prolog's Control Regime

- Prolog lang. is *defined* to use ***depth-first*** search:
  - Top to bottom (try the clauses in order of entrance)
  - Left to right
  - In pure logic prog., some complete deductive algorithm such as Robinson's ***resolution algorithm*** must be implemented.
- DFS other than BFS
  - Needs much fewer memory
  - Doesn't work for an infinitely deep tree (responsibility of programmer)
- Some programs may fail if clauses and subgoals are not ordered correctly (pp.471-474)
- Predictable execution of ***impure*** predicates (write, nl, read, retract, asserta, assertz, …)

SWI Prolog

```
[trace] ?- ancestor(X, cindy), sibling(X,jeffrey).
Event  Depth  Subgoal
==================================
Call:  (1)     ancestor(X, cindy)
Call:  (2)     parent(X, cindy)
Call:  (3)     father(X, cindy)
Exit:  (3)     father(george, cindy)
Exit:  (2)     parent(george, cindy)
Exit:  (1)     ancestor(george, cindy)
Call:  (1)     sibling(george, jeffrey)
Call:  (2)     mother(M, george)
Exit:  (2)     mother(alice, george)
Call:  (2)     mother(alice, jeffrey)
Exit:  (2)     mother(alice, jeffrey)
Call:  (2)     father(F, george)
Exit:  (2)     father(albert, george)
Call:  (2)     father(albert, jeffrey)
Exit:  (2)     father(albert, jeffrey)
Call:  (2)     george\=jeffrey
Exit:  (2)     george\=jeffrey
Exit:  (1)     sibling(george, jeffrey)

X = george
Yes
```

If we move *parent(X,Y) :- father(X,Y)* before *parent(X,Y) :- mother(X,Y)*, we have:

SWI Prolog

```
Event  Depth  Subgoal
==================================
Call:  (1)    ancestor(X, cindy)
Call:  (2)    parent(X, cindy)
Call:  (3)    mother(X, cindy)
Exit:  (3)    mother(mary, cindy)
Exit:  (2)    parent(mary, cindy)
Exit:  (1)    ancestor(mary, cindy)
Call:  (1)    sibling(mary, jeffrey)
Call:  (2)    mother(M, mary)
Exit:  (2)    mother(sue, mary)
Call:  (2)    mother(sue, jeffrey)
Fail:  (2)    mother(sue, jeffrey)
Redo:  (2)    mother(M, mary)
Fail:  (2)    mother(M, mary)
Fail:  (1)    sibling(mary, jeffrey)
Redo:  (3)    mother(X, cindy)
Fail:  (3)    mother(X, cindy)
Redo:  (2)    parent(X, cindy)
```

…

Cut!

- '!': Discard choice points of parent frame and frames created after the parent frame.

- Always is satisfied.
- Used to guarantee termination or control execution order.

- i.e. in the goal `:- p(X,a), !`
  - Only produce the 1st answer to X
  - Probably only one X satisfies *p* and trying to find another one leads to an infinite search!

- i.e. in the rule `color(X,red) :- red(X), !.`
  - Don't try other choices of red (mentioned above) and color if X satisfies red
  - Similar to ***then*** part of a if-then-elseif

Fisher, J.R., Prolog Tutorial, *http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html*

Red–Green Cuts (!)

- A ‘green’ cut
  - Only improves efficiency
  - e.g. to avoid additional unnecessary computation
- A ‘red’ cut
  - e.g. block what would be other consequences of the program
  - e.g. control execution order (procedural prog.)

Fisher, J.R., Prolog Tutorial,
http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html

Three Examples

See also MacLennan's example p.476

```
p(a).
p(X) :- s(X), r(X).
p(X) :- u(X).

r(a). r(b).

s(a). s(b). s(c).

u(d).

:- p(X), !
:- r(X), !, s(Y).
:- r(X), s(Y), !
:- r(X), !, s(X).
```

```
part(a). part(b). part(c).
red(a). black(b).

color(P,red) :- red(P),!.
color(P,black) :- black(P),!.
color(P,unknown).

:- color(a, C).
:- color(c, C).
:- color(a, unknown).
```

```
max(X,Y,Y) :- Y>X, !.
max(X,Y,X).
:- max(1,2,D).
:- max(1,2,1).
```

Fisher, J.R., Prolog Tutorial,
http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html

Higher-Order Rules

- Logic programming is limited to first-order logic: can't bind variables to predicates themselves.

- e.g. **red** (f-reduction) is illegal: (p(x,y,z) ↔ z=f(x,y))

```
red(P,I,[ ],I).
red(P,I,X.L,S) :- red(P,I,L,T), P(X,T,S).
```

- But is legal if the latter be defined as:

```
red(P,I,X.L,S):- red(P,I,L,T), Q=..[P,X,T,S],
  call(Q).
```

  - What's the difference?

Higher-Order Rules (cont.)

- In LISP, both code and data are *first-order* objects, but in Prolog aren't.
- Robinson *resolution algorithm* is refutation complete for *first-order* predicate logic.
- Gödel's *incompleteness theorem*: No algorithm is refutation complete for *higher-order* predicate logic.
- So, Prolog *indirectly* supports higher-order rules.

Negative Facts

- How to define **`nonsibling`**? Logically…

```
nonsibling(X,Y) :- X = Y.
nonsibling(X,Y) :- mother(M1,X), mother(M2,Y), M1 \= M2.
nonsibling(X,Y) :- father(F1,X), father(F2,Y), F1 \= F2.
```

- But if parents of X or Y are not in database?
  - What is the answer of nonsibling? Can be solved by…

```
nonsibling(X,Y) :- no_parent(X).
nonsibling(X,Y) :- no_parent(Y).
```

  - How to define **`no_parent`**?

Negative Facts (cont.)

- **Problem:** There is no *positive* fact expressing the *absence* of parent.

- Cause:
  - Horn clauses are limited to
  - C :- P1,P2,…,Pn ≡ C holds if P1^P2^…^Pn hold.
  - No conclusion if P1^P2^…^Pn don't hold!
  - If, *not* iff

Cut-fail

Solutions:

- Stating *all* negative facts such as no_parent
  - Tedious
  - Error-prone
  - Negative facts about sth are usually much more than positive facts about it

- *“Cut-fail”* combination
  - nonsibling(X,Y) is satisfiable if sibling(X,Y) is not (i.e. sibling(X,Y) is **unsatisfiable**)
  - nonsibling(X,Y) :- sibling(X,Y), !, fail.
  - nonsibling(X,Y).
  - how to define ‘fail’ ?!

negation :- unsatisfiablility

- **‘not’** predicate
  - not(P) is satisfiable if P is not (i.e. is **unsatisfiable**).
  - not(P) :- call(P), !, fail.
  - not(P).
  - nonsibling(X,Y) :- not(sibling(X,Y)).

- Is ‘not’ predicate the same as ‘logical negation’? (see p.484)

Evaluation and Epilog

13.5

Topics

- Logic programs are self-documenting
- Pure logic programs separate logic and control
- Prolog falls short of logic programming
- Implementation techniques are improving
- Prolog is *a step toward* nonprocedural programming

Self-documentation

- Programming in a higher-level, …
- Application orientation and…
- Transparency
  - programs are described in terms of predicates and individuals of the problem domain.
- Promotes clear, rapid, accurate programming

Separation of Logic and Control

- Simplifies programming
- Correctness only deals with logic
- Optimization in control cannot affect correctness
- Obeys **Orthogonality Principle**

Prolog vs. Logic Programming

- Definite control strategy
  - Programmers make explicit use of it and the result have little to do with logic
  - Reasoning about the order of events in Prolog is comparable in difficaulty with most imperative of conventional programming languages
- Cut doesn’t make any sense in logic!
- not doesn’t correspond to logical negation

Improving Efficiency

- Prolog is far from an efficient language.
- So, it's applications are limited to apps in which:
  - Performance is not important
  - Difficult to implement in a conventional lang.
- New methods are invented
- Some compilers produce code comparable to LISP

Toward Nonprocedural Programming

- *Pure* logic programs prove the possibility of nonprocedural programming.
- **In** *Prolog*, DFS requires programmers to think in terms of *operations* and their proper *ordering* in time (procedurally).
- **And** Prolog's control regime is more *unnatural* than conventional languages.
- **So,** there is still much more important work to be done before nonprocedural programming becomes *practical*.

Covered Sections of MacLennan

- 13.1
- 13.2
- 13.3
- 13.4
  - except topics starting on pp. 471, 475, 477, 484, 485, 486, 488
- 13.5

Presentation References

- Colmerauer, Alain, Philippe Roussel, The Birth of Prolog, *Nov. 1992, URL: http://www.lim.univ-mrs.fr/~colmer/ArchivesPublications/HistoireProlog/19novemb er92.pdf*

- Fisher, J.R., Prolog Tutorial*, 2004, URL: http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html*

- MacLennan, Bruce J., Principles of Programming Languages: Design, Evaluation and Implementation, *3rd ed, Oxford University Press, 1999*

- Merritt, Dennis, "Prolog Under the Hood: An Honest Look"*, PC AI magazine, Sep/Oct 1992*

- "Unification"*, Wikipedia, the free encyclopedia, 25 Sep. 2005, URL: http://en.wikipedia.org/wiki/Unification*

Thank You!