Prolog
Programming in Logic

Paper 7 Computer Science

Part 1B and Part II 50%

Ian Lewis, Andrew Rice

Agenda for this lecture

1) Aims & Objectives for the course
2) What's the point?
3) View Video #1 - "Prolog Basics"
4) Recap: Programming style, program structure, terms, unification
5) Course outline
6) Success vs. Failure in Prolog - life lessons

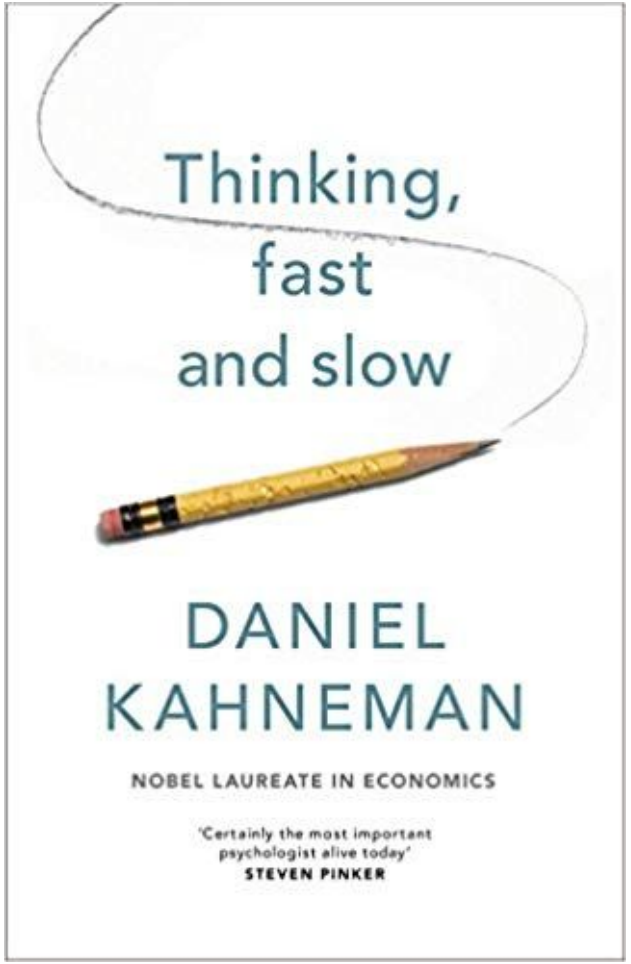
Aims

1. Introduce programming in the Prolog Language
2. A different programming style
3. Solve ‘real’ problems
4. Practical experimentation encouraged

Objectives

1. Understand the powerful capabilities of ‘pure’ Prolog: term structure, facts, rules and queries, unification.
2. Know how to model the backtracking behaviour of Prolog program execution, and recognize it as depth first, left-to-right search.
3. Appreciate the unique perspective Prolog gives to problem solving and algorithm design.
4. Understand how larger programs can be created using the basic programming techniques used in this course.

Why study Prolog?



Why study Prolog?

- In an imperative science, know and cherish the *declarative approach*

If you have a fact: taller(andy, ian). you are DECLARING, or ASSERTING, a relationship “taller” to hold between atoms “andy” and “ian”.

You can declare taller as an infix operator: op(500, xfx, taller).

Hence: andy taller ian.

?- andy taller X.

X = ian

Why study Prolog?

1200	xfx	-->, :-
1200	fx	:-, ?-
1150	fx	dynamic, discontiguous, initialization, meta_predicate, module_transparent, multifile, public, thread_local, thread_initialization, volatile
1100	xfy	; |
1050	xfy	->, *->
1000	xfy	,
990	xfx	:=
900	fy	\+
700	xfx	<, =, =.., =@=, \=@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, as, is, >:<, :<
600	xfy	:
500	yfx	+, -, /\, \/, xor
500	fx	?
400	yfx	*, /, //, div, rdiv, <<, >>, mod, rem
200	xfx	**
200	xfy	^
200	fy	+, -, \
100	yfx	.
1	fx	$

Table 5 : System operators

```
len([],0).
len([X|T],N) :- len(T,M),
                N is N + 1.
```

Why study Prolog?

```
fun fact(1) = 1;
    fact(N) = N * fact(N-1).

fun fact(N) = if (N = 1)
              then 1
              else N * fact(N-1).

fun append([ ],Y) = Y;
    append([X|Xs],Y) = [X|append(Xs,Y)].
```

These are all valid Prolog terms.

“Everything is a relation” (mostly, with a few hairy edges, like arithmetic)

You can write programs about programs.

Why study Prolog?

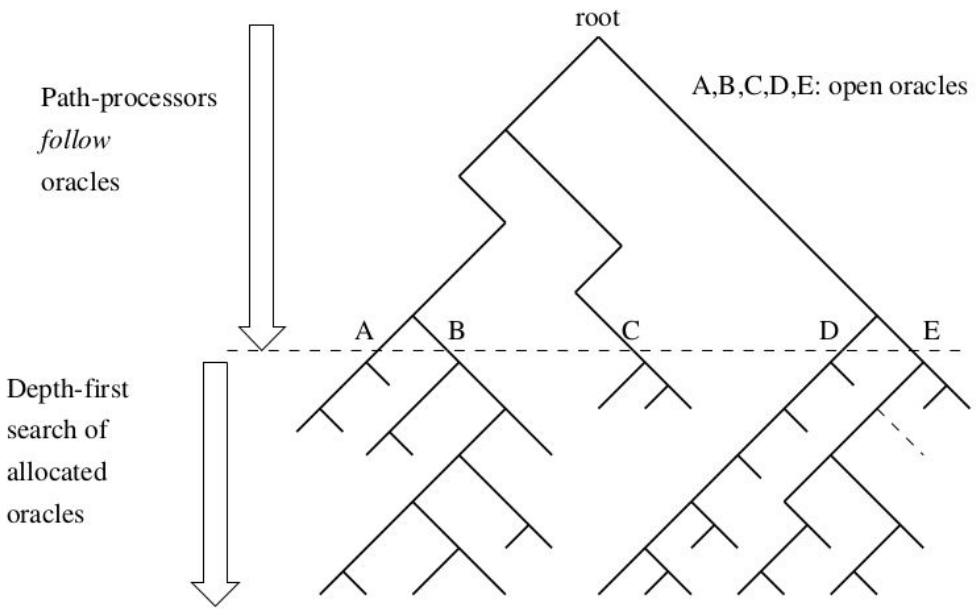


Figure 2.11: Second phase of *breadth-first partitioning*.

You will learn Prolog backtracking can be interpreted as a “search tree”.

Actually, given that a Prolog program is itself a valid Prolog term, you can apply *simple* transformations to that program to manipulate the tree. E.g.

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

Goes to:

```
last([X], X, [1]).
last([_|T],X,[2|P]) :- last(T,X,P).

?- last([a,b,c,d],X,P).
X= d
P = [2,2,2,1]
```

Why study Prolog?

Don’t worry about ANY of that.

Just recognize Prolog is all about DECLARING / ASSERTING RELATIONS.

“Everything” in Prolog is a ‘meaningless’ relation (with a few practical exceptions which are certain to torture you at some point).

Prolog programs are *facts* and *rules*, with *backtracking* providing a powerful search facility.

Unification on its own is an immensely powerful paradigm.

The combination of these ‘simple’ things can produce very complex behaviour.

Clauses + Unification + Backtracking = Programs.

Video #1: Prolog Basics

Programming Style

IMPERATIVE

```
l = [ 1,2,3,4,5 ];

sum = 0;

for (i=0; i<length(L); i++) {

    sum += 1;

}

return sum;
```

DECLARATIVE

```
fun sum([]) = 0

 |  sum(x::xs) = x + sum(xs);
```

```
sum([],0).

sum([X|L],S) :- sum(L,N), S is N+X.
```

Program Structure

Terms = atoms, variables, compound terms (can be infix)

Clauses = Facts + Rules.

Rules = Head :- Body.

Comments = % <anything>

?- = query prompt (often with side effects).

?- [<filename omitting .pl>]. = “consult” a file.

?- [user]. = “consult” user input (uses Prolog “assert”)

Terms

?- X = foo.

X = foo.

?- X = 1.

?- X = a.

?- X = 1.2.

?- X = a(1,a,Y,2).

?- X is 1+2

X = 3. (actually “?- is(X,+(1,2))”)

Compound term:

functor/arity

E.g.

foo(a,b(1),c) -> foo/3

Unification

Unification does not have a “direction”...

Atoms <-> Atoms (and constants)

Variable <-> Anything

Compound Term <-> (same functor/arity) & (arguments unify)

Occurs check e.g. X = a(X).

Unification

Term 1	Term 2	Result after unification
a	a	yes
1.2345	1.2345	yes
foo	X	X=foo
a(b,C)	a(X,p(q))	X=b,C=p(q)
a(b,c)	X(b,c)	

```
:- X = a(Y), Y = 7.
X = a(7),
Y = 7.
```

Backtracking

```
:- [user].
a(1).
a(3).
a(7).
a(9).
^D
:- X = a(Y), Y = 7.
X = a(7),
Y = 7
```

Prolog backtracking is depth-first, left-to-right

Life Lessons #1

Think DECLARATIVE.

`len([],0).` is asserting that “[]” and “0” are associated via the “len” relation.

Queries

```
:- len([],X).
```

```
:- len(X,0).
```

are equally reasonable.

Life Lessons #2

Think DEPTH-FIRST LEFT-TO-RIGHT

```
:- [user].
a(1).
a(3).
a(7).
a(9).
^D
:- X = a(Y), Y = 7.
X = a(7),
Y = 7
```

Your program might never end...

Life Lessons #2

Think DEPTH-FIRST LEFT-TO-RIGHT

```
:- [user].
len([],0).
len([_|T],N) :- len(T,M), N is M+1.
^D
:- len([a,b,c,d],N).
N = 4.
:- len(L,0).
```

Your program might never end...

Life Lessons #3

Don’t inject FUNCTIONAL support that doesn’t exist in Prolog

```
foo(L) :- ... X = max(L) ...
```

Life Lessons #4

Comment each relation:

```
% len(L,N) succeeds if L is a list and N is the length of that list.
len([],0)
. . .
```

Adhere to variable naming and ordering conventions:

If your relation has ‘input’ and ‘output’ arguments, say so in your comment AND put the input variables to the left of the output variables in the head of the clause.

Use variable names H and T (or L) for head and tail of a list (or H1, T1). Do not assume all variables have to be a single letter...

Summary:

Think DECLARATIVE.

Think DEPTH-FIRST LEFT-TO-RIGHT.

Comment each relation.

Adhere to variable naming and ordering conventions.

GOOD LUCK