4 Arithmetic and Nondeterminism

Prolog arithmetic can be a little surprising; you need to be aware of modes. Sometimes nondeterminism needs to be managed.

1. Arithmetic
2. Operators
3. Modes and Arithmetic
4. Expressions
5. Managing Nondeterminism
6. Arithmetic and Lists

4.1 Arithmetic

In most languages `3 + 4` is an *expression* that has a *value*

In Prolog, `3 + 4` is just a *term* whose functor is + and arguments are 3 and 4

It's not *code*, it's *data*

The Prolog builtin = just unifies (matches) two terms, it does not evaluate expressions

```
?- X = 3 + 4.

X = 3+4 ;

No
```

Arithmetic (2)

Use the built in `is/2` predicate to compute the value of an arithmetic expression:

```
?- is(X, 3+4).

X = 7 ;

No
```

`is(X, Y)` holds when `X` is a number and `Y` is an arithmetic expression and `X` is the value of `Y`

4.2 Operators

`is` is actually an infix **operator** in Prolog

This means that instead of writing `is(X,3+4)` we could also have written `X is 3+4`

This is easier to read, so it preferred, although both work

Similarly, `+` is an operator; `3+4` is the same as `+(3,4)`

Operators in Prolog are not exclusively for arithmetic, they are part of Prolog's syntax

Prolog has many other standard operators; some we have already seen, including `:- , ; mod`

Operators only affect *syntax*, not *meaning*

You can define your own operators

4.2.1 Precedence and Associativity

Operators have **precedence** and **associativity**

Use parentheses for grouping

E.g. * and / have higher precedence than + and -; they all associate to the left

```
?- X is 3 + 4 * 5, Y is (3 + 4) * 5.
X = 23
Y = 35
Yes

?- X is 5 - 3 - 1, Y is 5 - (3 - 1).
X = 1
Y = 3
Yes
```

4.2.2 Display

The built-in predicate `display/1` is useful for understanding how your input will be parsed when it includes operators

`display/1` prints a term as Prolog understands it, using only the standard form with the functor first, and arguments between parentheses

```
?- display(3 + 4 * 5), nl, display((3 + 4) * 5).
+(3, *(4, 5))
*(+(3, 4), 5)
Yes

?- display(5 - 3 - 1), nl, display(5 - (3 - 1)).
-(-(5, 3), 1)
-(5, -(3, 1))
Yes
```

4.3 Modes and Arithmetic

We could code a predicate to compute a square like this:

```
square(N, N2) :- N2 is N * N.
```

```
?- square(5, X).

X = 25 ;

No
?- square(X, 25).
ERROR: Arguments are not sufficiently instantiated
```

We can't use this definition to compute a square root!

Modes and Arithmetic (2)

Unfortunately, `is/2` only works when the second argument is ground

The first argument can be unbound or bound to a number

```
?- X is 3 + Y.
ERROR: Arguments are not sufficiently instantiated

?- 7 is 3 + 4.
Yes

?- 9 is 3 + 4.
No

?- 2 + 5 is 3 + 4.
No
```

4.4 Expressions

Some arithmetic expressions understood by `is/2`:

`E1 + E2`	addition	`float(E1)`	float equivalent of E1
`E1 - E2`	subtraction	`E1 << E2`	left shift
`E1 * E2`	multiplication	`E1 >> E2`	right shift
`E1 / E2`	division	`E1 /\ E2`	bitwise and
`E1 // E2`	integer division	`E1 \/ E2`	bitwise or
`E1 mod E2`	modulo (sign of E1)	`\ E1`	bitwise complement
`-E1`	unary minus	`min(E1, E2)`	minimum
`abs(E1)`	absolute value	`max(E1, E2)`	maximum
`integer(E1)`	truncate toward 0		

Expressions (2)

Some useful arithmetic predicates:

`E1 < E2`	less than	
`E1 =< E2`	equal or less	*Danger*: not <= !!!
`E1 >= E2`	greater or equal	
`E1 > E2`	greater than	
`E1 =:= E2`	equal (only numbers)	
`E1 =\= E2`	not equal (only numbers)	

All of these take ground arithmetic expressions as both arguments

4.5 Managing Nondeterminism

A common Prolog programming error is treating multiple clauses as if they formed an if-then-else construct

This is an erroneous definition of `absolute_value`:

```
absolute_value(X, X) :- X >= 0.
absolute_value(X, Y) :- Y is -X.
```

```
?- absolute_value(-3, V).
V = 3 ;
No
?- absolute_value(42, V).
V = 42 ;
V = -42 ;
No
```

Why do +ve numbers have two absolute values, one of them -ve?

Managing Nondeterminism (2)

The problem is that the second clause promises that

$$|n| = -n$$

for *any* n, regardless of its sign

The correct definition:

```
absolute_value(X, X) :- X >= 0.
absolute_value(X, Y) :- X < 0, Y is -X.
```

```
?- absolute_value(-3, V).
V = 3 ;
No
?- absolute_value(42, V).
V = 42 ;
No
```

Exercise

Define a Prolog predicate `maximum(X,Y,Z)` such that `Z` is the larger of `X` and `Y`

4.5.1 Fibonacci numbers

Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, . . .

```
fib(0, 1).
fib(1, 1).
fib(N, F) :-
        N > 1,
        N1 is N - 1,
        N2 is N - 2,
        fib(N1, F1),
        fib(N2, F2),
        F is F1 + F2.
```

What if we change the order of goals? What if we do `F is F1 + F2` first?

Note: each call makes *two* recursive calls. Are there better ways to compute Fibonacci numbers?

4.6 Lists and Arithmetic

Summing a list combines arithmetic with list processing:

```
sumlist([], 0).              % sum([]) = 0
sumlist([X|Xs], Sum) :-      % sum([X|Xs]) = Sum if
    sumlist(Xs, Sum1),       % sum(Xs) = Sum1 and
    Sum is Sum1 + X.         % Sum = Sum1 + X
```

This can be done more efficiently:

```
sumlist([], Sum, Sum).       % sum([]) + Sum = Sum
sumlist([X|Xs], Sum0, Sum):-% sum([X|Xs]) + Sum0 = Sum if
    Sum1 is Sum0 + X,        % Sum1 = Sum0 + X, and
    sumlist(Xs, Sum1, Sum)% sum(Xs) + Sum1 = Sum
```

See the lecture on efficiency for more on this

4.6.1 List Maximum

Compute the maximum element of a list

```
maxlist([X], X).
maxlist([X|Xs], Max) :-
    maxlist(Xs, Max1),
    maximum(Max1, X, Max).
```

Note: no clause for []! Base case is singleton list because [] *has* no maximum

4.6.2 List Length

The `length/2` built-in relates a list to its length

```
?- length([a,b,c], 3).
Yes

?- length([a,b,c], N).
N = 3 ;
No

?- length(L, 3).
L = [_G263, _G266, _G269] ;
No
```

```
?- length(L, N).
L = []
N = 0 ;

L = [_G278]
N = 1


Yes
```

It is not straightforward to implement `length/2` with flexible modes; covered later

4.6.3 List Handling

Many other list predicates can be defined using `append`, `length`.

`member/2` could be defined just in terms of append:

```
member(X, L) :-          % X is a member of L, if L of form
  append(_, [X|_], L). % any list ++ [X] ++ any list
```

`E` is the `N`th element of list `L` (counting from 0):

```
element(N, L, E) :-    % E is the Nth element of L
  append(F, [E|_], L), % if exists F where L = F ++ [E] ++ any
  length(F, N).        % and the length of F is N
```

List Handling (2)

T is a list taking the first N elements of L:

```
take(N, L, T) :-         % T is the first N elements of L
        append(T, _, L), % if L = T ++ any list
        length(T, N).    % and the length of T in N
```

T is what is left after removing the first N elements of L:

```
drop(N, L, T) :-         % T is the first N elements of L
        append(F, T, L), % if L = T ++ any list
        length(F, N).    % and the length of T in N
```

L is a list all of whose elements are E:

```
listof([], _).           % [] is a list of anything
listof([E|Es], E) :-     % [E|Es] is a list of Es if
        listof(Es, E).   % Es is
```

5 Coding in Prolog

We work through examples to illustrate how to write Prolog code

Our goal: support code for a computerized card game

1. Data and Representation
2. Generating a deck of cards
3. Shuffling
4. Dealing

5.1 Data

For any language, first determine:

- what data we manipulate,
- how they are related,
- what are their attributes, and
- what are their primitive operations

We need to represent a (partial) deck of cards, a hand, individual cards, suits, ranks

Assume we don't need jokers for this game

Data (2)

Suits and Ranks have no attributes

Their primitive operations are only distinguishing them and possibly enumerating them

A card comprises a suit and a rank — those are its attributes

Constructing a card from rank and suit, and finding a card's rank and suit are its only primitive operations

A deck is just an (ordered) sequence of cards

Primitive operations include removing the top card or adding a card on top; all other operations can be implemented in terms of those

5.2 Representation

Simplest Prolog representations for types without attributes are atoms and numbers, *e.g.*, `hearts`, `jack`, 10, *etc*

Good idea to write a predicate for each type to specify what are members of the type

Same code can enumerate values

```
suit(clubs).      suit(diamonds).
suit(hearts).     suit(spades).

rank(2).    rank(3).    rank(4).
rank(5).    rank(6).    rank(7).
rank(8).    rank(9).    rank(10).
rank(jack).       rank(queen).
rank(king).       rank(ace).
```

Representation (2)

Simplest Prolog representation for type *with* attributes is compound term, one argument for each attribute

For playing card: `card`(R, S), where R is a rank and S is a suit

```
card(card(R,S)) :-
        suit(S),
        rank(R).
```

Since `rank` and `suit` will enumerate the ranks and suits, this code will enumerate the cards of a deck

Representation (3)

Deck is just a sequence of any number of cards

Prolog has good support for lists — good representation for sequences

List is a deck if each element is a card, *i.e.*, if empty, or if first element is a card and rest is a deck

```
deck([]).
deck([C|Cs]) :-
        card(C),
        deck(Cs).
```

This representation does not ensure a full deck, nor that there are no repeated cards

Exercise: Bridge Hand Evaluation

In Bridge, a hand has 4 high card points for each ace, 3, for each king, 2 for each queen, and 1 for each jack.

Write a predicate to determine how many high card points a hand has.

5.3 Generating a Full Deck of Cards

deck predicate holds for any deck; we need a *full* deck

card predicate generates all cards in a deck:

```
?- card(C).
C = card(2, clubs) ;
C = card(3, clubs) ;
C = card(4, clubs) ;
C = card(5, clubs) ;
C = card(6, clubs) ;
C = card(7, clubs)
```

Just need a way to collect all solutions to a query

5.3.1 **All Solutions Predicates**

Prolog has built-in predicates for collecting solutions

`findall(`T, G, L**`)`** — L is a list of all terms T that satisfy goal G in the order solutions are found; variables in G are left uninstantiated; deterministic

`setof(`T, G, L**`)`** — L is a *non-empty* list of all terms T that satisfy goal G; L is sorted with duplicates removed; variables appearing only in G can be instantiated; can be nondeterministic (with different variable instantiations)

`bagof(`T, G, L**`)`** — like **`setof/3`** but without the sorting

NB: Template term T is left uninstantiated by all of these predicates

All Solutions Predicates (2)

```
?- bagof(C, card(C), Deck).

C = _G151
Deck = [card(2, clubs), card(3, clubs), card(4, clubs),
        card(5, clubs), card(6, clubs), card(7, clubs),
        card(8, clubs), card(9, clubs), card(..., ...)|...]

?- setof(C, card(C), Deck).

C = _G151
Deck = [card(2, clubs), card(2, diamonds), card(2, hearts),
        card(2, spades), card(3, clubs), card(3, diamonds),
        card(3, hearts), card(3, spades), card(..., ...)|...]
```

`findall/3` has the same solution: no variables occur only in G and the list of solutions is not empty

All Solutions Predicates (3)

We don’t care about order, and `card` predicate generates each card exactly once, so either will work

Use `bagof`: no point sorting if not needed

```
new_deck(Deck) :-
        bagof(C, card(C), Deck).
```

Call it `new_deck` because it only holds for lists of all 52 cards in the order of a brand new deck

NB: `setof` and `bagof` fail if goal G has no solutions — you don’t get the empty list

All Solutions Predicates (4)

NB: When goal G includes some variables not appearing in template T, `setof` and `bagof` generate a different list L for each distinct binding of those variables

```
?- setof(S, borders(State,S), Neighbors).
S = _G152
State = vic
Neighbors = [nsw, sa] ;
S = _G152
State = act
Neighbors = [nsw] ;
S = _G152
State = qld
Neighbors = [nsw, nt, sa]
```

To just get a list of states that border any state, use `State^borders(State,S)` for G

5.4 Coding Tips

Prolog lists are very flexible, used for many things

Can do a lot with just `length`/2 and `append`/3

When writing list processing predicates, often have one clause for `[]` and one for `[X|Xs]`

It is often easier to write a predicate if you pretend all arguments are *inputs*

Think of it as checking correctness of output, rather than generating output

Then consider whether your code will work in the modes you want

Exercise: Selecting the n^{th} Element

Implement `select_nth1`$(N, List, Elt, Rest)$ such that Elt is the N^{th} element of $List$, and $Rest$ is all the other elements of $List$ in order. It need only work when N is bound.

5.5 Shuffling

Don't want to faithfully simulate real shuffling — would not be random

Shuffling really means randomly permute list order

Simple method: repeatedly randomly select one card from deck as next card until all cards selected

```
shuffle([], []).
shuffle(Deck0, [C|Deck]) :-
    random_element(Deck0, C, Deck1),
    shuffle(Deck1, Deck).
```

Important: `random_element` must fail for empty deck

NB: Cannot work in reverse mode!

5.5.1 Selecting a Random Card

Need to know size of deck to know probability of selecting any one card

Then we can pick a random number n where $1 \leq n \leq \mathrm{length}$ and remove element n of the deck

Must give back remainder of deck as well as random card, since we cannot destructively remove card from deck

```
random_element(List, Elt, Rest) :-
    length(List, Len),
    Len > 0,
    random(1, Len, Rand),
    select_nth1(Rand, List, Elt, Rest).
```

5.5.2 Random Numbers

No such thing as a *random number*; it's how the number is selected that is random

No standard random number predicate in Prolog

Good practice: define reasonable interface, then look in manual for built-ins or libraries to implement it

This makes maintenance easier

SWI defines `random/1` function (not really a function!)

```
random(Lo, Hi, Rand) :-
    Rand is Lo+random(Hi-Lo+1).
```

5.6 Dealing

Interface for dealing:

```
deal(Deck, Numhands, Numcards, Hands, Rest)
```

`Hands` is a list of `Numhands` lists, each with `Numcards` cards

All cards come from the front of `Deck`; `Rest` is leftover cards

Dealing (2)

Easiest solution: take first `Numcards` cards from `Deck` for first player, next `Numcards` for next player, *etc*

Use `append` and `length` to take top (front) of `Deck`

```
deal(Deck, 0, _, [], Deck).
deal(Deck, N, Numcards, [H|Hs], Rest) :-
        N > 0,
        length(H, Numcards),
        append(H, Deck1, Deck),
        N1 is N - 1,
        deal(Deck1, N1, Numcards, Hs, Rest).
```

6 Determinism

Deterministic Prolog code is much more efficient than nondeterministic code. It can make the difference between running quickly and running out of stack space or time.

1. Prolog implementation
2. Determinacy
3. If-then-else
4. Indexing

6.1 Prolog implementation

Actual implementation of Prolog predicates is **stack-based**

```
a :- b, c.                      c :- fail.
b :- d, e.                      d :- e.
b :- e.                         e.
```

To execute `a`, we invoke `b`, then `c`

While executing `b`, we must remember to go on to `c`; Like most languages, Prolog uses a **call stack**

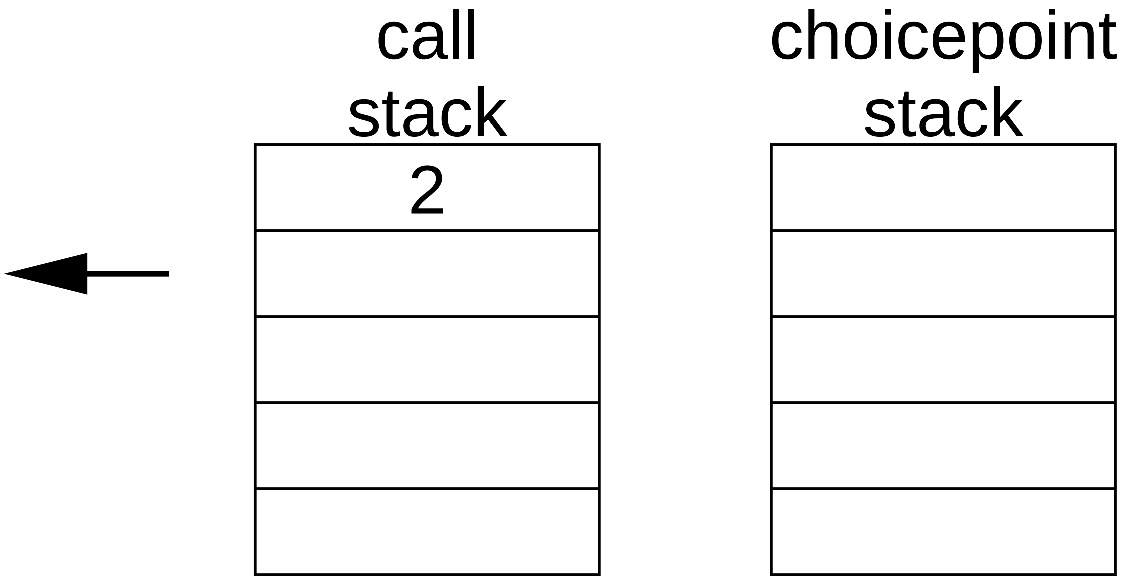
While executing first clause of `b`, we must remember if we fail, we should try the second

Prolog implements this with a separate stack, the **choicepoint stack**

When a choicepoint is created, the call stack must be frozen: cannot overwrite current call stack entries because they may be needed after choicepoint is popped

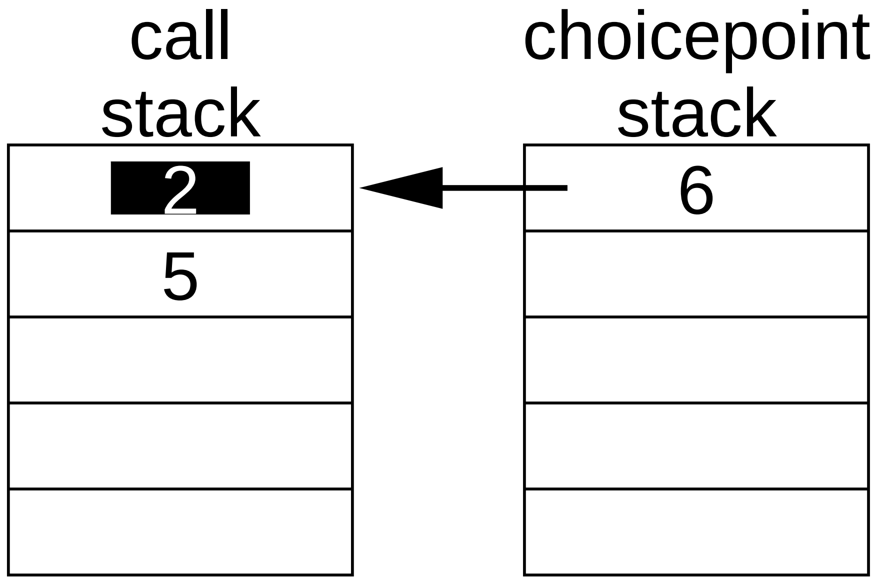
6.1.1 Stack Example

```
 0   a :-
              b,
 2            c.
 3   b :-
 4            d,
 5            e.
 6   b :-
 7            e.
 8   d :-
 9            e.
10   c :-
11            fail.
12   e.
```



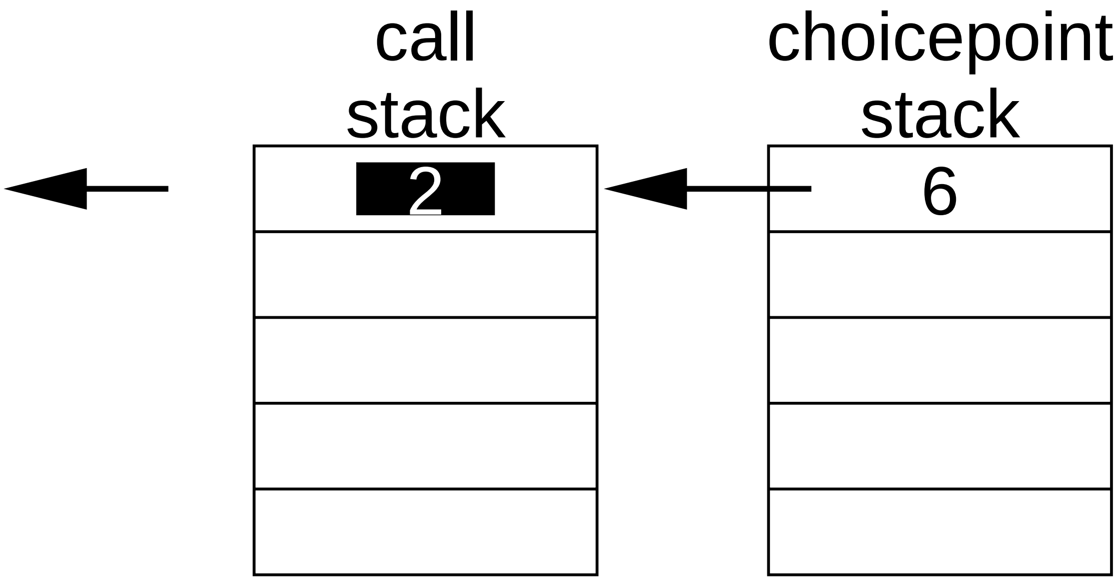
Stack Example (2)

```
 0    a :-
              b,
 2            c.
 3    b :-
 4            d,
 5            e.
 6    b :-
 7            e.
 8    d :-
 9            e.
10    c :-
11            fail.
12    e.
```



Stack Example (3)

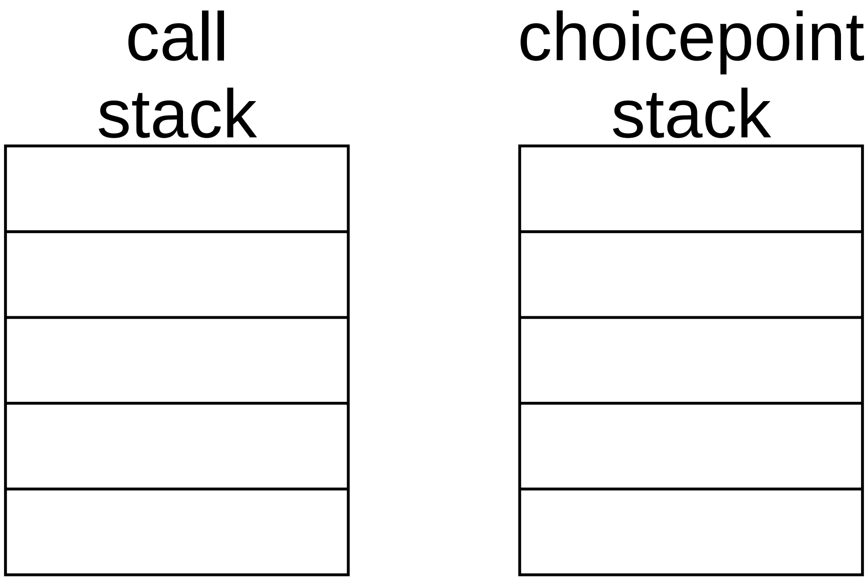
```
 0   a :-
 1          b,
 2          c.
 3   b :-
 4          d,
 5          e.
 6   b :-
 7          e.
 8   d :-
 9          e.
10   c :-
11          fail.
12   e.
```



NB: call stack not popped!

Stack Example (4)

```
 0    a :-
 1              b,
 2              c.
 3    b :-
 4              d,
 5              e.
 6    b :-
 7              e.
 8    d :-
 9              e.
10    c :-
11              fail.  ←
12    e.
```



6.2 Determinacy

Pushing a choicepoint causes current call stack to be frozen

We can’t write over frozen part of call stack, so it must be kept around indefinitely

One choicepoint can freeze a large amount of stack space

So avoiding choicepoints is important to efficiency

A predicate with multiple solutions *should* leave a choicepoint

A predicate with only one solution *should not*

6.3 Indexing

Indexing makes Prolog programs more deterministic than you might expect them to be

Indexing takes a sequence of adjacent clauses whose first argument is bound and constructs an index to quickly select the right clause(s)

When a goal has a non-variable first argument, indexing allows Prolog to immediately choose the clause(s) whose first argument matches the goal's first argument

Other clauses are not even considered

Indices are constructed automatically; no special action needed

You can ask SWI Prolog to index on arguments other than the first using the `index` or `hash` declarations

6.3.1 Indexing Example

For example, for the code

```
capital(tas, hobart).   capital(vic, melbourne).
capital(nsw, sydney).   capital(sa, adelaide).
capital(act, canberra). capital(qld, brisbane).
capital(nt, darwin).    capital(wa, perth).
```

and the goal

```
?- capital(vic, X)
```

Prolog will jump immediately to the second clause

More importantly, after trying the second clause, Prolog will know no other clause can possibly match, so will not leave a choicepoint

For a goal such as `capital(X, Y)`, the index will not be used; Prolog will try every clause

6.3.2 **Indexing for Determinism**

When possible, use indexing to avoid unnecessary choicepoints

On most Prolog systems, only the outermost constructor of the first argument is indexed, *e.g.* with code:

```
foo(bar(a), 1).  foo(bar(b), 2).  foo(bar(c), 3).
```

`foo(bar(b), X)` would still leave a choicepoint

For efficiency, rewrite to get indexing:

```
foo(bar(X), N) :- foobar(X, N).

foobar(a, 1).  foobar(b, 2).  foobar(c, 3).
```

Indexing for Determinism (2)

Indexing is used even on predicates with only two clauses: it allows many predicates to avoid choicepoints

```
append([], L, L).
append([J|K], L, [J|KL]) :-
        append(K, L, KL).
```

Indexing allows append never to leave a choicepoint when the first argument is bound

6.4 If-Then-Else

Our definition of `absolute_value/2` from slide 96 was a bit unsatisfying because both clauses compare `X` to 0:

```
absolute_value(X, X) :- X >= 0.
absolute_value(X, Y) :- X < 0, Y is -X.
```

Prolog actually performs the comparison twice

Arithmetic comparisons are cheap, but suppose the test were something time consuming: we would not want to do it twice

Conventional programming languages provide an **if-then-else** construct which evaluates the condition only once

If-Then-Else (2)

Prolog also has an if-then-else construct, whose syntax is

```
( A -> B ; C )
```

This means: execute `A`; if it succeeds, then execute `B`; otherwise, execute `C` instead

It also commits to the first solution to `A`; see **caution** below!

A form of negation defined in Prolog:

```
% "not provable"
\+ G :- (G -> fail ; true).

% a form of inequality: "not unifiable"
X \= Y :- \+ X=Y.
```

If-Then-Else (3)

Defining `absolute_value/2` with only one comparison:

```
absolute_value(X, Y) :-
        (   X < 0 ->
                Y is -X
        ;   Y = X
        ).
```

One clause which fills the role of both clauses in prev. defn.

```
?- absolute_value(-3, V).
V = 3 ;
No
?- absolute_value(42, V).
V = 42 ;
No
```

If-Then-Else (4)

If-then-elses can be nested to give an if-then-elseif-else structure.

```
comparison(X, Y, Rel) :-
        (   X < Y -> Rel = less
        ;   X = Y -> Rel = equal
        ;   Rel = greater
        ).
```

```
?- comparison(3, 4, X).
X = less ;
No
?- comparison(4, 3, X).
X = greater ;
No
?- comparison(4, 4, X).
X = equal ;
No
```

6.4.1 Caution

- If-then-else commits to the first solution to the condition goal, eliminating any other solutions
- If-then-else works by removing choicepoints
- This may limit the modes that code can be used in
- This may cause code to simply be wrong
- Use if-then-else with caution!

6.4.2 Cautionary Example

Example: `list_end(List, Sublist, End)` holds when `Sublist` is the `End` end of `List`, and `End` is `front` or `back`

Buggy implementation:

```
list_end(List, Sublist, End) :-
        (   append(Sublist, _, List) ->
                End = front
        ;   append(_, Sublist, List) ->
                End = back
        ).
```

Note no else part; this is equivalent to an else part of `fail`

Cautionary Example (2)

This code works in simple cases:

```
?- list_end([a,b,c,d], [a,b], End).
End = front ;
No
?- list_end([a,b,c,d], [d], End).
End = back ;
No
```

but gets it wrong when `Sublist` is found at both ends of `List`

```
?- list_end([a,b,c,a,b], [a,b], End).
End = front ;
No
```

The use of if-then-else means that the code will never admit that the sublist is at both ends of the list

Cautionary Example (3)

There is a bigger flaw: the if-then-else commits to the first solution of append

In many modes this is completely wrong

```
?- list_end([a,b,c], Sublist, End).

Sublist = []
End = front ;

No
?- list_end([a,b,c], Sublist, back).

No
```

Cautionary Example (4)

Correct implementation does not use if-then-else at all

```
list_end(List, Sublist, front) :- append(Sublist, _, List).
list_end(List, Sublist, back) :-  append(_, Sublist, List).
```

```
?- list_end([a,b], Sublist, End).
Sublist = []
End = front ;
Sublist = [a]
End = front ;
Sublist = [a, b]
End = front ;
Sublist = [a, b]
End = back ;
Sublist = [b]
End = back ;
Sublist = []
End = back ;
No
```

6.4.3 Removing Unnecessary Choicepoints

Suppose a **ground** list `L` represents a set

`member(foo,L)` checks if `foo`$\in$`L`, but leaves a choicepoint because there may be more than 1 way to prove `foo`$\in l$

```
    (   member(foo, L) -> true ; fail )
```

is equivalent, but will not leave a choicepoint

If we want to know whether there exists any `X` such that `foo(X)`$\in l$, and we do not care about what `X` is, then

```
    (   member(foo(_), L) -> true ; fail )
```

will stop after first match, removing choicepoint

Caution: only correct when `L` is ground

6.4.4 When to Use If-Then-Else

It's safe to use if-then-else when:

- The semantics $(a \wedge b) \vee (\neg a \wedge c)$ is what you want; and
- The condition is always ground when it is executed (be careful!)

More generally, when:

- The semantics $(\exists v_1 \exists v_2 \ldots (a \wedge b)) \vee (\forall v_1 \forall v_2 \ldots \neg a \wedge c)$ is what you want; and
- $v_1, v_2, \ldots$ are all the variables in `a` when it is executed (careful!); and
- the condition is deterministic

We will see later how if-then-else can be used to make predicates work in *more* modes than they otherwise would

7 Search

Prolog's efficient built-in search mechanism makes it ideal for solving many sorts of problems that are tricky in conventional languages

1. Path Finding
2. Iterative Deepening
3. 8 Queens
4. Constraint Programming

7.1 Path Finding

Many problems come down to searching for a path through a graph

For example, we may want to know if a person `alice` is a descendant of another person `zachary`

If we have a relation `parent`/2 that specifies who is a parent of whom, then `ancestor`/2 is the **transitive closure** of `parent`/2

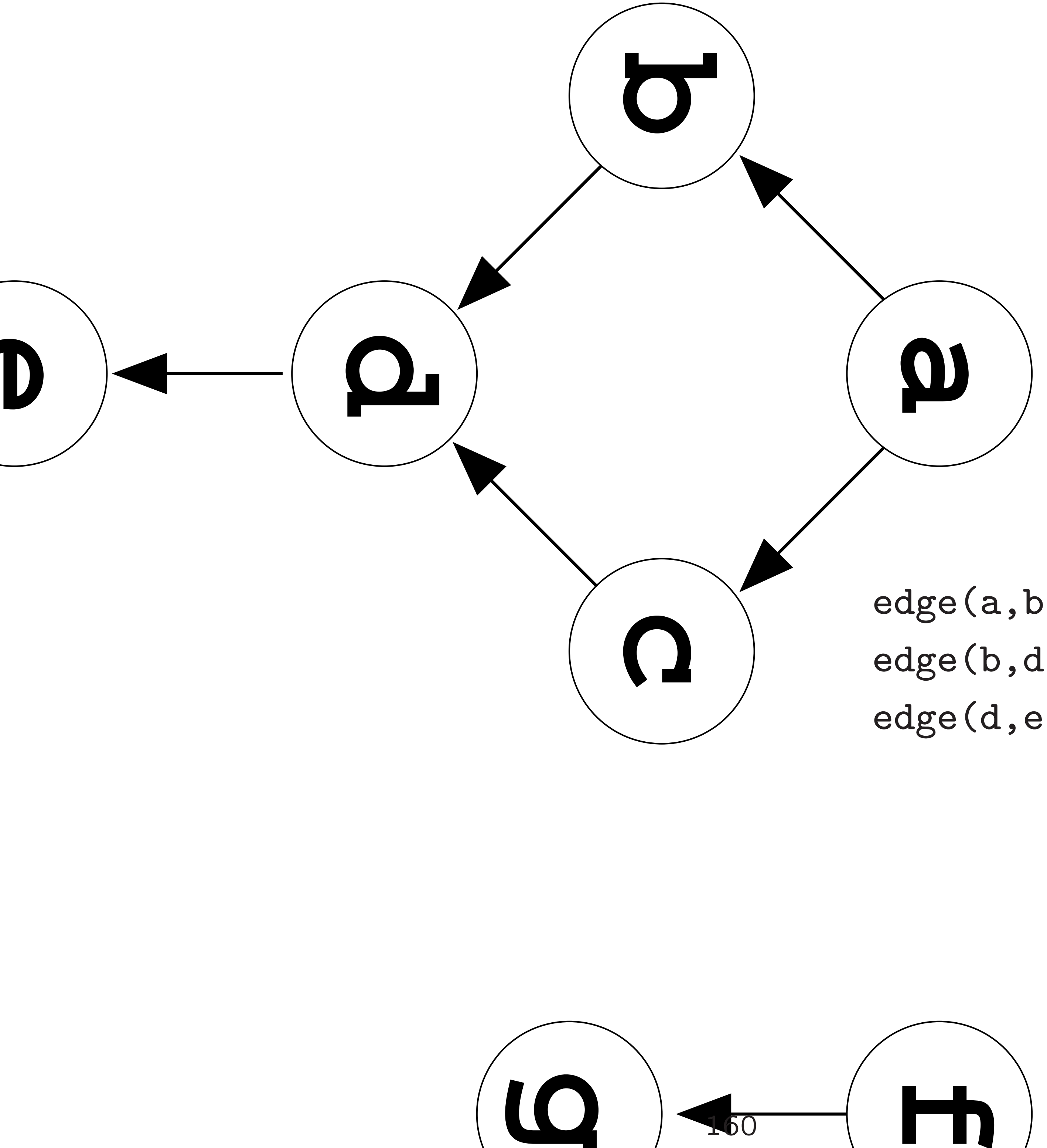
Transitive closure follows this pattern:

```
ancestor(A, A).
ancestor(D, A) :-
        parent(D, P),
        ancestor(P, A).
```

Path Finding (2)

Finding paths in a graph is finding the transitive closure of the adjacency/edge relation of the graph

Suppose a predicate `edge/2` defines a graph:

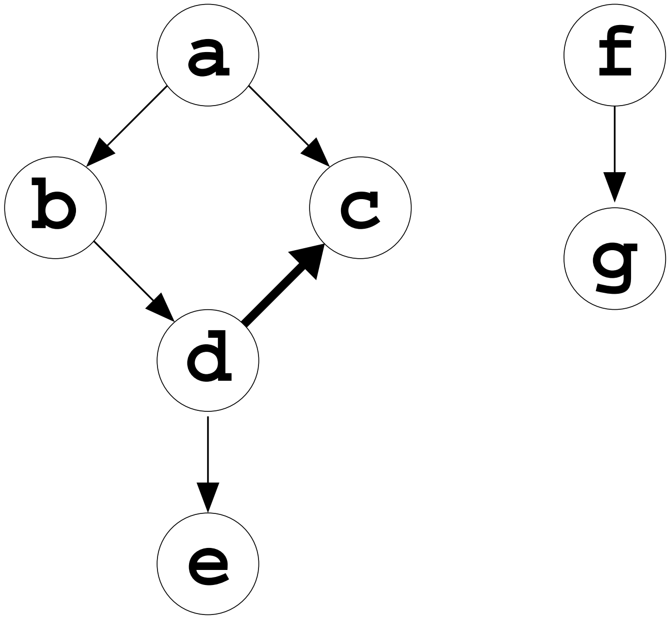


```
edge(a,b).  edge(a,c).
edge(b,d).  edge(c,d).
edge(d,e).  edge(f,g).
```

```
connected(N, N).
connected(L, N) :- edge(L, M), connected(M, N).
```

Path Finding (3)

A small change in the graph makes it go badly wrong:



```
edge(a,b).  edge(a,c).
edge(b,d).  edge(c,d).
edge(d,e).  edge(f,g).
```

The graph now has a “cycle” (cannot happen for ancestors!)

There is nothing to stop the Prolog code exploring the “infinite” path

Path Finding (4)

```
?- trace, connected(a,e).
   Call: (8) connected(a, e) ? creep
   Call: (9) edge(a, _G215) ? creep
   Exit: (9) edge(a, b) ? creep
   Call: (9) connected(b, e) ? creep
   Call: (10) edge(b, _G215) ? creep
   Exit: (10) edge(b, d) ? creep
   Call: (10) connected(d, e) ? creep
   Call: (11) edge(d, _G215) ? creep
   Exit: (11) edge(d, c) ? creep
   Call: (11) connected(c, e) ? creep
   Call: (12) edge(c, _G215) ? creep
   Exit: (12) edge(c, a) ? creep
   Call: (12) connected(a, e) ? creep
   Call: (13) edge(a, _G215) ? creep
   Exit: (13) edge(a, b) ?
```

Path Finding (5)

Solution: keep a list of nodes visited along a path, and don't revisit a node already on the list

```
connected(M, N) :- connected(M, [M], N).


connected(N, _, N).
connected(L, Path, N) :-
        edge(L, M),
        \+ member(M, Path),
        connected(M, [M|Path], N).
```

```
?- connected(a, e).


Yes
```

Remember: \+ is negation

Also note the algorithmic complexity of this code is $O(2^N)$

7.2 Word Paths

Sometimes the *path* is the desired output

Word game: transform one word into another of the same length by repeatedly replacing one letter of the word with another so that each step is a valid English word

E.g., tranform “big” to “dog”

$$\mathrm{big} \rightarrow \mathrm{bag} \rightarrow \mathrm{lag} \rightarrow \mathrm{log} \rightarrow \mathrm{dog}$$

Here we don’t just want to know if it can be done, we want to know the steps

Word Paths (2)

```
transform(Initial, Final, [Initial|Steps]) :-
        word(Initial),
        word(Final),
        transform(Initial, [Initial], Final, Steps).

transform(Final, _, Final, []).
transform(Initial, History, Final, [Next|Steps]) :-
        step(Initial, Next),
        word(Next),
        \+ member(Next, History),
        transform(Next, [Next|History], Final, Steps).
```

The second argument of `transform/4` is used to avoid loops

Word Paths (3)

```
step([_|Rest], [_|Rest]).    % all but the first is the same
step([C|Rest0], [C|Rest]) :- % the first is the same but
        step(Rest0, Rest).   % at most one letter differs in rest
```

```
word("bag").  word("big").  word("bog").  word("bug").
word("lag").  word("leg").  word("log").  word("lug").
word("dag").  word("dig").  word("dog").  word("dug").
```

Also load our `strings` module from slide 201 for readable output

Word Paths (4)

```
?- transform("big", "dog", Steps).
Steps = ["big", "dig", "dag", "bag", "lag",
         "leg", "log", "bog", "dog"] ;
Steps = ["big", "dig", "dag", "bag", "lag",
         "leg", "log", "bog", "bug"|...]
Yes
?- transform("big", "dog",
|   ["big","bag","lag","log","dog"]).
Yes
```

We'll see later how to get Prolog to print out lists of character codes as strings

7.3 Iterative Deepening

Often we don't want just any solution; we want a shortest one

Even if we don't need a shortest solution, we may want to avoid infinite (or just huge) branches in the search space

One way to do this is **breadth first search**:

1. Create a list of just a singleton list of the starting point in it

   ```
   [["big"]]
   ```

2. Repeatedly extend each element in the list with all possible next elements

   ```
   [["big","bag"], ["big","bog"], ["big","bug"], ["big","dig"]]
   ```

3. Terminate when one list is a solution

Iterative Deepening (2)

Breadth first search is often impractical due to its high memory usage

Often **Iterative Deepening** is a simple, practical, efficient alternative

Iterative deepening works by doing a depth first search for a short solution

If this works, great; if not, start over looking for a longer solution

Repeat until a solution is found

Work is repeated, but often the cost of doing a depth n search is much smaller than the cost of a depth $n + 1$ search, so the waste is relatively small

7.3.1 Word Paths Again

We can implement a predicate to find the shortest solutions to the word game as follows:

First determine the length of the shortest solution

Then commit to that length and find solutions of exactly that length

```
shortest_transform(Word0, Word, Steps) :-
        (   length(Steps0, Len),
            transform(Word0, Word, Steps0) ->
                length(Steps, Len),
                transform(Word0, Word, Steps)
        ;   fail
        ).
```

Word Paths Again (2)

```
?- time(shortest_transform("big","dog",X)).
% 100 inferences, 0.00 CPU in 0.00 seconds (0% CPU, Infinite Lips)

X = ["big", "dig", "dog"] ;

No
```

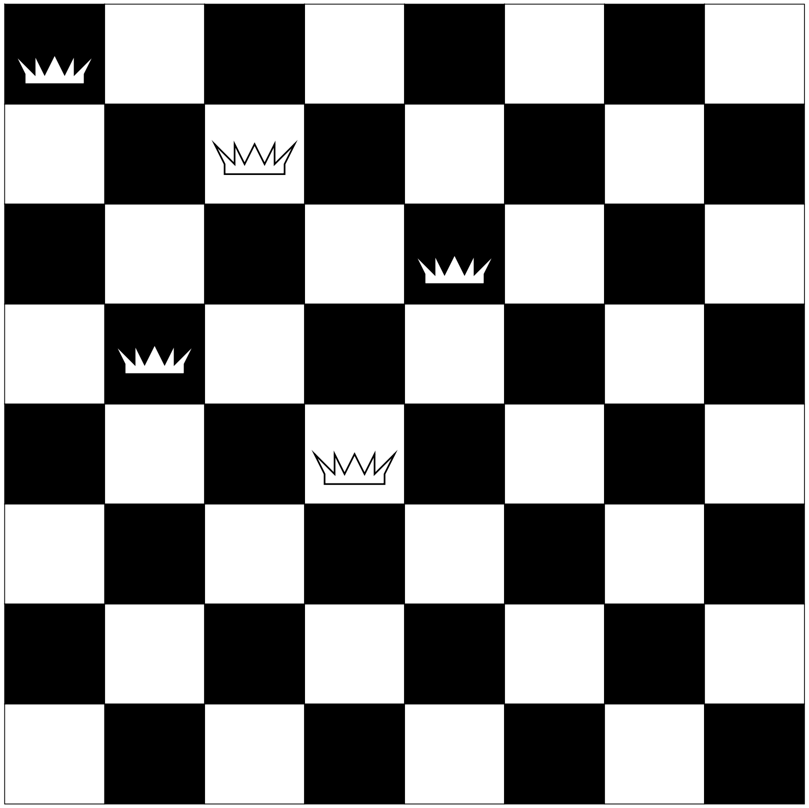
LIPS means “logical inferences per second”

One “logical inference” is defined to mean one predicate call

This code is so fast because it never tries to build long paths

7.4 8 Queens

Classic puzzle: place 8 queens on a chess board such that none attack any others.



Where to put the next queen?

8 Queens (2)

- The search space for this program is large: 64 choose 8 $= \frac{64!}{56!} \approx 10^{14}$
- If we check 1 million candidates per second, it would take 135 years to check them all
- Search space can be significantly reduced (as usual)
- One queen in every row, one in every column
- Right representation makes a big difference in search performance
- Represent board as a list of integers indicating in which column the queen in that row is placed
- Now 8 choose 8 $= 8! = 40320$ candidates — easy

8 Queens (3)

- To check if a queen attacks another queen, only need to check the diagonals
- Number rows bottom to top, columns left to right
- Queens on the same NE↔SW diagonal have same column − row
- Queens on the same SE↔NW diagonal have same column + row
- Queen's row number is position in list

8 Queens (4)

If first queen attacks second, no point placing any more, so: add queens to board one at a time, checking after each addition

Sneaky trick: add new queen in first row, sliding other queens down
Another trick: number rows from 0, so column $-$ row $=$ column $+$ row $=$ column

Check if queen in row 0 attacks queens in rows $1 \ldots n$:

```
noattack(Q, Qs) :- noattack(Q, 1, Qs).

noattack(_,  _,   []).
noattack(Q0, Row, [Q|Qs]) :-
        Q0 =\= Q + Row,
        Q0 =\= Q - Row,
        Row1 is Row + 1,
        noattack(Q0, Row1, Qs).
```

8 Queens (5)

The main body of the code chooses columns from a list of all possible columns until all queens have been placed.

Note the the last two arguments of `queens/3` are an accumulator pair.

```
queens(N, Qs) :-
    range(1, N, Columns),    % Columns = 1..8
    queens(Columns, [], Qs).

queens([], Qs, Qs).                 % no positions left, done
queens(Unplaced, Safe, Qs) :-
    select(Q, Unplaced, Unplaced1), % choose an unused position
    noattack(Q, Safe),              % check doesnt attack earlier Q
    queens(Unplaced1, [Q|Safe], Qs).% continue placing remainder
```

8 Queens (6)

The necessary utility predicates

```
select(X, [X|Ys], Ys).          % X is removed from list leaving Ys
select(X, [Y|Ys], [Y|Zs]) :-    % leave first element Y in list and
    select(X, Ys, Zs).          % select from rest of list

range(Lo, Hi, L) :-             % L is the list from Lo .. Hi
    (   Lo > Hi ->              %   if Lo > Hi this is
        L = []                  %       the empty list
    ;   L = [Lo|L1],            %   otherwise the list starts
        Next is Lo + 1,         %       with Lo and then is
        range(Next, Hi, L1)     %       the list Lo+1 .. Hi
    ).
```

8 Queens (7)

Try it:

```
?- queens(8, L).
L = [4, 2, 7, 3, 6, 8, 5, 1] ;
L = [5, 2, 4, 7, 3, 8, 6, 1] ;
L = [3, 5, 2, 8, 6, 4, 7, 1]
Yes

?- time(queens(8, L)).
% 4,691 inferences in 0.01 seconds (469100 Lips)
L = [4, 2, 7, 3, 6, 8, 5, 1]
Yes
?- time(queens(20, L)).
% 35,489,394 inferences in 211.64 seconds (167688 Lips)
```

7.5 Constraint Programming

- Much better handled by *constraint (logic) programming*.
- CP can answer 1,000,000 queens!
- Current code fails as soon as possible after generating a bad position
- Better: don't generate a bad position in the first place
- Rather than *generate and test*, employ *constrain and generate:*
  - Add all the constraints first!
  - Then search

8 Efficiency and I/O

Correctness is much more important than efficiency, but once your code is correct, you may want to make it fast. You may also want to input or output data.

1. Tail Recursion
2. Accumulators
3. Difference Pairs

8.1 Tail Recursion

For efficiency, when calling the last goal in a clause body, Prolog jumps straight to that predicate without pushing anything on the call stack

This is called **last call optimization** or **tail recursion optimization**

tail recursive means the recursive call is the last goal in the body

A tail recursive predicate is efficient, as it runs in constant stack space; it behaves like a loop in a conventional language

8.2 Accumulators

The natural definition of `factorial` in Prolog:

```
% F is N factorial

factorial(0, 1).
factorial(N, F) :-
        N > 0,
        N1 is N - 1,
        factorial(N1, F1),
        F is F1 * N.
```

This definition is not tail recursive because it performs arithmetic after the recursive call.

Accumulators (2)

We make factorial tail recursive by introducing an **accumulating parameter**, or just an **accumulator**

This is an extra parameter to the predicate which holds a partially computed result

Usually the base case for the recursion will specify that the partially computed result is actually the result

The recursive clause usually computes more of the partially computed result, and passes this in the recursive goal

The key to getting the implementation correct is specifying what the accumulator *means* and how it relates to the final result

8.2.1 Accumulator Example

A tail recursive definition of `factorial` using an accumulator:

```
factorial(N, F) :- factorial(N, 1, F).

% F is A times the factorial of N
factorial(0, F, F).
factorial(N, A, F) :-
        N > 0,
        N1 is N - 1,
        A1 is N * A,
        factorial(N1, A1, F).
```

Typical structure of a predicate using an accumulator

Accumulator Example (2)

To see how to add an accumulator, determine what is done after the recursive call

Respecify the predicate so it performs this task, too

For factorial, we compute `factorial(N1, F1), F is F1 * N`, so we want `factorial` to perform the multiplication for us too

```
% F is A times the factorial of N
new_factorial(N, A, F) :-
        factorial(N, FN),
        F is FN * A.
```

Accumulator Example (3)

Replace the call to `factorial(N, FN)` by the body: *unfold*

```
new_factorial(0, A, F) :-
    F is 1 * A.
new_factorial(N, A, F) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, F1),
    F2 is F1 * N,
    F is F2 * A.
```

Simplifying arithmetic:

```
new_factorial(0, F, F).
new_factorial(N, A, F) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, F1),
    F is (F1 * N) * A.
```

Accumulator Example (4)

By associativity of multiplication:

```
new_factorial(0, F, F).
new_factorial(N, A, F) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, F1),
    NA is N * A,
    F is F1 * NA.
```

Replace the copy of the definition of new factorial: `factorial(N1, F1), F is F1 * ?` by call to `new_factorial`: *fold*

```
new_factorial(0, F, F).
new_factorial(N, A, F) :-
    N > 0,
    N1 is N - 1,
    NA is N * A,
    new_factorial(N1, NA, F).
```

Exercise

Rewrite the `mult` predicate from an earlier exercise to be tail recursive. Here is the old code:

```
mult(0, _, 0).         % multiplying anything by 0 gives 0
 mult(s(X), Y, Z) :-% Z = Y(X + 1) if
    mult(X, Y, W),  %       W = XY
    add(W, Y, Z).   % and Z = W + Y
```

8.3 Difference Pairs

Accumulators can be even more useful for building up lists

Recall our `rev/2` predicate:

```
rev([], []).
rev([A|BC], R) :-
        rev(BC, CB),
        append(CB, [A], R).
```

The same approach works for this example

First respecify `rev` to append a list to the result:

```
% rev(BC, A, CBA)
% CBA is BC reversed with A appended
% to the end
```

8.3.1 Difference Pairs Example

Next revise the definition of `rev`

```
rev(AB, BA) :- rev(AB, [], BA).

rev([], A, A).
rev([B|CD], A, DCBA) :-
        rev(CD, [B|A], DCBA).
```

This definition is tail recursive

Much more importantly, it does not call append

If l is the length of the input list, the original version performs about $l^2/2$ steps, while the new one performs about l steps

Difference Pairs Example (2)

When working with lists, using accumulators is very common

Common to have an accumulator for each list being constructed

Thus lists are often passed as a pair of arguments: one the actual result, and the other the list to be appended to the natural result to get the actual result

Can think of the natural result of the operation as the actual result *minus* the accumulator argument

Such pairs of arguments are called **difference pairs** or **difference lists**

Common to put such pairs in the opposite order, accumulator second, to emphasize that the accumulator is the tail end of the actual result

Difference Pairs Example (3)

```
%  tree_list(Tree, List)
%  List is a list of the elements of tree,
% in order
tree_list(Tree, List) :-
        tree_list(Tree, List, []).

%  tree_list(Tree, List, List0)
%  List is a list of the elements of tree,
%  in order, followed by the list List0
tree_list(empty, List, List).
tree_list(tree(L,V,R), List, List0) :-
        tree_list(L, List, [V|List1]),
        tree_list(R, List1, List0).
```

8.4 Term I/O

```
?- write(hello).
hello
Yes
?- write(42).
42
Yes
?- write([a,b,c]).
[a, b, c]
Yes
?- write((3+(4*5)*6)).
3+4*5*6
Yes
```

`write`/1 writes any term in its normal format (considering operators), with mimimal parentheses, to current output stream.

8.4.1 Read Example

```
?- read(X).
|: [a,b,c].
X = [a, b, c] ;
No
?- read(X).
|: foo(A,3,A /* repeated variable */).
X = foo(_G231, 3, _G231) ;
No
?- read(X).
|: 7
|: .
X = 7 ;
No
```

8.5 Character I/O

An individual character can be written with the `put/1` predicate

Prolog represents characters as integers **character codes**

Can specify character code for a character by preceding it with “`0'`” (no matching close quote)

```
?- put(65), nl.
A
Yes
?- put(0'a), nl.
a
Yes
?- put(0'a), put(0' ), put(0'z).
a z
Yes
```

8.5.1 Reading Characters

The built-in `get0/1` predicate reads a single charararacter from the current input stream as a character code

`get0/1` returns -1 at end of file

```
?- get0(C1), get0(C2), get0(C3).
|: hi

C1 = 104
C2 = 105
C3 = 10 ;

No
```

8.5.2 Defining I/O Predicates

A predicate to write a line of text as a list of character codes:

```
write_line([]) :- nl.
write_line([C|Cs]) :- put(C), write_line(Cs).
```

Exercise: Read a Line of Input

Write a predicate to read a line of input as a list of character codes. `read_line(Line)` should read a line from the current input stream, binding `Line` to the list of character codes read, without the terminating newline character.

8.5.3 “String” Notation

```
?- read_line(X), write_line(X).
|: hello world!
hello world!
X = [104, 101, 108, 108, 111, 32, 119, 111, 114|...] ;
No
```

Prolog has a special syntax for lists of character codes: characters between double quotes

```
?- X = "hello, world!", write_line(X).
hello, world!
X = [104, 101, 108, 108, 111, 44, 32, 119, 111|...] ;
No
```

8.6 print and portray

Query and debugger output is done with print/1

Mostly the same as writeq/1, but you can reprogram it by defining predicate user:portray/1

```
:- multifile user:portray/1.

user:portray(Term) :-
        ground(Term),
        chars(Term),
        !,
        put(0’"),
        putchars(Term),
        put(0’").
```

```
chars([]).
chars([C|Cs]) :-
        printable(C),
        chars(Cs).

printable(C) :-
        integer(C),
        (   code_type(C, graph)
        ;   code_type(C, space)
        ).
```

putchars/1 is like write_line/1 without call to nl/0

Note: ! is called “cut”. It prevents backtracking to other clauses (use -> instead if at all possible)

print and portray (2)

With this code loaded, Prolog prints lists of character codes as quoted strings.

Without portray code

```
?-  print("abcd").
[97, 98, 99, 100]
Yes
?- X = "abcd".
X = [97, 98, 99, 100]
Yes
```

With portray code

```
?- print("abcd").
"abcd"
Yes
?- X = "abcd".
X = "abcd"
Yes
```

Code is available in strings.pl

8.7 Order Sensitivity

Like arithmetic in Prolog, I/O predicates are sensitive to the order in which they are executed

`write(hello), write(world)` always prints out `helloworld`, and never `worldhello`, although conjunction is meant to be commutative

Backtracking cannot undo I/O, e.g. `get0(0'a)` reads a character and fails if it is not an “a”, but does not put back the character if it fails

Term output puts out a term as it is, so

```
?- write(foo(X)), X=bar.
foo(_G258)

X = bar ;


No
```

```
?- X=bar, write(foo(X)).
foo(bar)

X = bar ;


No
```

8.7.1 Best Practice

Due to the procedural nature of Prolog I/O, it is usually best to keep the I/O part of an application as isolated as possible

E.g. write predicates that read in all input and build some appropriate term to represent it, then write code that processes the term

E.g. write predicates that build up output in some appropriate term representation, then have separate code to perform the actual output

This makes debugging easier: you cannot easily retry goals that read input, as retry does not automatically rewind input streams

9 Parsing in Prolog

Prolog was originally developed for programming natural language (French) parsing applications, so it is well-suited for parsing

1. DCGs
2. DCG Translation
3. Tokenizing
4. Example

9.1 DCGs

- A **parser** is a program that reads in a sequence of characters, constructing an internal representation
- Prolog's `read`/1 predicate is a parser that converts Prolog's syntax into Prolog terms
- Prolog also has a built-in facility that allows you to easily write a parser for another syntax (e.g., C or French or student record cards)
- These are called **definite clause grammars** or **DCG**s
- DCGs are defined as a number of clauses, much like predicates
- DCG clauses use `-->` to separate head from body, instead of `:-`

9.1.1 Simple DCG

Could specify a number to be an optional minus sign, one or more digits optionally followed by a decimal point and some more digits, optionally followed by the letter 'e' and some more digits:

```
number -->
        (   "-"
        ;   ""
        ),
        digits,
        (   ".", digits
        ;   ""
        ),
        (   "e", digits
        ;   ""
        ).
```

$$\begin{array}{rcl} number & \rightarrow & (\texttt{-}|\epsilon) digits (\texttt{.}\ digits|\epsilon) \\ & & (\texttt{e}\ digits|\epsilon) \end{array}$$

Simple DCG (2)

Define `digits`:

```
digits -->
        ( "0" ; "1" ; "2" ; "3" ; "4"
        ; "5" ; "6" ; "7" ; "8" ; "9"
        ),
        (   digits
        ;   ""
        ).
```

$$\begin{aligned} \mathit{digit} &\rightarrow \mathsf{0|1|2|3|4|} \\ & \mathsf{5|6|7|8|9} \\ \mathit{digits} &\rightarrow \mathit{digit\ digits}|\mathit{digit} \end{aligned}$$

A sequence of digits is one of the digit characters followed by either a sequence of digits or nothing

9.1.2 Using DCGs

Test a grammar with `phrase(Grammar,Text)` builtin, where `Grammar` is the grammar element (called a **nonterminal**) and `Text` is the text we wish to check

```
?- phrase(number, "3.14159").
Yes
?- phrase(number, "3.14159e0").
Yes
?- phrase(number, "3e7").
Yes
?- phrase(number, "37").
Yes
?- phrase(number, "37.").
No
?- phrase(number, "3.14.159").
No
```

Using DCGs (2)

It's not enough to know that `"3.14159"` spells a number, we also want to know what number it spells

Easy to add this to our grammar: add arguments to our nonterminals

Can add "actions" to our grammar rules to let them do computations

Actions are ordinary Prolog goals included inside braces `{}`

9.1.3 DCGs with Actions

DCG returns a number `N`

```
number(N) -->
        (   "-" ->
                { Sign = -1 }
        ;   { Sign = 1 }
        ),
        digits(Whole),
        (   ".", frac(Frac),
            { Mantissa is Whole + Frac }
        ;   { Mantissa = Whole }
        ),
        (   "e", digits(Exp),
            { N is Sign * Mantissa * (10 ** Exp) }
        ;   { N is Sign * Mantissa }
        ).
```

DCGs with Actions (2)

```
digits(N) -->
        digits(0, N).   % Accumulator
digits(N0, N) -->       % N0 is number already read
        [Char],
        { 0’0 =< Char },
        { Char =< 0’9 },
        { N1 is N0 * 10 + (Char - 0’0) },
        (   digits(N1, N)
        ;   { N = N1 }
        ).
```

`"c"` syntax just denotes the list with `0’c` as its only member

The two comparison goals restrict `Char` to a digit

DCGs with Actions (3)

```
frac(F) -->
        frac(0, 0.1, F).
frac(F0, Mult, F) -->
        [Char],
        { 0’0 =< Char },
        { Char =< 0’9 },
        { F1 is F0 + (Char - 0’0)*Mult },
        { Mult1 is Mult / 10 },
        (   frac(F1, Mult1, F)
        ;   { F = F1 }
        ).
```

`Mult`iplier argument keeps track of scaling of later digits.

Like `digits`, `frac` uses accumulator to be tail recursive.

Exercise: Parsing Identifiers

Write a DCG predicate to parse an identifier as in C: a string of characters beginning with a letter, and following with zero or more letters, digits, and underscore characters. Assume you already have DCG predicates `letter(L)` and `digit(D)` to parse an individual letter or digit.

9.2 DCG Translation

- DCGs are just an alternative syntax for ordinary Prolog clauses
- After clauses are read in by Prolog, they may be transformed by `expand_term(Original,Final)` where `Original` is the clause as read, and `Final` is the clause actually compiled
- Clauses with `-->` as principal functor transformed by adding two more arguments to the predicate and two arguments to each call outside `{ curley braces }`
- `expand_term/2` also calls a predicate `term_expansion/2` which you can define
- This allows you to perform any sort of translation you like on Prolog programs

9.2.1 DCG Expansion

- Added arguments form an accumulator pair, with the accumulator “threaded” through the predicate.

- nonterminal `foo(X)` is translated to `foo(X,S,S0)` meaning that the string `S` minus the tail `S0` represents a `foo(X)`.

- A grammar rule `a --> b, c` is translated to

```
          a(S, S0) :- b(S, S1), c(S1, S0).
```

- `[Arg]` goals are transformed to calls to built in predicate `’C’(S, Arg, S1)` where `S` and `S1` are the accumulator being threaded through the code.

- `’C’/3` defined as: `’C’([X|Y], X, Y).`

- `phrase(a(X), List)` invokes `a(X, List, [])`

9.2.2 DCG Expansion Example

For example, our `digits/2` nonterminal is translated into a `digits/4` predicate:

```
digits(N0, N) -->             digits(N0, N, S, S0) :-
        [Char],                       'C'(S, Char, S1),
        { 0'0 =< Char },              48 =< Char,
        { Char =< 0'9 },              Char =< 57,
        { N1 is N0*10                 N1 is N0*10
                + (Char-0'0) },             + (Char-48),
        (   digits(N1, N)             (   digits(N1, N, S1, S0)
        ;   "",                       ;   N=N1,
            { N = N1 }                    S0=S1
        ).                            ).
```

(SWI Prolog's translation is equivalent, but less clear.)

9.3 Tokenizing

- Parsing turns a linear sequence of things into a structure.

- Sometimes it's best if these "things" are something other than characters; such things are called **tokens**.

- Tokens leave out things that are unimportant to the grammar, such as whitespace and comments.

- Tokens are represented in Prolog as terms; must decide what terms represent which tokens.

Tokenizing (2)

Suppose we want to write a parser for English

It is easier to parse *words* than *characters*, so we choose words and punctuation symbols as our tokens

We will write a tokenizer that reads one character at a time until a full sentence has been read, returning a list of words and punctuation

9.3.1 A Simple Tokenizer

```
sentence_tokens(Words) :-
        get0(Ch),
        sentence_tokens(Ch, Words).
sentence_tokens(Ch, Words) :-
        (   Ch = 10 -> %% end of line
            Words = []
        ;   alphabetic(Ch) ->
            get_letters(Ch, Ch1, Letters),
            atom_codes(Word, Letters),
            Words = [Word|Words1],
            sentence_tokens(Ch1, Words1)
        ;   get0(Ch1),
            sentence_tokens(Ch1, Words)
        ).
```

A Simple Tokenizer (2)

```
alphabetic(Ch) :-
        (   Ch >= 0'a, Ch =< 0'z ->
            true
        ;   Ch >= 0'A, Ch =< 0'Z ->
            true
        ).
get_letters(Ch0, Ch, Letters) :-
        (   alphabetic(Ch0) ->
                Letters = [Ch0|Letters1],
                get0(Ch1),
                get_letters(Ch1, Ch, Letters1)
        ;   Ch = Ch0,
            Letters = []
        ).
```

9.4 Parsing Example

- Can write parser using standard approach to English grammar
- Many better approaches to parsing natural language than we use here, this just gives the idea
- An english sentence is usually a noun phrase followed by a verb phrase, e.g. “the boy carried the book”
- Noun phrase is the subject, verb phrase describes the action and the object (if there is one)
- Sentence may be an imperative: just a verb phrase, e.g. “walk the dog”

9.4.1 DCG Phrase Parser

```
sentence --> noun_phrase, verb_phrase.
sentence --> verb_phrase.

noun_phrase --> determiner, noun.
noun_phrase --> proper_noun.

verb_phrase --> verb, noun_phrase.
verb_phrase --> verb.

determiner --> word(determiner).
noun --> word(noun).
proper_noun --> word(proper_noun).
verb --> word(verb).
```

9.4.2 Parsing for Meaning

- Two problems:
    1. no indication of the meaning of the sentence
    2. no checking of agreement of number, person, etc
- Solution to both is the same: make grammar rules take arguments
- Arguments give meaning
- Unification can ensure agreement

9.4.3 Parsing for Structure

```
sentence(action(Verb,Tense,Subject,Object)) -->
        noun_phrase(Subject, Number, Person, nominative),
        verb_phrase(Verb, Tense, Number, Person, Object).
sentence(action(Verb,Tense,imperative,Object)) -->
        verb_phrase(Verb, Tense, _, second, Object).

noun_phrase(count(Thing,Definite,Number), Number, third, _) -->
        determiner(Definite,Number),
        noun(Thing, Number).
noun_phrase(Thing, Number, third, _) -->
        proper_noun(Thing, Number).
noun_phrase(pro(Person,Number,Gender), Number, Person, Case) -->
        pronoun(Person, Number, Case, Gender).
```

Parsing for Structure (2)

```
verb_phrase(Verb, Tense, Number, Person, Object) -->
        verb(Verb, Number, Person, Tense),
        (   noun_phrase(Object, _, _, objective)
        ;   { Object = none }
        ).

determiner(Definite,Number) -->
        word1(determiner(Definite,Number)).
noun(Thing, Number) -->
        word1(noun(Thing,Number)).
proper_noun(Thing,Number) -->
        word1(proper_noun(Thing,Number)).
pronoun(Person, Number, Case, Gender) -->
        word1(pronoun(Person,Number,Case,Gender)).
verb(Verb,Number,Person,Tense) -->
        word1(verb(Verb,Ending)),
        { verb_agrees(Ending, Number, Person, Tense) }.
```

9.4.4 Parsing Examples

```
?- phrase(sentence(Meaning), [the,boy,carried,the,book]).
Meaning = action(carry,past,count(boy,definite,singular),
                 count(book,definite,singular)) ;
No
?- phrase(sentence(Meaning), [walk,the,dog]).
Meaning = action(walk,present,imperative,
                 count(dog,definite,singular)) ;
No
?- phrase(sentence(Meaning), [mary,walked]).
Meaning = action(walk, past, mary, none) ;
No
?- phrase(sentence(Meaning), [mary,walk]).
No
```

9.4.5 Generation of Sentences

Carefully designed grammar can run “backwards” generating text from meaning

```
?- phrase(sentence(action(carry, past,
                          count(boy,definite,singular),
                          count(book,definite,singular))), Sentence).
Sentence = [the, boy, carried, the, book] ;
No
?- phrase(sentence(action(walk, present,
                          pro(third,plural,masculine),
                          count(dog,definite,singular))), Sentence).
Sentence = [they, walk, the, dog] ;
No
?- phrase(sentence(action(walk, present,
                          count(dog,definite,singular),
                          pro(third,plural,masculine))), Sentence).
Sentence = [the, dog, walks, them] ;
No
```

10 More on Terms

1. Handling Terms in General
2. Sorting
3. Term comparison
4. Variable and Type Testing

10.1 Handling Terms in General

Sometimes it is useful to be able to write predicates that will work on any kind of term

The =.. predicate, pronounced “univ,” turns a term into a list

The first element of the list is the term’s functor; following elements are the term’s arguments, in order

```
?- p(1,q(3)) =.. X.
X = [p, 1, q(3)] ;
No

?- T =.. [p, 1, q(3)].
T = p(1, q(3)) ;
No
```

```
?- 1 =.. X.
X = [1] ;
No

?- [1,2,3] =.. X.
X = ['.', 1, [2, 3]] ;
No
```

10.1.1 `functor/3`

Because univ builds a list that is typically not needed, it is often better to use the separate builtin predicates `functor/3` and `arg/3`

`functor(Term,Name,Arity)` holds when the functor of `Term` is `Name` and its arity is `Arity`

For atomic terms, the whole term is considered to be the `Name`, and `Arity` is 0

Either the `Term` or both the `Name` and `Arity` must be bound, otherwise `functor` throws an exception

10.1.2 `functor/3` in Action

```
?- functor(p(1,q(3)), F, A).
F = p
A = 2 ;
No

?- functor(T, p, 2).
T = p(_G280, _G281) ;
No

?- functor(1, F, A).
F = 1
A = 0 ;
No

?- functor(X, 42, 0).
X = 42 ;
No
```

10.1.3 `arg/3`

`arg/3` can be used to access the arguments of a term one at a time

`arg(N,Term,Arg)` holds if `Arg` is the $\mathtt{N}^{th}$ argument of `Term` (counting from 1)

`Term` must be bound to a compound term

For most Prolog systems, `N` must be bound; SWI Prolog will backtrack over all arguments of term

`arg/3` is deterministic when first two arguments are bound

10.1.4 arg/3 in Action

```
?- arg(2, foo(a,b), Arg).
Arg = b ;

No
?- arg(N, foo(a,b), Arg).
N = 1
Arg = a ;

N = 2
Arg = b ;

No
?- arg(N, 3, 3).
ERROR: arg/3: Type error: ‘compound’ expected, found ‘3’
```

10.1.5 Using `functor` and `arg`

Collect a list of all the functors appearing anywhere in a ground term:

```
all_functors(Term, List) :- all_functors(Term, List, []).
all_functors(Term, List, List0) :-
        functor(Term, Name, Arity),
        (   Arity > 0 ->
            List = [Name|List1],
            all_functors1(Arity, Term, List1, List0)
        ;   List = List0
        ).
all_functors1(N, Term, List, List0) :-
        (   N =:= 0 ->
            List = List0
        ;   arg(N, Term, Arg),
            all_functors(Arg, List1, List0),
            N1 is N - 1,
            all_functors1(N1, Term, List, List1)
        ).
```

Using `functor` and `arg` (2)

```
?- all_functors(foo(bar([1,2],zip,baz(3)),bar(7)), L).
L = [foo, bar, ’.’, ’.’, baz, bar] ;
No

?- X=bar(42), all_functors(foo(X), L).
X = bar(42)
L = [foo, bar] ;
No

?- all_functors(foo(X), L), X=bar(42).
ERROR: Arguments are not sufficiently instantiated
   Exception: (11) functor(_G342, _G448, _G449) ? creep
```

Since `Name` and `Arity` are not bound when `functor/3` is called, that goal causes an error if `Term` is not bound.

10.2 Sorting

Our `all_functors/2` predicate returns a list of functors appearing, once for each time they appear

May want to produce a *set*, with each functor appearing only once

Easiest way: produce the list and sort it, removing duplicates

Built-in `sort(List,Sorted)` predicate holds when `Sorted` is a sorted version of `List`, *with duplicates removed*

```
all_functors(Term, List) :-
        all_functors(Term, List0, []),
        sort(List0, List).
```

10.2.1 `keysort/2`

`keysort(List, Sorted)` is like `sort`, except:

- elements of `List` must be terms of the form `Key-Value`
- only the `Key` parts of the terms are considered when sorting
- duplicates are not removed, `keysort` is a stable sort

```
?- sort([a,a,r,d,v,a,r,k], L).
L = [a, d, k, r, v] ;
No

?- keysort([a,a,r,d,v,a,r,k], L).
No

?- keysort([a-1,a-2,r-3,d-4,v-5,a-6,r-7,k-8], L).
L = [a-1, a-2, a-6, d-4, k-8, r-3, r-7, v-5] ;
No
```

10.3 Term Comparison

The sorting predicates work on lists of any kinds of terms

Built on general term comparison predicates

The predicates @<, @=<, @>, @>=, == \== work like <, =<, >, >=, =:= =\=, but they work on any terms

Atoms are compared lexicographically, numbers numerically

Compound terms are compared first by arity, then by functor, then by the arguments in order

Variables @< numbers @< atoms @< compound terms

Beware variables in terms being sorted!

== and \== check whether terms are identical — without binding any variables

Exercise

Why should you beware variables in terms being sorted?
What could happen if you sort a non-ground list?

10.4 Variable and Type Testing

Earlier we saw that the simple definition of `rev` used in the "backwards mode" goes into an infinite loop after finding the correct answer

```
rev([], []).
rev([A|BC], R) :-
        rev(BC, CB),
        append(CB, [A], R).
```

Recursive call to rev produces longer and longer reversed lists; once it has found the right length, no other length will be correct

10.4.1 Variable Testing Example

Can fix this by exchanging the calls in the clause body

```
rev([], []).
rev([A|BC], R) :-
        append(CB, [A], R),
        rev(BC, CB).
```

In this order, when `R` is bound, the call to `append`/3 has only one solution, so it solves the problem

But now `rev`/2 gets in trouble in the “forwards mode”

Once it finds the solution, asking for more solutions enters an infinite loop

10.4.2 Variable Testing

To implement a *reversible* version of `rev/2`, we need to decide which order to execute the goals based on which arguments are bound.

Prolog has a built in predicate `var/1` which succeeds if its argument is an unbound variable, and fails if not.

```
?- var(_).


Yes
?- var(a).


No
```

Exercise

Use the `var` builtin to implement a version of `rev` that will work in any mode. Recall that only this version of `rev`:

```
rev([], []).
rev([A|BC], R) :- rev(BC, CB), append(CB, [A], R).
```

works when the second argument is unbound; otherwise the goals in the body should be swapped

10.4.3 What's Wrong With `var/1`?

One of the most basic rules of logic is that conjunction is commutative

`p, q` should behave exactly the same way as `q, p` (if they terminate without error).

`var/1` breaks that rule:

```
?- var(X), X = a.
X = a ;
No

?- X = a, var(X).
No
```

10.4.4 When to use `var/1`

Irony: sometimes we can only write logical, reversible predicates by using an *extralogical* predicate such as `var/1`

But we must be careful

For `rev/2`, the code is logically equivalent whether the `var/1` goal succeeds or fails; the only difference is the order of the goals

Sometimes we must have other difference, such as having

```
        X is Y + 1,
```

when Y is bound, and

```
        Y is X - 1,
```

when X is.

A better answer for this is *constraint programming*
$X = Y + 1$

10.4.5 Other Non-logical Builtins

All of the following have the same problem of potentially making conjunction not commutative

`nonvar(X)` `X` is bound.

`ground(X)` `X` is bound, and every variable in the term it is bound to is also ground.

`atom(X)` `X` is bound to an atom.

`integer(X)` `X` is bound to an integer.

`float(X)` `X` is bound to a float.

`number(X)` `X` is bound to a number.

`atomic(X)` `atom(X) ; number(X).`

`compound(X)` `X` is bound to a compound term.

10.4.6 **Defining** `length`/2

We can define `length`/2 to work when the list is supplied:

```
len1([], 0).
len1([_|L], N) :-
        len1(L, N1),
        N is N1 + 1.
```

and when the length is supplied:

```
len2([], 0).
len2([_|L], N) :-
        N > 0,
        N1 is N - 1,
        len2(L, N1).
```

Defining `length/2` (2)

A version that works in both modes:

```
len(L,N) :-
        (   var(N) ->
                len1(L,N)
        ;   len2(L,N)
        ).
```

NB: don't check if `L` is `var`: even if it's not, its tail may be!

Defining length/2 (3)

```
?- len(L, 4).

L = [_G242, _G248, _G254, _G260] ;

No
?- len([a,b,c], N).

N = 3 ;

No
```

```
?- len(L, N).

L = []
N = 0 ;

L = [_G257]
N = 1 ;

L = [_G257, _G260]
N = 2

Yes
```

11 Metaprogramming

Prolog has powerful facilities for manipulating and generating Prolog programs

1. Higher order programming
2. Interpreters
3. Prolog Interpreter
4. Generating Prolog code

11.1 Higher order programming

One powerful feature of functional languages is **higher order programming**: passing functions or predicates as arguments

Prolog supports this, too

The built-in predicate `call/1` takes a term as argument and executes it as a goal

```
?- Goal = append([a,b,c],X,[a,b,c,d,e]),
|  call(Goal).
Goal = append([a, b, c], [d, e], [a, b, c, d, e])
X = [d, e] ;
No
```

We can use any code we like to build goal term

11.1.1 call/n

Some Prolog systems, including SWI, also support `call` of higher arities

First argument is goal to execute

Extra arguments are given as arguments to the goal

```
?- Goal = append,
|  call(Goal, [a,b,c], X, [a,b,c,d,e]).
Goal = append
X = [d, e] ;
No
```

Saves the effort of constructing the goal as a term before calling it

11.1.2 Closures

When the goal argument of `call/n` is a compound term, later arguments of `call` are *added* to the goal

```
?- Goal = append([a,b,c]),
|  call(Goal, X, [a,b,c,d,e]).
Goal = append([a, b, c])
X = [d, e] ;
No
```

The `Goal` in this case is a predicate with some, but not all, of its arguments supplied; other arguments are given when it is called

This is a **closure**

11.1.3 map/3

Using call/n it is easy to implement the standard higher order operations of functional languages, *e.g.*

```
map(_, [], []).
map(P, [X|Xs], [Y|Ys]) :-
        call(P, X, Y),
        map(P, Xs, Ys).
```

```
?- map(append([a,b]), [[c,d],[e,f,g],[h]], L).
L = [[a, b, c, d], [a, b, e, f, g], [a, b, h]] ;
No
?- map(append([a,b]), L,
        [[a,b,c,d],[a,b,c],[a,b,w,x,y],[a,b]]).
L = [[c, d], [c], [w, x, y], []] ;
No
```

Exercise: `all/2`

Write a predicate `all/2` such that `all(Pred,L)` holds when every element of L satisfies `Pred`. For example,

```
    all(member(c), [[a,b,c],[a,e,i],[a,c]]]
```

fails, and

```
    all(member(X), [[a,b,c],[a,e,i],[a,c]])
```

succeeds with `X=a`.

11.2 **Interpreters**

Some programming problems are best solved by defining a new **minilanguage**, a small language targeted at one specific task

Consider the control of a robot vacuum cleaner

Programming this would be rather complex, involving separate control of several motors, receiving input from several sensors, *etc*.

This task would be far simpler to handle if we defined a minilanguage for robot control, and then implemented an interpreter for this language

11.2.1 Robot Minilanguage

At the simplest level, we can consider primitive commands to turn and move forward

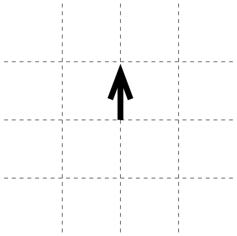
These could be represented as Prolog terms:

```
advance(Distance)
left
right
```

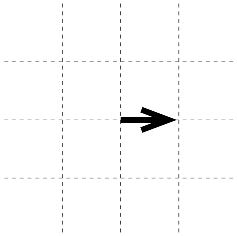
11.2.2 Interpreter Design

An interpreter (simulator) for this language would track the robot's position

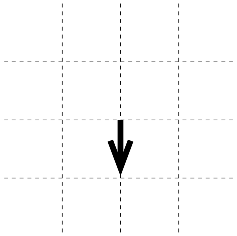
Also needs to keep track of the robot's heading, represented as X and Y components, called H_x and H_y



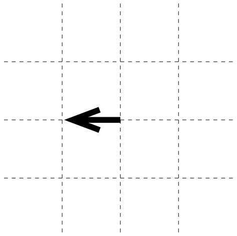
$H_x = 0$
$H_y = 1$



$H_x = 1$
$H_y = 0$



$H_x = 0$
$H_y = -1$



$H_x = -1$
$H_y = 0$

H_x and H_y are either -1, 0, or 1, and $|H_x| + |H_y| = 1$

On moving d units, $\Delta x = dH_x$ and $\Delta y = dH_y$

To turn clockwise (right): $H'_x = H_y, \quad H'_y = -H_x$,
anticlockwise (left): $H'_x = -H_y, \quad H'_y = H_x$

11.2.3 Interpreter

```
%  sim(Cmd, X0, Y0, Hx0, Hy0, X, Y, Hx, Hy)
%  Robot starts at position X0, Y0, with
%  Heading Hx0, Hy0.  After command Cmd,
%  robot is at position X, Y, with heading
%  Hx, Hy

sim(advance(Dist), X0, Y0, Hx, Hy, X, Y, Hx, Hy) :-
        X is X0 + Hx * Dist,
        Y is Y0 + Hy * Dist.
sim(right, X, Y, Hx0, Hy0, X, Y, Hx, Hy) :-
        Hx = Hy0,
        Hy is -Hx0.
sim(left, X, Y, Hx0, Hy0, X, Y, Hx, Hy) :-
        Hx is -Hy0,
        Hy = Hx0.
```

11.2.4 Robot Programs

Construct a robot program as a list of commands:

```
sim([], X, Y, Hx, Hy, X, Y, Hx, Hy).
sim([C|Cs], X0, Y0, Hx0, Hy0, X, Y, Hx, Hy) :-
        sim(C, X0, Y0, Hx0, Hy0, X1, Y1, Hx1, Hy1),
        sim(Cs, X1, Y1, Hx1, Hy1, X, Y, Hx, Hy).
```

11.2.5 Walls

The interpreter is simple, so it is easy to extend

Add walls to the room the robot operates in by preventing X and Y from being less than 0 or more than room width or height

Just need to replace one sim/9 clause and define room width and height

```
sim(advance(Dist), X0, Y0, Hx, Hy, X, Y, Hx, Hy) :-
        room_width(Width),
        room_height(Height),
        X is max(0, min(Width, X0 + Hx * Dist)),
        Y is max(0, min(Height, Y0 + Hy * Dist)).

room_width(400).
room_height(300).
```

11.2.6 Other Possible Extensions

- Keep track of minimum and maximum x and y positions visited, by adding 8 more arguments for previous and new min and max x and y
- Allow robot to turn at angles other than $90°$, by adding a `turn(Angle)` command that uses trigonometry to determine new real number H_x and H_y
- Add obstacles to room by defining predicate `obstacle` that supplies shapes and positions of each obstacle, and having `advance` command find position of first obstacle encountered

11.3 Prolog meta-interpreter

We can write an interpreter for any language we can conceive, including Prolog itself

This is made much easier by the fact that Prolog goals are ordinary Prolog terms

An interpreter for a language written in the language itself is called a **meta-interpreter**

Prolog systems have a built-in interpreter invoked by the `call` built-in predicates

Need access to the clauses of the program to be interpreted; built-in predicate `clause(Head,Body)` gives this

`Body` is `true` for unit clauses

Generally you need to declare predicates "dynamic":

```
:- dynamic foo/3.
```

11.3.1 Simple meta-interpreter

```
%  solve(Goal)
%  interpret Goal, binding variables
%  as needed.

solve(true).
solve((G1,G2)) :-
        solve(G1),
        solve(G2).
solve(Goal) :-
        clause(Goal, Body),
        solve(Body).
```

Simple meta-interpreter (2)

Extend this to handle disjunction, if->then;else, negation:

```
solve((G1;G2)) :-
        (   G1 = (G1a->G1b) ->
                (   solve(G1a) ->
                        solve(G1b)
                ;   solve(G2)
                )
        ;   solve(G1)
        ;   solve(G2)
        ).
solve(\+(G)) :-
        \+ solve(G).
```

Simple meta-interpreter (3)

To handle Prolog builtins, modify the last clause on slide 267:

```
solve(Goal) :-
        (   builtin(Goal) ->
                call(Goal)
        ;   clause(Goal, Body),
            solve(Body)
        ).

builtin(_ = _).
builtin(_ is _).
builtin(append(_,_,_)).

       ⋮
```

11.3.2 Extensions

Can modify interpreter to construct a proof

```
proof(Goal, Proof) :- proof1(Goal, Proof, []).


proof1(true, Proof, Proof).
proof1((G1,G2), Proof, Proof0) :-
        proof1(G1, Proof, Proof1),
        proof1(G2, Proof1, Proof0).
proof1(Goal, Proof, Proof0) :-
        clause(Goal, Body),
        proof1(Body, Proof,
               [(Goal:-Body)|Proof0]).
       ⋮
```

Lots of other extensions, *e.g.* debuggers, other search strategies

11.4 Manipulating Prolog code

The fact that Prolog code is just a Prolog term also makes it very easy to manipulate

To take advantage of this, the Prolog compiler automatically calls a predicate `term_expansion`/2 (if it is defined) for each term it reads

First argument of `term_expansion`/2 is a term Prolog read from a source file; `term_expansion`/2 should bind the second to a clause or list of clauses to use in place of what was read

This is the same mechanism used to convert DCG clauses into ordinary Prolog clauses

This feature allows programmers to easily write Prolog code to generate or transform Prolog code

11.4.1 Generating Accessor Predicates

Example: automatically generating accessor predicates

Suppose a student is represented as `student(Name,Id)` in a large program

Everywhere the name is needed, the code might contain

```
        Student = student(Name,_)
```

If we later need to also store degree code in student terms, we must modify every `student`/2 term in the program

Programmers sometimes define accessor predicates, *e.g.*:

```
        student_name(student(Name, _), Name).
        student_id(student(_, Id), Id).
```

then use `student_name(Student, Name)` instead of `Student = student(Name,_)`

Generating Accessor Predicates (2)

Using `term_expansion`, it is easy to write code to automatically generate accessor predicates

We will transform a directive like

```
        :- struct student(name,id).
```

into the clauses on the previous slide

```
:- op(1150, fx, (struct)).

term_expansion((:- struct Term), Clauses) :-
        functor(Term, Name, Arity),
        functor(Template, Name, Arity),
        gen_clauses(Arity, Name, Term, Template, Clauses).
```

First line tells Prolog that `struct` is a prefix operator

Generating Accessor Predicates (3)

```
gen_clauses(N, Name, Term, Template, Clauses) :-
        (   N =:= 0 ->
                Clauses = []
        ;   arg(N, Term, Argname),
            arg(N, Template, Arg),
            atom_codes(Argname, Argcodes),
            atom_codes(Name, Namecodes),
            append(Namecodes, [0’_|Argcodes], Codes),
            atom_codes(Pred, Codes),
            Clause =.. [Pred, Template, Arg],
            Clauses = [Clause|Clauses1],
            N1 is N - 1,
            gen_clauses(N1, Name, Term, Template, Clauses1)
        ).
```

12 Propositional Logic

- introduce basic ideas of formal logic
- define truth tables for propositions
- use of truth tables to establish tautologies, equivalences

Exercise: Knights and Knaves

On the island of Knights and Knaves, everyone is a knight or knave. Knights always tell the truth. Knaves always lie. You are a census taker, going from house to house. Fill in what you know about each house.

house 1 Husband: We are both knaves.

house 2 Wife: At least one of us is a knave.

house 3 Husband: If I am a knight then so is my wife.

House 1		House 2		House 3	
Husband	Wife	Husband	Wife	Husband	Wife

12.1 Logic and Reasoning

- Logic is about reasoning

```
All humans have 2 legs
Jane is a human
Therefore Jane has 2 legs
```

- key issue for (any) logic

FORM of an argument *versus* its MEANING

```
All dogs have 6 legs
Rex is a dog
Therefore Rex has 6 legs
```

12.1.1 **Arguments: Form and Meaning**

- “form” is determined by the syntax of the logic.
  Both arguments above have correct form, HENCE
  both arguments are (logically) correct.

- “truth” depends on “meaning” – relation to the world.
  The first argument has a true conclusion: its premises
  are true and the argument is correct.
  The second argument has a false conclusion: even
  though the argument is correct, its premises are not
  true.

$\boxed{Q}$ 12.1.2 **Propositions**

- A **proposition** is a sentence that is either true or false, but not both.
- use letters to represent basic propositions
- e.g. A for "I look into the sky", B for "I am alert"
- logical connectives: $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), $\leftrightarrow$ (iff)
- e.g. $A \wedge B$ for "I look into the sky and I am alert"
- rules for forming propositions

(a) a propositional variable is a proposition

(b) if A and B are propositions then so are
$(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, $(A \leftrightarrow B)$

(c) nothing else is a proposition

12.1.3 **Improving Readability**

Which is more readable?

$$(P \to (Q \to (\neg R)) \text{ or } P \to (Q \to \neg R)$$

Rules for improving readability:

- omit parentheses where possible
- precedence from highest to lowest is: $\neg$, $\wedge$, $\vee$, $\to$, $\leftrightarrow$
- all binary operators are left associative.
  That is,
  $P \to Q \to R$
  is an *abbreviation* for
  $(P \to Q) \to R$

Questions:

Is $(P \vee Q) \vee R \equiv P \vee (Q \vee R)$? Is $(P \wedge Q) \wedge R \equiv P \wedge (Q \wedge R)$?
Is $(P \to Q) \to R \equiv P \to (Q \to R)$?

$\boxed{Q}$ 12.2 Truth Tables

- propositions can be true (T) or false (F)
- A *truth table* for a proposition records its truth value for all assignments of T or F to its propositional variables. (see Kelly Section 1.2) e.g.

A	B	$A \wedge B$	$A \vee B$	$\neg A$
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

- value of (compound) propositions built up from components
  i.e. A and B can be any (compound) proposition

12.2.1 Terminology

A proposition is a **tautology** if it is always true, *i.e.* each line in the truth table is T. For example $A \vee B \vee \neg A$.

A proposition is **satisfiable** if it is sometimes true, *i.e.* some line in the truth table is T. For example, $A \wedge B$.

A proposition is **unsatisfiable** if it is always false, *i.e.* each line in the truth table is F. For example, $A \wedge B \wedge \neg A$.

Two propositions are **equivalent** if each line in their truth tables are the same. For example A is equivalent to $A \vee (A \wedge A)$.

$\boxed{Q}$

12.2.2 Previous Example

Does $(A \wedge B) \wedge C = A \wedge (B \wedge C)$?

A	B	C	$A \wedge B$	$B \wedge C$	$A \wedge (B \wedge C)$	$(A \wedge B) \wedge C$
T	T	T	T	T	T	T
T	T	F	T	F	F	F
T	F	T	F	F	F	F
T	F	F	F	F	F	F
F	T	T	F	T	F	F
F	T	F	F	F	F	F
F	F	T	F	F	F	F
F	F	F	F	F	F	F

Truth table columns for both propositions are identical: propositions are equivalent

$\boxed{Q}$ 12.3 **Implication**

Truth Table For Implication:

A	B	$A \rightarrow B$
T	T	T
T	F	F
F	T	T
F	F	T

Terminology:

- **antecedent**: what's to the left of the $\rightarrow$
- **consequent**: what's to the right of the $\rightarrow$

Implication is true unless antecedent is true and consequent is false

$\boxed{Q}$ Exercise

Are $(A \rightarrow B)$ and $\neg A \vee B$ are equivalent?

$\boxed{Q}$ 12.3.1 Material Implication

- The only time $A \rightarrow B$ evaluates to false is when A is true and B is false.

- This definition is standard in propositional logic and is termed **material implication**.

- English usage often suggests a causal connection between antecedent and consequent; this is not reflected in the truth table for material implication

- $(P \wedge \neg P) \rightarrow (Q \rightarrow R)$ is a tautology and worse still, $(P \wedge \neg P) \rightarrow \{anything\}$ is a tautology.

- Other logics (e.g. modal logics, relevance logics) involve alternative meanings for implication

$\boxed{Q}$ 12.3.2 Example (Kelly Example 1.2)

- Express in propositional logic:
  "If I look into the sky and I am alert then either I will see the flying saucer or if I am not alert then I will not see the flying saucer."

A	"I look into the sky"
B	"I am alert"
C	"I will see the flying saucer"

$$(A \wedge B) \rightarrow (C \vee (\neg B \rightarrow \neg C))$$

- does the truth of this compound proposition depend on the individual truth values of A, B, C?

  That is, is $(A \wedge B) \rightarrow (C \vee (\neg B \rightarrow \neg C))$ a tautology?

$\boxed{Q}$ Example (Kelly Example 1.2) (2)

Truth Table (Kelly Fig. 1.13)

A	B	C	$A \wedge B$	$\neg B$	$\neg C$	$\neg B \rightarrow \neg C$	$C \vee (\neg B \rightarrow \neg C)$	$(A \wedge B) \rightarrow (C \vee (\neg B \rightarrow \neg C)$
T	T	T	T	F	F	T	T	T
T	T	F	T	F	T	T	T	T
T	F	T	F	T	F	F	T	T
T	F	F	F	T	T	T	T	T
F	T	T	F	F	F	T	T	T
F	T	F	F	F	T	T	T	T
F	F	T	F	T	F	F	T	T
F	F	F	F	T	T	T	T	T

Example (Kelly Example 1.2) (3)

- The truth table contains much unnecessary information.
- How can we reduce our work?
- Remember that $X \to Y$ is false iff X is true and Y is false and $A \land B$ is false unless A is true and B is true. So we only need to look at the last two lines of the truth table (the final expression in the other six is always true).
- Thus:

A	B	C	$A \land B$	$\neg B$	$\neg C$	$\neg B \to \neg C$	$C \lor (\neg B \to \neg C)$	$(A \land B) \to (C \lor (\neg B \to \neg C))$
T	T	T	T	F	F	T	T	T
T	T	F	T	F	T	T	T	T

is the simplified truth table for this example.

$\boxed{Q}$ Exercise

Express in propositional logic:
“If Michelle wins at the Olympics, everyone will admire her, and she will get rich; but if she does not win, all her effort was in vain.” Begin by determining the the propositional variables you will use and what they mean.

$\boxed{Q}$ 12.3.3 Knights and Knaves

Express in propositional logic the census taker puzzle, for house 1.

- H : the husband is a knight.
- W : the wife is a knight.

$$H \leftrightarrow (\neg H \wedge \neg W)$$

H	W	$\neg H \wedge \neg W$	$H \leftrightarrow (\neg H \wedge \neg W)$
T	T	F	F
T	F	F	F
$\mathbf{F}$	$\mathbf{T}$	F	$\mathbf{T}$
F	F	T	F

Exercise: do the same for houses 2 and 3.

12.4 Equivalences (Kelly Section 1.3)

- lots of them ... infinitely many!

- especially important:
  - $\neg\neg A$ is equivalent to A
  - $\neg A \vee B$ is equivalent to $A \rightarrow B$
  - $\neg(A \wedge B)$ is equivalent to $(\neg A \vee \neg B)$
  - $\neg(A \vee B)$ is equivalent to $(\neg A \wedge \neg B)$

- The final two equivalences are **De Morgan's laws**

- Exercise: Use truth tables to verify the correctness of the above four equivalences.

12.4.1 More Important Equivalences

Commutativity	$B \vee A \equiv A \vee B$
	$A \wedge B \equiv B \wedge A$
Associativity	$A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$
	$A \vee (B \vee C) \equiv (A \vee B) \vee C$
Distributivity	$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
	$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$
Absorption	$A \wedge (A \vee B) \equiv A$
	$A \vee (A \wedge B) \equiv A$
	$A \wedge (\neg A \vee B) \equiv A \wedge B$
	$A \vee (\neg A \wedge B) \equiv A \vee B$

  See Kelly page 12 for further equivalences

- proofs via truth tables (or other equivalences)

12.4.2 The Many Faces of Implication

- if P then Q
- P implies Q
- $P \rightarrow Q$
- P only if Q
- Q if P
- Q is a necessary condition for P
- P is sufficient for Q
- $\neg Q \rightarrow \neg P$ (called the *contrapositive* of $P \rightarrow Q$)
- $\neg P \vee Q$

Exercise: Verify the equivalence of $P \rightarrow Q$ and $\neg Q \rightarrow \neg P$.

12.4.3 If and Only If

A "**if and only if**" B means A if B and A only if B, that is, $(B \rightarrow A) \wedge (A \rightarrow B)$

A if and only if B is commonly abbreviated A **iff** B and written $A \leftrightarrow B$

Truth table for $\leftrightarrow$:

A	B	$A \rightarrow B$	$B \rightarrow A$	$A \leftrightarrow B$
T	T	T	T	T
T	F	F	T	F
F	T	T	F	F
F	F	T	T	T

$A \leftrightarrow B$ is true exactly when A and B have the same truth value

$\boxed{Q}$

12.5 How Many Connectives?

- (Kelly Section 1.4)
- started with: $\neg\ \wedge\ \vee\ \rightarrow\ \leftrightarrow$
- we can express everything in terms of $\neg\ \vee$, so $\{\neg, \vee\}$ is called a *complete* set of connectives.
- $\{\neg, \wedge\}$ is also a complete set of connectives.
  exercise: prove this.
- Why do we care about how many connectives we need?

$\boxed{Q}$ 12.5.1 NAND

- $\{|\}$ also complete where $|$ (nand) has truth table

A	B	$A\|B$
T	T	F
T	F	T
F	T	T
F	F	T

- $\neg A$ can be expressed as $A|A$
- $A \wedge B$ can be expressed as $(A|B)|(A|B)$
- How can we express $A \vee B$?
  (Hint: By De Morgan's laws $A \vee B = \ldots$)
- Exercise: Is $\{\neg, \rightarrow\}$ a complete set of connectives?
- Exercise: Give the truth table for the only other complete connective.

12.6 Summary

- logic is based on the assumption that reasoning is based on argument FORM
- a valid argument gives a true conclusion if its premises are (all) true
- truth tables define truth of compound propositions in terms of components
- A is a *tautology* if all rows of the truth table for A are TRUE
- A and B are *equivalent* if all rows of the truth table for A and B are the same

13 Arguments

- history of logic and logic programming
- use of propositional logic to formalize arguments
- notions of validity, satisfiability, consistency
- use of these concepts to establish (logical) correctness of arguments

13.1 History

- Goal: to systematize reasoning
- Propositional logic: George Boole, 1847
  - Good for basic understanding of reasoning, but
  - No ability to reason about properties of individuals

13.2 Terminology

- An **assignment** gives a truth value to each propositional variable in a formula

- A **model** of a formula is any assignment for which the formula is true

- A proposition is **satisfiable** if it is true under some assignment, *i.e.* if it has a model

  e.g. $P \rightarrow \neg Q$ is satisfiable because . . .

- A proposition is **valid** if it is true under all possible assignments (*i.e.*, all assignments are models; it is a *tautology*)

  e.g. $P \rightarrow \neg Q$ is not valid because . . .

$\boxed{Q}$ Terminology (2)

- A set of propositions is **consistent** if they can be true simultaneously; *i.e.* the conjunction of the propositions is satisfiable

e.g. $\{P \rightarrow \neg Q,\ P \rightarrow Q\}$ is consistent because ...

e.g. $\{P \rightarrow \neg Q,\ P \rightarrow Q,\ P\}$ is inconsistent because ...

- A proposition A is a **logical consequence** of a set S of propositions if whenever all propositions in S are true, A is true.
- alternative terminology: S **semantically entails** A
- Denoted: $S \models A$
- *e.g.*, $P \wedge Q \models P$

$\boxed{Q}$ Terminology (3)

- An argument is **valid** if its conclusion is semantically entailed by its premises.

  e.g.

$P \rightarrow Q$	$P \rightarrow Q$
P	Q
$\therefore Q$	$\therefore P$
is valid	is invalid

- Thus an argument is valid if $premises \cup \{\neg conclusion\}$ is inconsistent.

$\boxed{Q}$ 13.2.1 Example (Kelly Example 1.4)

- S_1: If the violinist plays the concerto, then crowds will come if the prices are not too high.
  S_2: If the violinist plays the concerto, the prices will not be too high.
  C: If the violinist plays the concerto, crowds will come.

- Is the argument "S_1, S_2, therefore C" valid?
  i.e. $S_1, S_2 \models C$?
  i.e. is $\{S_1, S_2, \neg C\}$ inconsistent?
  i.e. is $S_1 \wedge S_2 \wedge \neg C$ unsatisfiable?

- S_1 is $P \rightarrow (\neg H \rightarrow A)$
  S_2 is $P \rightarrow \neg H$
  C is $P \rightarrow A$
  so you can check the truth table of

$$(P \rightarrow (\neg H \rightarrow A)) \wedge (P \rightarrow \neg H) \wedge \neg(P \rightarrow A)$$

$\boxed{Q}$ Example (Kelly Example 1.4) (2)

P	A	H	$\neg H \to A$	$P \to (\neg H \to A)$	$P \to \neg H$	$P \to A$	$\neg(P \to A)$	mess
T	T	T	T	T	F	T	F	F
T	T	F	T	T	T	T	F	F
T	F	T	T	T	F	F	T	F
T	F	F	F	F	T	F	T	F
F	T	T	T	T	T	T	F	F
F	T	F	T	T	T	T	F	F
F	F	T	T	T	T	T	F	F
F	F	F	F	T	T	T	F	F

Note that the mess:
$(P \to (\neg H \to A)) \wedge (P \to \neg H) \wedge \neg(P \to A)$ always evaluates to FALSE. Therefore the argument is valid.

13.2.2 Another Example

H_1: If she studies the sciences then she prepares to earn a good living.

H_2: If she studies the humanities, then she prepares for a good life.

H_3: If she prepares for a good living or for a good life then the years are well spent.

H_4: The years were not well spent.

C: She didn't study science or humanities.

Is the argument "H_1, H_2, H_3, H_4, therefore C" valid?

H_1 is $S \rightarrow G$, H_2 is $H \rightarrow L$, H_3 is $G \vee L \rightarrow W$, H_4 is $\neg W$, C is $\neg(S \vee H)$

Check the $2^5 = 32$ rows of the truth table

$$(S \rightarrow G) \wedge (H \rightarrow L) \wedge (G \vee L \rightarrow W) \wedge (\neg W) \wedge (S \vee H)$$

13.3 A Better Way

- Often we can check validity more easily than filling out a truth table
- What are necessary conditions for

$$(S \rightarrow G) \wedge (H \rightarrow L) \wedge (G \vee L \rightarrow W) \wedge (\neg W) \wedge (S \vee H)$$

  to evaluate to true (and our argument to be invalid)?

- Each conjunct must be true. In particular, $\neg W$ must be true (pick simplest propositions first)
- So W must be false
- But $G \vee L \rightarrow W$ must be true, so $G \vee L$ must be false
- So G and L must both be false
- But $S \rightarrow G$ and $H \rightarrow L$ must both be true, so S and H need to be false

A Better Way (2)

- But $S \vee H$ must be true: contradiction!
- So the original argument was valid

Simple technique can help with the bookkeeping: write out the formula and write T or F under each variable and conective as you determine them; propagate variable values

$$(S \rightarrow G) \wedge (H \rightarrow L) \wedge (G \vee L \rightarrow W) \wedge (\ \neg\ W) \wedge (S \ \vee\ H)$$

F T F T F T F T F F T F T T F T F **X** F

$\boxed{Q}$ Exercise

Determine whether or not the following formula is unsatisfiable. If not, find values for the variables that make it satisfiable.

$$(p \rightarrow (\neg h \rightarrow c)) \land (p \rightarrow \neg h) \land \neg(p \rightarrow c)$$

$\boxed{Q}$ 13.4 Example

WHY WORRY?? There are only two things to worry about, either you're healthy or you're sick. If you're healthy there is nothing to worry about and if you're sick, there are two things to worry about . . . either you'll get well or you won't. If you get well there is nothing to worry about, but if you don't, you'll have two things to worry about . . . either you'll go to heaven or to hell. If you go to heaven you have nothing to worry about and if you go to hell you'll be so busy shaking hands with all of us you'll have no time to worry.

CONCLUSION: Either you'll have nothing to worry about or you'll have no time to worry.

H: you are healthy, S you are sick, W you have something to worry about, A you will get well, B you won't get well, C you will go to heaven, D you will go to hell, T you will have time to worry.

$\boxed{Q}$ 13.4.1 Translation to Logic

1.	$H \vee S$	healthy or sick
2.	$H \rightarrow \neg W$	if healthy then no worries
3.	$S \rightarrow (A \vee B)$	if sick then either will get well or won't
4.	$A \rightarrow \neg W$	if get well then no worries
5.	$B \rightarrow (C \vee D)$	if won't get well then will either go to heaven or hell
6.	$C \rightarrow \neg W$	if heaven then no worries
7.	$D \rightarrow \neg T$	if hell the no time to worry
8.	$\therefore \quad \neg W \vee \neg T$	therefore, no need to worry or no time to worry

13.4.2 Checking Validity

Begin by negating conclusion and simplifying:

$\neg(\neg W \vee \neg T)$	negation of conclusion
$\neg\neg W \wedge \neg\neg T$	De Morgan
$W \wedge T$	double negation

So W and T must both be true

Then premise 7 says $D \rightarrow \neg T$, but since T is true, then D must be false (check with a truth table)

Similarly, premise 6 says $C \rightarrow \neg W$, so C must be false, premise 4 says $A \rightarrow \neg W$ so $\neg A$, and premise 2 says $H \rightarrow \neg W$, so $\neg H$

$\boxed{Q}$ Checking Validity (2)

Now we know $W \wedge T \wedge \neg D \wedge \neg C \wedge \neg A \wedge \neg H$

Premise 5 is $B \rightarrow (C \vee D)$, but we know $\neg C \wedge \neg D$, so we conclude $\neg B$

Premise 3 is $S \rightarrow (A \vee B)$, but now we know $\neg A \wedge \neg B$, so $\neg S$

Premise 1 is $H \vee S$, but we now know $\neg H$ and $\neg S$ — contradiction!

Conclusion: either some of our premises were false or our conclusion was true

Argument was valid

13.4.3 Caution!

Why have A mean will get well and B mean won't get well?

Why not use $\neg A$ for won't get well?

Answer: "will get well" and "won't get well" are not the only two possibilities — could also not be sick

Hint: "if you're sick, . . . either you'll get well or you won't."

Translation would be $S \rightarrow (A \vee \neg A)$ — tautology

Tautologies say nothing new; useless as premise

Could have used $\neg H$ for sick, because sick and healthy are the only choices

13.5 Argument Validity in Prolog

Validity can easily be tested in Prolog

```
valid(Premise, Conclusion) :-       % Argument is valid if
        valid(Premise->Conclusion). % premise implies conclusion

valid(Prop) :-                      % Prop is valid if
        unsatisfiable(\+Prop).      % negation is unsatisfiable

unsatisfiable(Prop) :-              % Prop is unsatisfiable
        \+ true(Prop).              % if it can’t be made true
```

Argument Validity in Prolog (2)

```
true(v(true)).
true(\+ Prop) :-
        false(Prop).
true((X;Y)) :-
        (   true(X)
        ;   true(Y)
        ).
true((X,Y)) :-
        true(X),
        true(Y).
true((X->Y)) :-
        (   false(X)
        ;   true(Y)
        ).
true((X<->Y)) :-
        (   true(X),
            true(Y)
        ;   false(X),
            false(Y)
        ).
```

```
false(v(false)).
false(\+ Prop) :-
        true(Prop).
false((X;Y)) :-
        false(X),
        false(Y).
false((X,Y)) :-
        (   false(X)
        ;   false(Y)
        ).
false((X->Y)) :-
        true(X),
        false(Y).
false((X<->Y)) :-
        (   true(X),
            false(Y)
        ;   false(X),
            true(Y)
        ).
```

Argument Validity in Prolog (3)

Represent a propositional variable as v(t), where t is either `true` or `false`.

```
?- valid(((v(P)->v(Q)),(v(Q)->v(R))), v(P)->v(R)).

P = _G157
Q = _G159
R = _G166

Yes
?- valid((v(P)->v(Q))->v(R), v(P)->v(R)).

No
```

Why is last answer `No`?

Argument Validity in Prolog (4)

Can use **false**/1 predicate directly to see why

```
?- false(((v(P)->v(Q))->v(R))-> (v(P)->v(R))).

P = true
Q = false
R = false ;

No
```

(T → F) → F is true, yet T → F is false

Note: extra parentheses are used in the Prolog goals since the builtin precedence and associativity of **->** differs from →.

13.6 Summary: Argument Correctness

- general strategy for determining if an argument of the form

  $$S_1, S_2, \cdots, S_n \models C$$

  is valid

- investigate the unsatisfiability of

  $$S_1 \wedge S_2 \wedge \cdots \wedge S_n \wedge \neg C$$

  using a truth table, or trying for a contradiction when $\neg C$ is true.
  If this formula is unsatisfiable, the argument is valid.
  If this formula is satisfiable, the argument is invalid.

Summary: Argument Correctness (2)

Non truth-table strategies for determining validity

- Motivation: the truth table for a proposition with n propositional variables has 2^n rows – exponential growth. (Recall the last example)

- Seek a strategy which at least for some situations does better (though *worst case performance still exponential*)
  - seek to exploit "relevant" parts of the formula you are working with, e.g. the previous strategy for the college example.

14 Axiom Systems for Propositional Logic

- notions of axiomatic system, deduction
- particular axiom system for propositional logic
- use of this system to establish tautologies, equivalences, consistency, argument validity

Basic Idea (Kelly Ch 4)

- so far . . .
  - *truth tables* used to define
    - tautologies, (in)consistency, validity, . . .
    - **model theory** approach
  - gives the *semantics* of propositional logic
    - semantics = considerations involving “truth” (meaning)
- now . . .
  - a **proof theory** approach
    - axioms, inference rules, proofs, theorems, . . .
- later – *connections* between the semantic and proof theory approaches

14.1 Axiomatic Systems (Kelly 4.2)

- **Proof**:
  - set of initial hypotheses
  - each step generates a new consequence of the hypotheses
  - until the desired proposition is reached.
- In Propositional Logic
  - start with **axioms** (simple tautological propositions)
  - each step uses a **rule of inference**

  (Axioms ≈ Prolog program; conclusion ≈ query.)
- Infinitely many possible axiom systems; we use **AL**

$\boxed{Q}$ 14.1.1 **Syntactic Description**

Axiomatic system has 4 parts:

Σ **alphabet** of symbols; we use $\neg, \rightarrow, (,), P, Q, R, \ldots$

WF the **well formed formulae** (**wff**)

Ax the set of axioms, a subset of WF

R the set of rules of deduction

Well Formed Formulae (wff):

- Any propositional letter (*e.g.*, P, Q, R) is a wff
- If W and V are wff then $W \rightarrow V$ and $\neg W$ are wff
- Nothing else is a wff

Usually want to assume infinite number of propositional letters; assume we can subscript them

$\boxed{Q}$ 14.1.2 **Axioms**

There are infinitely many axioms, but all have one of these forms:

Axiom schemas — allow any wff for A, B, and C

Ax1 $A \to (B \to A)$

Ax2 $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$

Ax3 $(\neg A \to \neg B) \to (B \to A)$

Examples:

- $P \to (Q \to P)$ is an *instance* of Ax1.
- $(\neg(D \to E) \to \neg S) \to (S \to (D \to E))$ is an instance of Ax3.

$\boxed{Q}$ 14.1.3 Inference Rule

One rule of deduction, or **inference rule**:

modus ponens from A and $A \rightarrow B$ infer B

Given a set of hypotheses H, where $H \subseteq WF$, a **deduction** is a sequence of wff $F_1, F_2, \ldots, F_n$ where each F_i is either:

- An axiom ($F_i \in Ax$); or
- A hypothesis ($F_i \in H$); or
- Follows from *earlier* steps by a (the) rule of inference

Inference Rule (2)

We say a deduction ending in F_n is a deduction of F_n from H or that F_n is a deductive consequence of H, and write

$$H \vdash F_n$$

When $H = \emptyset$, we write

$$\vdash F_n$$

and say F_n is a *theorem*

$\boxed{Q}$ 14.1.4 **Trivial Examples**

- A boring example with a very short proof

$$\vdash P \rightarrow (Q \rightarrow P) \qquad \text{Ax1}$$

- Another boring example

$$\vdash (\neg(D \rightarrow E) \rightarrow \neg S) \rightarrow (S \rightarrow (D \rightarrow E)) \qquad \text{Ax3}$$

Q

14.1.5 Short Proof (Kelly Theorem 4.1)

$$\vdash A \to A$$

1.	$A \to ((B \to A) \to A)$	Ax1
2.	$(A \to ((B \to A) \to A)) \to ((A \to (B \to A)) \to (A \to A))$	Ax2
3.	$(A \to (B \to A)) \to (A \to A)$	MP 1, 2
4.	$A \to (B \to A)$	Ax1
5.	$A \to A$	MP 3, 4

Exercise: Prove the following Simplification MetaTheorems:

$$\vdash (A \wedge B) \to A$$

$$\vdash (A \wedge B) \to B$$

Here $A \wedge B$ is *defined* as $\neg(A \to \neg B)$.

14.2 Meta-theorems

We have proved $\vdash A \to A$, but what about $\vdash P \to P$?

We haven't proved it, but we could: duplicate proof, replacing A with P

Works because for every axiom with A, there's one with P

Modus ponens works for P as well as for A

Take our theorem as a *meta*-theorem — a theorem template

As a shortcut, we could use $A \to A$ as if it were an axiom schema, because we could always replace it by its proof

Like using subroutines when programming

14.2.1 Using $A \to A$

Prove $B \to (A \to A)$

1.	$A \to A$	Theorem 4.1
2.	$(A \to A) \to (B \to (A \to A))$	Ax1
3.	$B \to (A \to A)$	MP 1, 2

Could replace Theorem 4.1 line with proof of Theorem 4.1:

1.	$A \to ((B \to A) \to A)$	Ax1
2.	$(A \to ((B \to A) \to A)) \to ((A \to (B \to A)) \to (A \to A))$	Ax2
3.	$(A \to (B \to A)) \to (A \to A)$	MP 1, 2
4.	$A \to (B \to A)$	Ax1
5.	$A \to A$	MP 3, 4
6.	$(A \to A) \to (B \to (A \to A))$	Ax1
7.	$B \to (A \to A)$	MP 5, 6

$\boxed{Q}$ Exercise: Transitive Implication

Fill in the blanks in the following proof of

$$\{A \rightarrow B, B \rightarrow C\} \vdash A \rightarrow C$$

1. $B \rightarrow C$ __________
2. $(B \rightarrow C) \rightarrow (A \rightarrow (B \rightarrow C))$ __________
3. ________________________ MP 1, 2
4. $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$ __________
5. $(A \rightarrow B) \rightarrow (A \rightarrow C)$ __________
6. $A \rightarrow B$ Hypothesis
7. ________________________ MP 5, 6

This proves implication is transitive; call it TI

$\boxed{Q}$

14.2.2 **Using TI**

Can use meta-theorems with hypotheses as rules of inference

First prove instances of hypotheses, then use meta-theorem to conclude instance of conclusion

Prove $\vdash \neg A \to (A \to B)$

1.	$\neg A \to (\neg B \to \neg A)$	Ax1
2.	$(\neg B \to \neg A) \to (A \to B)$	Ax3
3.	$\neg A \to (A \to B)$	TI 1, 2

Could turn into proper proof by replacing step 3 with proof of TI, with $\neg A$ substituted for A, $(\neg B \to \neg A)$ for B and $(A \to B)$ for C, and lines 1 and 2 for the hypothesis lines

But this proof is much easier!

$\boxed{Q}$ 14.2.3 More Meta-theorems

Kelly proves other useful Meta-theorems

Deduction meta-theorem:

$$\text{If} \quad H \cup \{A\} \vdash B \quad \text{then} \quad H \vdash A \rightarrow B$$

Proof is more complex (by induction)

$\boxed{Q}$ More Meta-theorems (2)

Use deduction meta-theorem to prove $\{A \rightarrow (B \rightarrow C)\} \vdash B \rightarrow (A \rightarrow C)$:

1.	$A \rightarrow (B \rightarrow C)$	Hypothesis
2.	A	Hypothesis (for sake of proof
3.	$B \rightarrow C$	MP 1, 2
4.	B	Hypothesis (for sake of proof
5.	C	MP 3, 4
So,	$\{A \rightarrow (B \rightarrow C), A, B\} \vdash C$	
then	$\{A \rightarrow (B \rightarrow C), B\} \vdash A \rightarrow C$	Deduction theorem
then	$\{A \rightarrow (B \rightarrow C)\} \vdash B \rightarrow (A \rightarrow C)$	Deduction theorem

14.3 Finding axiomatic proofs

- no firm search strategy
- proofs generally hard to find
  - usually a mixture of seeing what you can do with the hypotheses you have, and seeing what might lead to the desired conclusion.

$\boxed{Q}$ 14.4 Truth and Deduction (Kelly 4.5)

What is the relationship between *truth* ($\models$, see Kelly chapter 1) and *deduction* ($\vdash$, see Kelly chapter 4)?

What makes an axiom system useful?

1. **Consistency**: if $\vdash A$ then $\not\vdash \neg A$
   Is it impossible to prove a wff and its negation?

2. **Soundness**: $\vdash A$ implies $\models A$
   Is it impossible to prove a wff that is not a tautology?

3. **Completeness**: $\models A$ implies $\vdash A$
   Can everything that is a tautology be proved?

4. **Decidability**:
   Is there an algorithm to always decide if $\vdash A$?

$\boxed{Q}$ Truth and Deduction (Kelly 4.5) (2)

The *AL* system of Kelly chapter 4 has all these properties

1. *Consistency*: follows from *soundness* (below), and that $\models$ is consistent

2. *Soundness*: proof by induction; axioms are tautologies, and modus ponens only yields tautologies from tautologies

3. *Completeness*: complex proof, see Kelly pp. 87–90

4. *Decidability*: *AL* is complete, so it is enough to decide if $\models A$, which can be decided by truth table

$\boxed{Q}$ 14.5 **Summary**

- proof system based on axioms and inference rules defines a computational procedure for verifying deductions . . . though not necessarily practical for *generating* deductions

- correctness of proof system with respect to semantics is guaranteed by soundness, completeness

15 Resolution

- Kelly chapter 5
- an alternative computational mechanism for establishing theorems—actually a way of establishing *unsatisfiability*
- better suited to computer implementation than axiomatic methods—the basis of the Prolog system!
- generalises usefully to predicate logic (see later)
- to apply the method, wff must be in a special form, *clausal form*

$\boxed{Q}$ Terminology

- A **literal** is a basic proposition or the negation of a basic proposition.

- **conjunctive normal form (CNF)**

  e.g. $(A \vee \neg B) \wedge (B \vee C) \wedge A$

  conjunction of disjunctions of literals

- **disjunctive normal form (DNF)**

  e.g. $(\neg P \wedge \neg Q) \vee (\neg Q \wedge R) \vee P$

  disjunction of conjunctions of literals

- **Theorem:** Every Propositional Logic formula can be expressed in CNF and in DNF.

$\boxed{Q}$

15.1 Converting to CNF or DNF

1. Eliminate $\leftrightarrow$ using $A \leftrightarrow B$ equivalent to $(A \rightarrow B) \wedge (B \rightarrow A)$.

2. Eliminate $\rightarrow$ using $A \rightarrow B$ equivalent to $\neg A \vee B$.

3. Use De Morgan's Laws to push $\neg$ inward to immediately before propositional variables.

4. Eliminate double negations such as $\neg\neg A$ equivalent to A.

5. Use distributive laws to get the required form.

$\boxed{Q}$ 15.1.1 **Example (Kelly Example 5.3)**

Convert $(\neg p \land (\neg q \to r)) \leftrightarrow s$ to conjunctive normal form.

1. $((\neg p \land (\neg q \to r)) \to s) \land (s \to (\neg p \land (\neg q \to r)))$

2a. $(\neg(\neg p \land (\neg q \to r)) \lor s) \land (\neg s \lor (\neg p \land (\neg q \to r)))$

2b. $(\neg(\neg p \land (q \lor r)) \lor s) \land (\neg s \lor (\neg p \land (q \lor r)))$

3a. $((p \lor \neg(q \lor r)) \lor s) \land (\neg s \lor (\neg p \land (q \lor r)))$

3b. $((p \lor (\neg q \land \neg r)) \lor s) \land (\neg s \lor (\neg p \land (q \lor r)))$

4. not needed

5a. $(((p \lor \neg q) \land (p \lor \neg r)) \lor s) \land ((\neg s \lor \neg p) \land \neg s \lor (q \lor r)))$

5b. $(p \lor \neg q \lor s) \land (p \lor \neg r \lor s) \land (\neg s \lor \neg p) \land (\neg s \lor q \lor r)$

Note that Kelly gives a different derivation of this CNF.
See also Kelly Example 5.2.

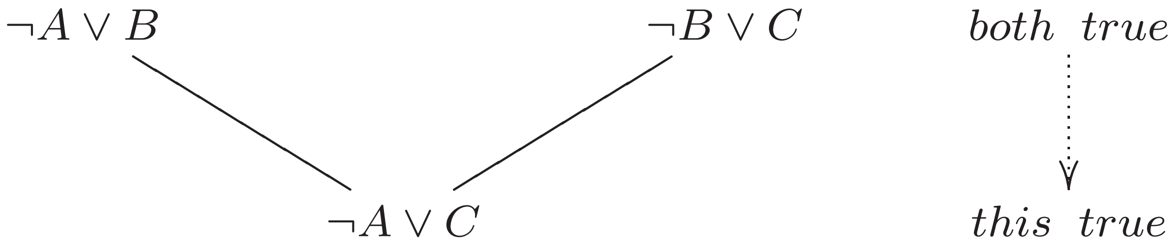
$\boxed{Q}$ Exercise

Convert the following proposition into CNF:

$$P \leftrightarrow (Q \vee R)$$

15.2 The Key to Resolution (Kelly page 99)

- consider the formulae $(\neg A \vee B)$ and $(\neg B \vee C)$
  - If B is true then the truth depends only on C.
  - If B is false then the truth depends only on A.
- only one of B, $\neg B$ is true, so
  if $(\neg A \vee B)$ and $(\neg B \vee C)$ are both true, then either $\neg A$ is true or C is true, i.e. $(\neg A \vee C)$ is true: this is the **resolvent** of the original two clauses.



15.2.1 Resolution Principle.

(Kelly Theorem 5.1, page 102)

If D is a resolvent of clauses C_1 and C_2, then $C_1 \wedge C_2 \models D$

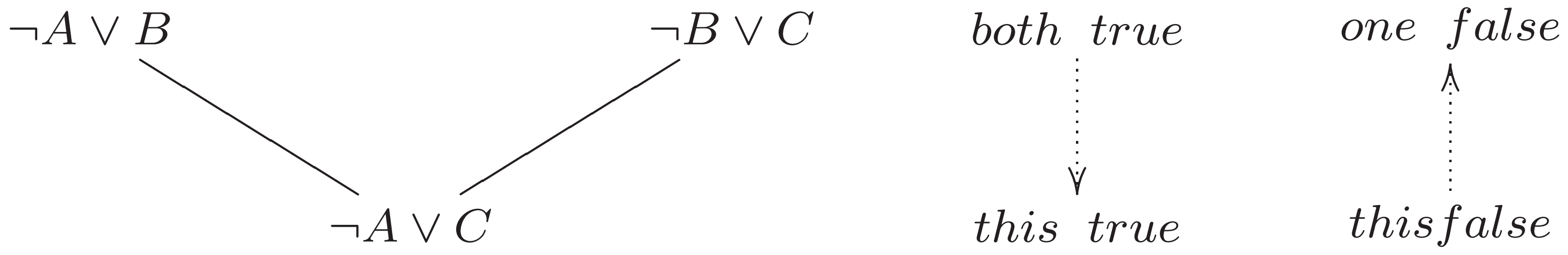
- Proof – by formalising the above key idea.

Another view: $\neg A \vee B$ means $A \rightarrow B$

Remember: $A \rightarrow B, B \rightarrow C \models A \rightarrow C$

So if $\neg A \vee B$ and $\neg B \vee C$, then $\neg A \vee C$ must hold

15.3 The Key to Refutations



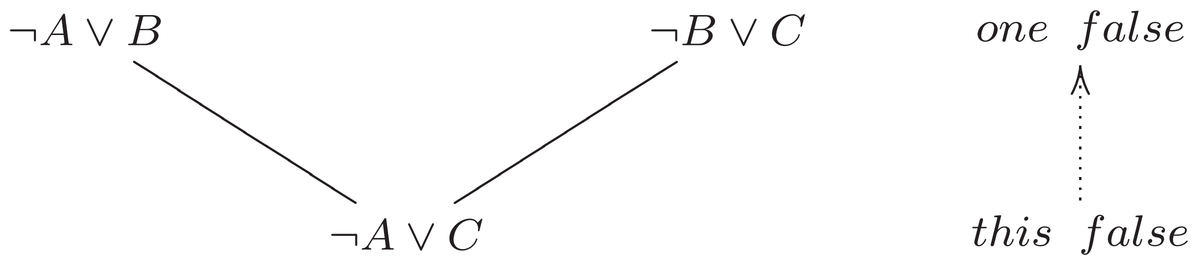
$\neg A \vee B$ $\quad$ $\neg B \vee C$ $\quad$ $both\ \ true$ $\quad$ $one\ \ false$

$\neg A \vee C$ $\quad$ $this\ \ true$ $\quad$ $thisfalse$

- if $(\neg A \vee C)$ is false, then one of $(\neg A \vee B)$ or $(\neg B \vee C)$ must be false.

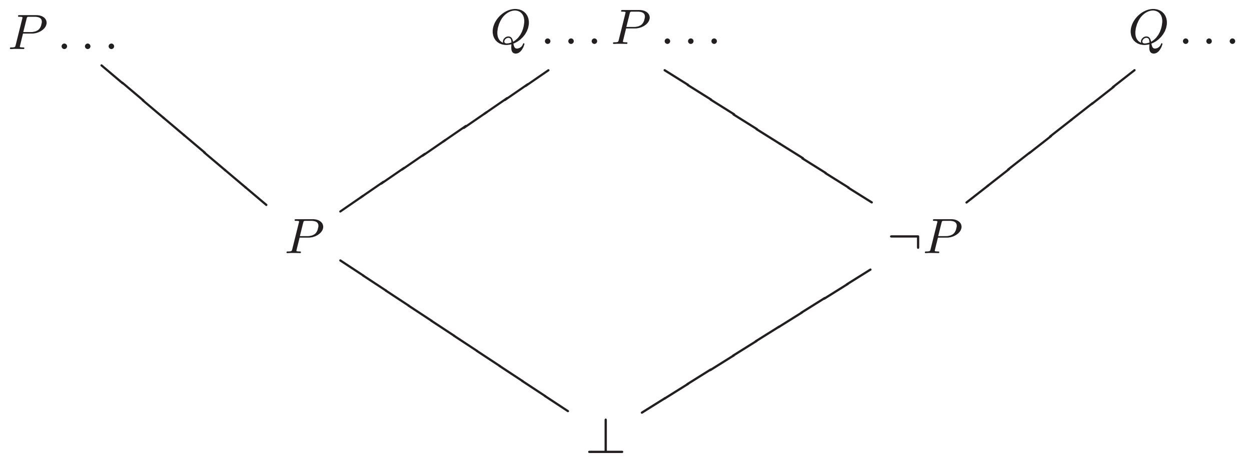
(i.e. using the *contrapositive* of the observation before).

15.3.1 Refutations



- repeatedly "cancel out" B with $\neg B$ and seek $\perp$ (empty clause).

- if you reach $\perp$, argue "backwards" to conclude the starting formulae were inconsistent.



Q

15.3.2 Deductions and Refutations (Kelly page 105)

- A **resolution deduction** of clause C from a set S of clauses is a sequence $C_1, C_2, \ldots, C_n$ of clauses such that $C_n = C$ and for each i, $1 \leq i \leq n$,
  - C_i is a member of S, or
  - C_i is obtained, by resolution from C_j and C_k, $j, k \leq i$.
- A **resolution refutation** of a set S of clauses is a resolution deduction of $\perp$ from S.

$\boxed{Q}$ 15.3.3 Resolution Theorem

Given a clause set S, $R(S)$ is the set of all clauses derivable from S by one resolution step

$R^*(S)$ is the set of *all* clauses obtainable after a finite number of resolution steps, starting with S.

Resolution theorem: S is unsatisfiable iff $\perp \in R^*(S)$

Proof. By induction, using the Resolution Principle.

15.4 Establishing Validity With Resolution

- Algorithm for establishing formula A is *valid*
  - put $\neg A$ in conjunctive normal form
  - take set of conjuncts and apply resolution repeatedly
  - if eventually you get $\perp$ then $\neg A$ unsatisfiable (so A is valid); otherwise A is not valid
- How can you be sure that this procedure *terminates*?

Q 15.5 Set Representation of Clauses

- **clause**: a finite set of literals representing a disjunction

e.g. $\{A, \neg B\}$ represents $A \vee \neg B$

- **clause set**: a set of clauses representing a CNF formula

e.g. $\{\{A, \neg B\}, \{B, C\}, \{A\}\}$ for $(A \vee \neg B) \wedge (B \vee C) \wedge A$

- **empty clause**: $\{\}, \perp$ represent a formula always false
- literals L and $\neg L$ are called **complementary**
- **resolvent** of two clauses containing complementary literals is their union omitting L and $\neg L$

$\boxed{Q}$ Set Representation Examples

- e.g. $C_1 = \{A, B\},\ C_2 = \{\neg A, E\}:$ resolvent $\{B, E\}$
- e.g. $C_3 = \{A, B, \neg C\},\ C_4 = \{\neg A, C\}$
  - one resolvent of C_3 and C_4 is $\{B, \neg C, C\}$
  - another resolvent of C_3 and C_4 is $\{A, B, \neg A\}$
  - but $\{B\}$ is not a resolvent of C_3 and C_4
  - both clauses $\{B, \neg C, C\}$ and $\{A, B, \neg A\}$ are *tautological* (always true) and can be ommitted.

$\boxed{Q}$ 15.5.1 **Example**

Remember the disappointed student?

- H_1: If she studies the sciences then she prepares to earn a good living.
  H_2: If she studies the humanities, then she prepares for a good life.
  H_3: If she prepares for a good living or for a good life then the years are well spent.
  H_4: The years were not well spent.
  C: She didn't study science or humanities.
- Is the argument "H_1,H_2,H_3,H_4, therefore C" valid?

$\boxed{Q}$ 15.6 Example Proof

H_1	is	$S \rightarrow G$
H_2	is	$H \rightarrow L$
H_3	is	$G \vee L \rightarrow W$
H_4	is	$\neg W$
C	is	$\neg(S \vee H)$

We convert

$$(S \rightarrow G) \wedge (H \rightarrow L) \wedge (G \vee L \rightarrow W) \wedge (\neg W) \wedge (S \vee H)$$

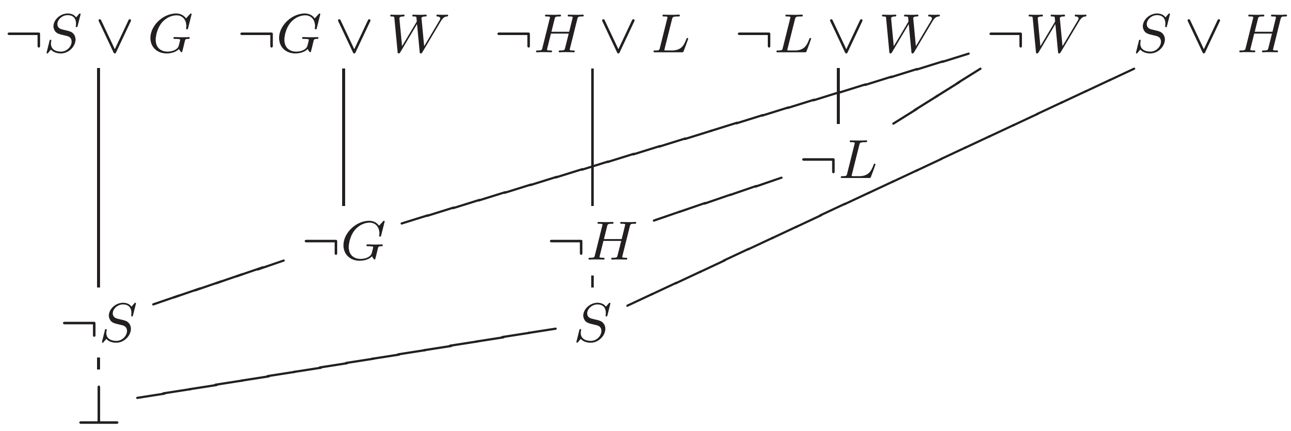
to CNF.

2a. $(\neg S \vee G) \wedge (\neg H \vee L) \wedge (\neg(G \vee L) \vee W) \wedge \neg W \wedge (S \vee H)$

3. $(\neg S \vee G) \wedge (\neg H \vee L) \wedge ((\neg G \wedge \neg L) \vee W) \wedge \neg W \wedge (S \vee H)$

5. $(\neg S \vee G) \wedge (\neg H \vee L) \wedge (\neg G \vee W) \wedge (\neg L \vee W) \wedge \neg W \wedge (S \vee H)$

15.6.1 Example Proof Diagram



1	$\neg L \vee W$	
2	$\neg W$	
3	$\neg L$	1,2
4	$\neg G \vee W$	
5	$\neg G$	2,4
6	$\neg S \vee G$	
7	$\neg S$	5,6

a	$\neg L \vee W$	
b	$\neg W$	
c	$\neg L$	a,b
d	$\neg H \vee L$	
e	$\neg H$	c,d

$\boxed{Q}$ Exercise

Use resolution to show

$$\{\{A, B, \neg C\}, \{\neg A\}, \{A, B, C\}, \{A, \neg B\}\}$$

is unsatisfiable. Show a resolution diagram.

15.6.2 Important Results

- **Theorem.** (Soundness of resolution)
  If there is a resolution refutation of S, then S is unsatisfiable.
  Proof: straightforward induction.

- **Theorem.** (Completeness of resolution)
  If S is unsatisfiable, there is a resolution refutation of S.
  Proof: delicate induction, using the compactness theorem to reduce the problem to a finite one.

$\boxed{Q}$ 15.7 Summary: Propositional Logic so Far

- syntax: the shape of “legal” formulae
- semantics: defined using truth tables. A valid iff $\neg A$ unsatisfiable
- proof definitions - axioms, inference rules
- notation

$A \vdash B$	B is deducible from A
$A \models B$	B is semantically entailed by A

- techniques
  - truth tables
  - axioms and inference
  - resolution
- key results: $\vdash A$ iff $\models A$ iff there is a resolution refutation of A

Q

15.7.1 **Discussion**

- *Any problem that involves deciding amongst a finite number of choices can be expressed in propositional satisfiability*.
- resolution implementable easily, but no polynomial time algorithm known (for this or any of the other techniques)
- the satisfiability problem (SAT) for propositional logic is “typical” of a large class of difficult problems – so-called *NP-complete* problems – for which there are no known efficient algorithms.
- however, *by limiting expression to restricted forms of formulae,* resolution can lead to a practical programming language – Prolog

16 Linear Resolution

- present linear resolution – a more efficient form of resolution

- show connections between linear (input) resolution and Prolog

- Introduce negation as failure

Linear Resolution (2)

- Linear resolution is a refinement of resolution (restricts the search space).

- A **linear (resolution) deduction** (Definition 10.1) or proof of C from S is a sequence of pairs $(C_0, B_0), \ldots, (C_n, B_n)$ such that $C = C_{n+1}$ and

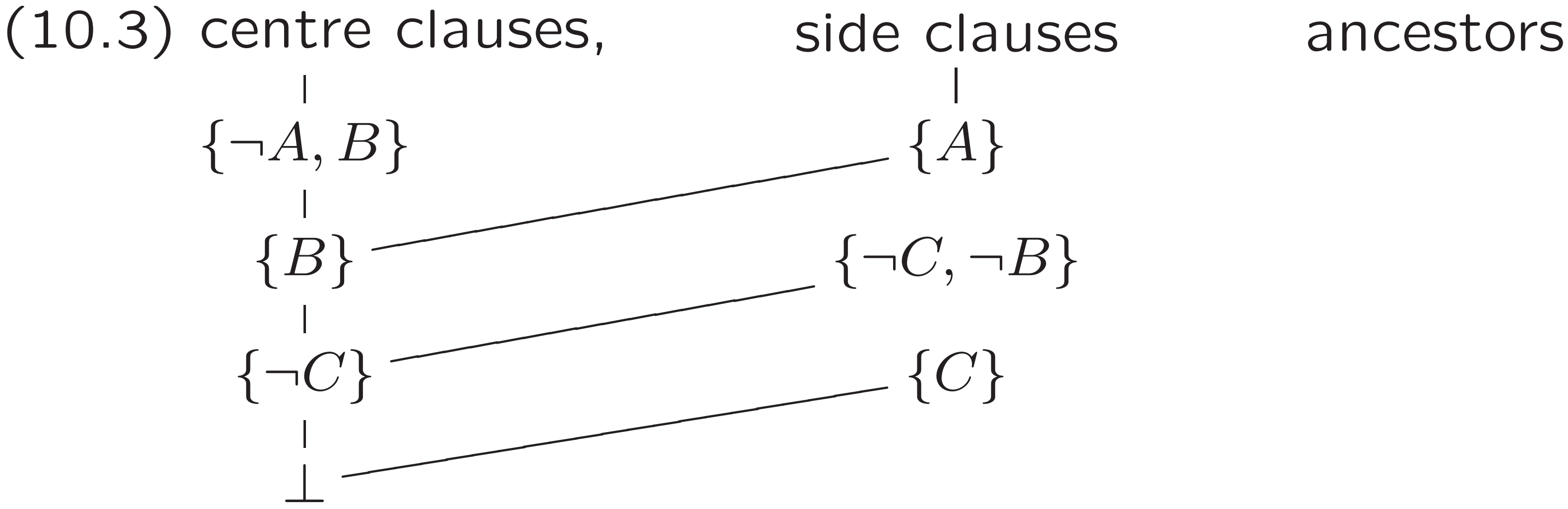
  1. C_0 and each B_i are elements of S or some C_j with $j < i$

  2. each $C_{i+1}, i \leq n$ is a resolvent of C_i and B_i.

- A **linear refutation** (Definition 10.1) is a linear deduction of $\perp$ from S

Q Linear Resolution (3)

Basic idea: when choosing two clause to resolve, always make one be the result of the previous resolution



(10.3) centre clauses, side clauses ancestors

$\{\neg A, B\}$ $\{A\}$

$\{B\}$ $\{\neg C, \neg B\}$

$\{\neg C\}$ $\{C\}$

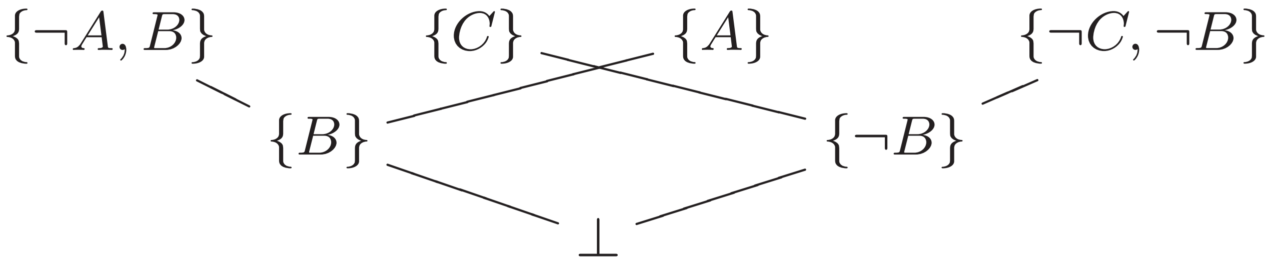
$\bot$

•

Main result: linear resolution is complete — if S is an unsatisfiable set of propositional Horn clauses then there is a linear refutation of S

Linear Resolution Illustration

Resolution in general



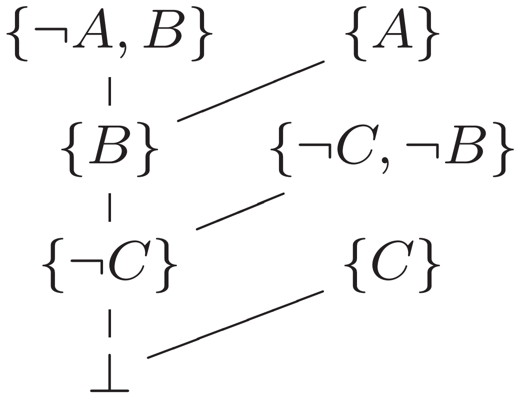
$\{\neg A, B\}$ $\{C\}$ $\{A\}$ $\{\neg C, \neg B\}$

$\{B\}$ $\{\neg B\}$

$\bot$

versus

Linear Resolution



$\{\neg A, B\}$ $\{A\}$

$\{B\}$ $\{\neg C, \neg B\}$

$\{\neg C\}$ $\{C\}$

$\bot$

$\boxed{Q}$ 16.1 Horn clauses and Prolog

- conjunctive normal form = conjunction of clauses

- clause $\{L_1, \ldots, L_n\}$ represents

$$(A_1 \vee A_2 \cdots \vee A_m \vee \neg B_1 \vee \cdots \vee \neg B_k) \ \ [n = m + k]$$

- which, using DeMorgan's laws, can be written

$$B_1 \wedge \cdots \wedge B_k \rightarrow A_1 \vee A_2 \cdots \vee A_m$$

- a clause with at most one positive literal is called a **Horn clause**

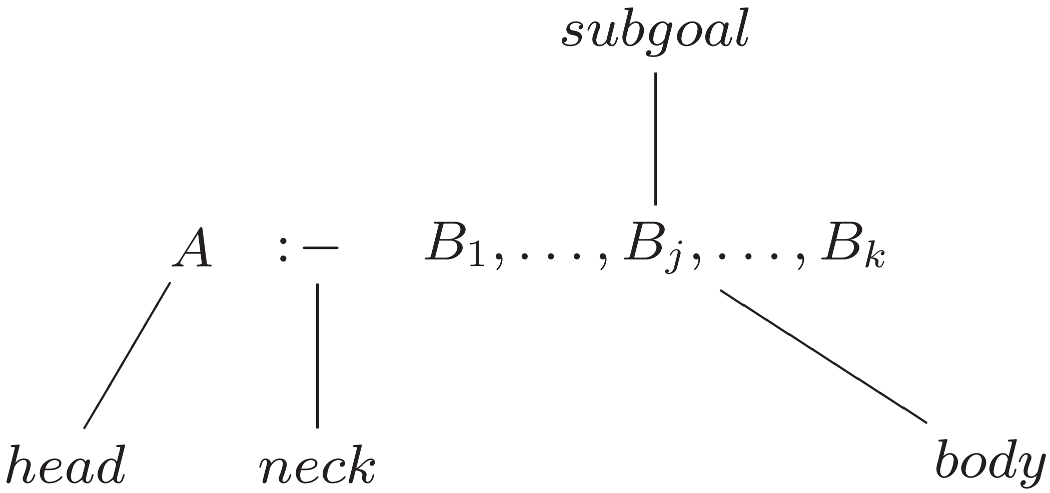
$$B_1 \wedge \cdots \wedge B_k \rightarrow A$$

- written with $:-$ for $\leftarrow$ and , for $\wedge$ in Prolog notation

$$A :- B_1, \ldots, B_k$$

Horn clauses and Prolog (2)

- with one positive and some negative literals – *rule* e.g. $A :- B_1, \ldots, B_k$
- with no negative literals – *fact* e.g. A
- with no positive literals – *goal* e.g. $:- B_1, \ldots, B_k$.



subgoal

$A \quad :- \quad B_1, \ldots, B_j, \ldots, B_k$

head *neck* *body*

$\boxed{Q}$ 16.2 Propositional Logic Programs

- a propositional logic program P is a collection of facts and rules; typically, we want to know if some other fact(s) are a logical consequence of P

Example:

`mg :- mgo, h2.`	i.e.	$\mathtt{h2} \wedge \mathtt{mgo} \longrightarrow \mathtt{mg}$
`h20 :- mgo, h2.`		$\mathtt{h2} \wedge \mathtt{mgo} \longrightarrow \mathtt{h20}$
`co2 :- c, o2.`		$\mathtt{c} \wedge \mathtt{o2} \longrightarrow \mathtt{co2}$
`h2co3 :- co2, h2o.`		$\mathtt{co2} \wedge \mathtt{h2o} \longrightarrow \mathtt{h2co3}$
`mgo.`		$\mathtt{mgo}$
`h2.`		$\mathtt{h2}$
`o2.`		$\mathtt{o2}$
`c.`		$\mathtt{c}$

- given these facts and rules, can you prove `h2co3` is generated?

16.2.1 Clausal Form of Logic Program

A1	$\mathrm{h2} \wedge \mathrm{mgo} \rightarrow \mathrm{mg}$	$\neg\mathrm{h2} \vee \neg\mathrm{mgo} \vee \mathrm{mg}$
A2	$\mathrm{h2} \wedge \mathrm{mgo} \rightarrow \mathrm{h20}$	$\neg\mathrm{h2} \vee \neg\mathrm{mg0} \vee \mathrm{h20}$
A3	$\mathrm{c} \wedge \mathrm{o2} \rightarrow \mathrm{co2}$	?
A4	$\mathrm{co2} \wedge \mathrm{h2o} \rightarrow \mathrm{h2co3}$	?
A5	mgo	mgo
A6	h2	h2
A7	o2	o2
A8	c	c
A9	$\neg\mathrm{h2co3}$	$\neg\mathrm{h2co3}$

$\boxed{Q}$

16.2.2 Propositional Logic Programs and Resolution

- given a propositional logic program P to know if some other fact(s) are a logical consequence of P

- add the conjunction of the facts as the goal clause G

  $$:- h2co3$$

  and, using resolution, show that $P \cup \{G\}$ is unsatisfiable

- remember, in clausal form :− h2co3 is ¬h2co3

- i.e. adding the goal is just adding the negation of the query

16.3 Linear Input Resolution

- (Nerode and Shore Definition 10.8) A **linear input (LI) resolution** of a Goal G from a set of program clauses P is a linear resolution refutation of $S = P \cup \{G\}$ that starts with G and in which all the side clauses are from P (input clauses).

- **Theorem** (Nerode and Shore Theorem 10.10) For *propositional logic programs*, there is a linear refutation which starts with the goal clause and uses only clauses from the program as side clauses;

- *i.e.* linear input resolution is complete for propositional logic programs.

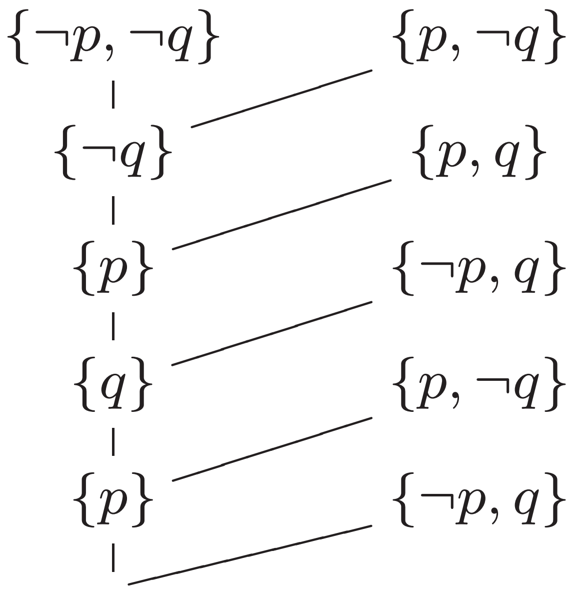
- but linear input resolution is not complete for *arbitrary* clause sets.

16.3.1 Counterexample

- e.g. consider given clauses $\{p, q\}, \{p, \neg q\}, \{\neg p, q\}, \{\neg p, \neg q\}$

- unsatisfiable, *but* not detected by linear input resolution

- one attempt looks like . . .



$\{\neg p, \neg q\}$ $\{p, \neg q\}$

$\{\neg q\}$ $\{p, q\}$

$\{p\}$ $\{\neg p, q\}$

$\{q\}$ $\{p, \neg q\}$

$\{p\}$ $\{\neg p, q\}$

- no attempt will work, because . . .

$\boxed{Q}$ Exercise

Use *linear input* resolution to refute the clause set:

$$\{\{q, \neg p, \neg s\}, \{p, \neg r\}, \{r\}, \{s\}, \{\neg q\}\}$$

Show a resolution diagram.

16.4 Connections with Prolog

- representation . . . a program, with a goal

1.	$\{q, \neg p, \neg s\}$	q :- p, s.
2.	$\{p, \neg r\}$	p :- r.
2.	$\{p, \neg s\}$	p :- t.
4.	$\{s\}$	r :- v.
4.	$\{s\}$	s.
3.	$\{t\}$	t.
5.	$\{\neg q\}$	:- q

- execution – a form of linear input resolution
- the system responds YES, which is short for . . .
- issues of efficiency, completeness . . .

16.4.1 (Abstract) Interpreter for Propositional Prolog

```
Input: A query Q and a logic program P
Output: TRUE if Q ‘implied’ by P, FALSE otherwise

    To Prove(G1,... Gn):
        For each rule from P of the form
                G1 :- B1, ... ,Bn:
            if we can Prove(B1,...Bn,G2,...Gn):
                return TRUE
        return FALSE (no rule was satisfiable)
```

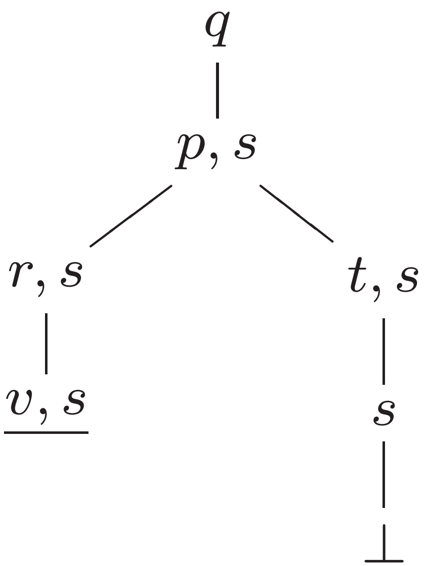
$\boxed{Q}$ 16.4.2 **SLD Resolution**

- Selection rule: always resolve on first literal in clause
- Linear resolution: produce linear input refutations
- Definite clauses: Horn clauses with exactly one positive literal

16.4.3 **SLD Tree**

Prove q with earlier example program:

```
q :- p, s.    p :- r.     p :- t.
r :- v.       s.          t.
```



Branching represents choice points (revisited on backtracking)

$\perp$ indicates *success*; underlined goals indicate *failure*

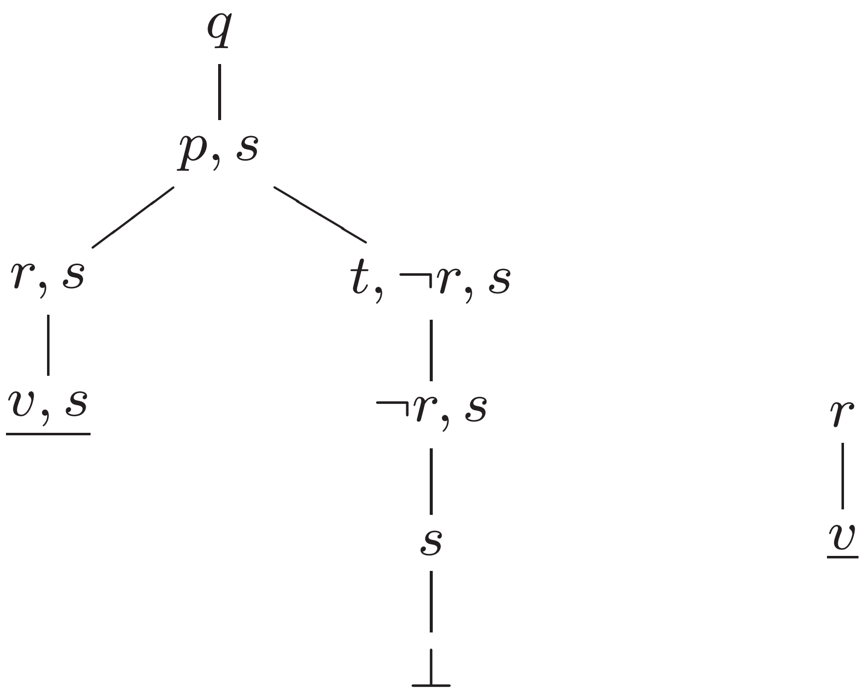
16.4.4 SLDNF Resolution

- Prolog actually uses a variation on SLD resolution called SLDNF
- **SLDNF resolution** is SLD resolution with *negation as failure*
- **negation as failure** means a negated literal is proved by trying to prove the positive literal; if this succeeds, the negation fails, and vice versa
- Negation as failure allows Prolog clauses to have negations in the body
- Never needed for Horn (or definite) clauses
- *E.g.*: `p :- t, \+r.`
- Equivalent to $\{p, \neg t, r\}$: not Horn

Negation as Failure Example

Prove q with updated program:

```
q :- p, s.     p :- r.      p :- t, \+r.
r :- v.        s.           t.
```



$\neg r$ is proved because separate proof of r *fails*

16.5 Summary: Linear Resolution and Prolog

- Nerode and Shore Theorem 10.15 ensures that the resolution method underlying Prolog, SLD-resolution, is complete; however, Prolog's search strategy is not guaranteed to find a refutation even if one exists!

- All of this needs further care when dealing with predicate logic programs – i.e. admitting variables – more later.

- **Note:** The tree structures in SLD-trees seen in tracing program execution are generated by searching the different possibilities for side clauses.

16.6 Summary: Prolog “theory”

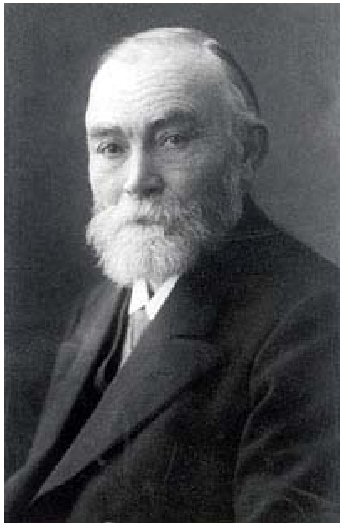
- SLD resolution – a refinement of linear resolution
  - SLD resolution is sound and complete
- Prolog = SLD resolution + search strategy
  - Prolog is sound – answers indicate logical consequence
  - Prolog is not complete – will not always give an answer
- other issues with Prolog (undermining soundness!):
  - occur check, cut, negation, . . .

$\boxed{\forall x}$ 17 Predicate Logic

- Relations and Functions
- From English to Predicate Logic
- Formal Language of Predicate Logic

$\boxed{\forall x}$ History

- Propositional logic can only make absolute statements, not statements about some/all individual(s)
- Can say “if it rains, I will get wet”
- Can’t say “if it rains, anyone who is outside will get wet”
- Gotlob Frege proposed the Predicate Calculus in 1879 to solve this problem
- can reason about properties of individuals
- can make sweeping statements about *all* individuals
- can reason about relationships between individuals



Examples of Predicate Logic Formulae

- every shark eats a tadpole

$$\forall x(S(x) \rightarrow \exists y(T(y) \wedge E(x,y)))$$

- all large white fish are sharks

$$\forall x(W(x) \rightarrow S(x))$$

- colin is a large white fish living in deep water

$$W(colin) \wedge D(colin)$$

- any tadpole eaten by a deep water fish is miserable

$$\forall z((T(z) \wedge \exists y(D(y) \wedge E(y,z))) \rightarrow M(z))$$

- $\therefore$ some tadpole(s) are miserable

$$\therefore \exists z(T(z) \wedge M(z))$$

Is this argument valid?

17.1 Relations and Functions

A **relation** relates objects in the domain

- *cf.* Prolog *predicates*
- *E.g.*, $<$ is a relation over integers (or reals), “is parent of” is a relation on humans
- Can relate any number of objects, *e.g.*, “are the lengths of the sides of a right triangle” relates triples of numbers

17.1.1 Relations

- A relation is a set of tuples of objects

- *E.g.*, $<$ on the integers is the set:

$$\{\langle 0,1\rangle,\langle 0,2\rangle,\ldots,\langle 1,2\rangle,\ldots,\langle -1,0\rangle,\ldots\ldots\}$$

- *E.g.*, Difference is:

$$\{\langle 0,0,0\rangle,\langle 1,0,1\rangle,\ldots\langle 0,1,-1\rangle,\ldots\}$$

- Nothing wrong with infinite sets (just don't try to compute with them!)

- Usually write this as, *e.g.*, $\mathrm{Less}(0,1)$, or $\mathrm{Difference}(3,1,2)$

17.1.2 Functions

- A *function* is a relation where the last element of each tuple is unique for each combination of the others
- $\mathrm{Difference}$ is a function, because given any a and b, there is only one c such that $\langle a, b, c\rangle \in \mathrm{Difference}$
- Less is not a function because, *e.g.*, $\langle 0, 1\rangle \in \mathrm{Less}$ and $\langle 0, 2\rangle \in \mathrm{Less}$
- Functions are usually written as function applications, which show all but the last argument, and stand for the unique value for the last argument given the other arguments
- $\mathrm{difference}(3, 1)$ stands for the unique x such that $\mathrm{difference}(3, 1, x)$ holds
- Function applications can be nested

17.2 From English to Predicate Logic

- Take predicates (i.e. sentences with the subject abstracted away) and use symbols for them.

- e.g. Sentence: “He is a man.” Predicate: is a man
  Symbol for predicate: $M(\ )$
  “x is a man”, $M(x)$, cannot be assigned a truth value.
  Bill is a man, $M(bill)$, can be assigned a truth value.

- Can abstract away sentence *object* as well as *subject*:
  “Bob is taller than Bill” $T(bob, bill)$

- Quantifier examples:

  “Every man is mortal” $\forall x(man(x) \rightarrow mortal(x))$

  “Some cat is mortal” $\exists x(cat(x) \wedge mortal(x))$

- Usually use $\rightarrow$ with $\forall$ and $\wedge$ with $\exists$

$\boxed{\forall x}$ 17.2.1 Examples

Let $L(x, y)$ be “x loves y”; let $I(x, y)$ be “x is y”

$L(james, jean)$	James loves Jean
$\forall x L(x, jean)$	Everyone loves Jean (including Jean!)
$\forall x(\neg I(x, jean) \rightarrow L(x, jean))$	Jean is loved by everyone else
$\exists x(\neg I(x, james) \wedge L(x, james)$	Someone other than James loves James
$\forall x(\exists y L(x, y))$	Everybody loves somebody
$\exists y(\forall x L(x, y))$	Someone is loved by everybody
$\exists x(\forall y L(x, y))$	Someone loves everybody

$\boxed{\forall x}$ Exercise

Translate the following statement to predicate logic:

Everyone barracks for a footy team

Use the following predicates:

$P(x)$	x is a person
$T(x)$	x is a footy team
$B(x, y)$	x barracks for y

17.2.2 Word Order

- follow word order with care
  - “there is something which is not P” :
    $\exists y \neg P(y)$
  - “there is not something which is P” (“nothing is P”):
    $\neg \exists y P(y)$
  - “all S are not P” *vs.* “not all S are P:”
    $\forall x(S(x) \to \neg P(x))$ or $\neg \forall x(S(x) \to P(x))$?
    * consider: “all that glitters is not gold”
    * consider: “all 255 students are not asleep”

17.2.3 **Quantification Order**

- Order of different quantifiers is important

- $\forall x \exists y$ is not the same as $\exists y \forall x$

- First says each x has a y that satisfies $P(x, y)$, second says there's an individual y that satisfies $P(x, y)$ for every x

- But $\forall x \forall y$ is the same as $\forall y \forall x$ and $\exists x \exists y$ is the same as $\exists y \exists x$

$\boxed{\forall x}$ Quantification Order (2)

- May help to think of a game where I make a claim and you try to disprove it

- If I claim $\forall x \exists y P(x, y)$, then you can challenge me by choosing an x and asking me to find the y that satisfies $P(x, y)$, *but I get to know the x you chose*

- If I claim $\exists y \forall x P(x, y)$, then you can challenge me by asking me to find the y, then you just have to find an x that does not satisfy $P(x, y)$, *and you get to know the y I chose*

- If I claim $\exists x \exists y P(x, y)$, then I have to find both x and y, so it doesn't matter what order they appear

- If I claim $\forall y \forall x P(x, y)$, then you get to pick both x and y, so again it doesn't matter what order they appear

17.2.4 **Implicit Quantifiers**

Sometimes quantifiers are implicit in English

Look for nouns (especially plural) without determiners (words to indicate which members of a group are intended)

- “Men are mortal” means “all men are mortal:” $\forall x(Man(x) \rightarrow Mortal(x))$
- “A woman is stronger than a man” would usually mean: $\forall x \forall y((Woman(x) \land Man(y)) \rightarrow Stronger(x,y))$
- “A woman is stronger than a man” could also mean: $\exists x \exists y(Woman(x) \land Man(y) \land Stronger(x,y))$
- Must consider context of statement; ask yourself what speaker was trying to say

17.2.5 **Ambiguity**

- Often, English statements are ambiguous; just do your best to determine what was intended

- Logic is unambiguous

- “A blue-eyed person is a dole bludger” could mean $(1)\forall x(Blueeyed(x) \rightarrow Dolebludger(x))$ *or* $(2)\exists x(Blueeyed(x) \wedge Dolebludger(x))$

- If previous sentence is “I know several blue-eyed people who are dole bludgers,” then (2) is probably intended

- If following sentence is “Joe has blue eyes, so Joe is a dole bludger,” then must have intended (1)

17.2.6 Choice of Predicates

Often there are many ways to encode an idea in predicate logic

"People who live in glass houses shouldn't throw stones" could be expressed in any of these ways:

- $\forall x((Person(x) \wedge Glasshouseresident(x)) \rightarrow \neg Shouldthrowstones(x))$
- $\forall x \forall y((Person(x) \wedge Glasshouse(y) \wedge Livesin(x,y)) \rightarrow \neg Shouldthrowstones(x))$
- $\forall x \forall y \forall z((Person(x) \wedge Glasshouseresident(x) \wedge Stone(y) \wedge Throwingaction(z) \wedge Performs(x,z,y)) \rightarrow \neg Advisable(z))$

Choose predicates to expose important concepts and hide unimportant detail, depending on your goals

Exercise

Given the following interpretation, translate the formula

$$\exists x(P(x) \land \forall y((P(y) \land x \neq y) \to h(x) > h(y)))$$

to English

$P(x)$	x is a person
$x > y$	x is greater than y
$x \neq y$	x and y are different
$h(x)$	the height of x

17.2.7 What Logic Cannot Handle

- Logic is not good with imprecise concepts
- Words like “many,” “most,” “few,” “usually,” “seldom,” *etc*, are not easy to handle
- Sentences like “most birds can fly” are difficult to express in logic

$\boxed{\forall x}$ 17.3 The Language of Predicate Logic

- Alphabet
  - constants (c)
  - variables (x, y in $P(x, y)$)
  - function letters (f in $P(f(x))$)
  - relation symbols (P in $P(x, y)$)
  - logical connectives ($\neg\ \wedge\ \vee\ \rightarrow \leftrightarrow$)
  - quantifiers ($\forall$, $\exists$)
  - punctuation symbols (,) and ,

$\boxed{\forall x}$ The Language of Predicate Logic (2)

- Understanding Quantifiers
  - $\forall x A(x)$ “For each and every thing, A(that thing) is true”
  - $\exists x A(x)$ “There is at least one thing for which A(that thing) is true”
- Terms (Kelly 6.4.1)

(i) every constant is a term

(ii) every variable is a term

(iii) If $t_1, \ldots, t_n$ are terms and f is a function letter of arity n, then $f(t_1, \ldots, t_n)$ is a term

(iv) nothing else is a term.

$\boxed{\forall x}$ The Language of Predicate Logic (3)

- A predicate symbol of arity n applied to n terms, *e.g.* $P(t_1, \ldots, t_n)$, is called an **atom**

  (note Prolog terminology conflicts)

- **Well formed formulae** (**wff**) (Kelly 6.4.1)

  (i) An atom is a wff

  (ii) if A and B are wff, so are $\neg A$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, $(A \leftrightarrow B)$

  (iii) if A is a wff and x is a variable, then $\forall x A$ is a wff

  (iv) if A is a wff and x is a variable, then $\exists x A$ is a wff

  (v) Nothing else is a wff

17.3.1 Scope, Bound and Free Variables

- scope: a quantifier applies to the formula immediately to its right
  - in $\forall x(P(x) \rightarrow Q(x))$, $\forall x$ applies to $P(x) \rightarrow Q(x)$
  - in $\forall x P(x) \rightarrow Q(x)$, $\forall x$ applies to $P(x)$
- a variable x occuring within the scope of a quantifier $\forall x$ or $\exists x$ is *bound*
- a variable outside the scope of any quantification of that variable is *free*

17.3.2 Bound, Free, Sentences and Formulae

- Examples
  - in $\exists x A(x, c)$, x occurs bound
  - in $\exists x(A(x, c) \wedge B(y))$, x occurs bound, y occurs free
  - in $\exists x(A(x, y) \wedge \forall y B(y))$, x occurs bound, y is free in its first occurrence and bound in its second
- Sentence
  - a **sentence**, or **closed formula**, is a wff where every variable occurrence is bound
  - e.g $\exists x \forall y(A(x, y) \wedge \exists z(B(z, y) \rightarrow B(x, z)))$

17.4 Summary

- Language of Predicate Logic extends Propositional Logic with predicates, functions, universal and existential quantifiers

- Order of quantifiers in formulae is significant

- English is ambiguous, translating to logic is not always straightforward

- Logic is unambiguous, translating to English is OK

- In $\forall x \mathcal{F}$ and $\exists x \mathcal{F}$, x is bound throughout $\mathcal{F}$

- Variable that is not bound is free

18 Determining Truth

- Domains of Interpretation
- Truth and Satisfaction
- Valuations

18.1 Domains of Interpretation

- Truth value of a Predicate Logic formula depends on the set of things we limit our attention to, called the **domain of interpretation** or **domain of discourse**

- A domain of interpretation is a non-empty set

- Can be anything: *e.g.*, the students taking this subject, the integers, the atoms in the universe

- Can only make statements about the objects in the domain

- Consider $\forall x (x \geq 0)$:
  - True if domain is the set of natural numbers
  - False if domain is the set of integers

18.1.1 Interpretations

An **interpretation** relates the constant, function, and relation symbols of our language to the objects, functions and relations over the domain

Gives a *meaning* to our language

Example: language with binary predicate symbols P and Q, unary function symbol f and constant symbol c

- domain U is the set of integers
- P_U the relation $\{(n, m) : n < m\}$ i.e. $\{(0,1), \ldots, (23, 12345), \ldots\}$
- Q_U the relation $\{(u, v) : v = u^2$ and $u > 0\}$ i.e. $\{(1,1), (2,4), \ldots\}$
- $f_U(x) = x + 7$
- c_U is 0

$\boxed{\forall x}$ Interpretations (2)

- We use domains of interpretation all the time without thinking about it
- In a discussion with friends, you might ask "has anyone read *The Hitchhiker's Guide to the Galaxy*?"
- Of course *someone* has read it; you're asking if anyone in the *group* of friends has read it
- You're implicitly restricting the domain of discourse to your group of friends
- "Nobody goes there anymore; it's too crowded."
  — Yogi Berra
- Only makes sense if domain of discourse is narrower than all people

18.1.2 **Valuations**

- A **valuation** over an interpretation is an assignment of domain objects to the free variables of the language

  example: $\theta = \{x \mapsto 7, y \mapsto 2, z \mapsto 4\}$

- A *term* is assigned a value in the domain under a valuation.

  ex: $f(c)$ under U and θ is $f_U(c_U) = 7$

  ex: $f(f(y))$ under U and θ is $f_U(f_U(\theta(y))) = f_U(f_U(2)) = 16$.

- An *atom* of the form $P(t_1, \cdots, t_n)$ is *true* (or *satisfied*) under interpretation U and valuation θ if t'_i is the interpretation of t_i under U and θ, and $(t'_1, \cdots, t'_n) \in P_U$.

  ex: $Q(y, z)$ is true under U and θ since $(2, 4) \in Q_U$.

  ex: $P(f(c), x)$ is not true under U and θ since $(7, 7) \notin P_U$.

18.1.3 Relation to Propositional Logic

- Predicate logic has interpretations and valuations
- Propositional logic just had assignments
- Can think of a *proposition* as a niladic (0-arity) predicate
- Then an interpretation fills the role of an assignment
- With no arguments, no terms and no variables
- With no variables, no need for quantifiers or valuations

$\boxed{\forall x}$ 18.2 Truth and Satisfaction

A formula A is *satisfied* by an interpretation U and some valuation θ over that interpretation:

- an atom is true or not according to the interpretation U, after applying the valuation θ

- connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$: as for Propositional Logic

- $\forall x A$: if for every value u in the domain of U, the interpretation U and the valuation $\theta' = (\theta - \{x \mapsto \_\}) \cup \{x \mapsto u\}$ satisfies $A(x)$.
  Alternatively $A(u_1) \wedge A(u_2) \wedge \cdots \wedge A(u_N)$ is satisfied

- $\exists x A$: if for some value u in the domain of U, the interpretation U and the valuation $\theta' = (\theta - \{x \mapsto \_\}) \cup \{x \mapsto u\}$ satisfies $A(x)$.
  Alternatively $A(u_1) \vee A(u_2) \vee \cdots \vee A(u_N)$ is satisfied

18.2.1 Truth, Satisfiability, Validity

- A formula A is **true in an interpretation** if it is satisfied by every valuation over that interpretation.
- For sentences (closed wffs), there are no free variables so the valuation is irrelevant
- A sentence A is **satisfiable** if it is true in some interpretation.
- A sentence A is **valid**, written $\models A$, if A is true in every interpretation.

$\boxed{\forall x}$ Truth, Satisfiability, Validity (2)

example: interpretations U where $P(x)$ is "x is a woman" :

- domain of U is {Mary, Kathleen, Teresa}:
  $\forall x P(x)$ is true in this interpretation
- domain of U is {Mary, Fred, Teresa}:
  $\forall x P(x)$ is false in this interpretation
- So, $\forall x P(x)$ is *satisfiable*, but not *valid*

$\boxed{\forall x}$ Exercise

In the interpretation of slide 423:

- domain U is the set of integers
- P_U the relation $\{(n,m) : n < m\}$
- Q_U the relation $\{(u,v) : v = u^2$ and $u > 0\}$
- $f_U(x) = x + 7$

which of the following are true:

1. $\forall x \exists y P(x,y)$
2. $\forall x \exists y Q(y,x)$
3. $\forall x \exists x P(x, f(x))$

18.3 Deciding Validity

To determine *validity* of a sentence:

- **What to do:** For all possible interpretations, use (propositional) refutation to show that the set of formulae is unsatisfiable.
- **How:** For each interpretation:
  1. Eliminate quantifiers by substituting all values from the domain for the variables
  2. *I.e.*, a big conjunction for $\forall$ and big disjunction for $\exists$
  3. Consider each atom to be a propositional variable
- Similarly for determining *satisfiability*: try different interpretations until one makes sentence true

18.3.1 The Problems

- There are infinitely many interpretations; can't check them all!

- Most interpretations are infinite, how to truth of quantified formulas?

- Jacques Herbrand (1929) solved first problem: restricted the number of interpretations that need to be considered to just one (the Herbrand interpretation)

- J. Alan Robinson (1965) solved second problem: work with variables directly using resolution with unification

18.4 Herbrand Approach

Herbrand's Theorem (1929) A set of clauses is unsatisfiable iff there is a finite subset of ground instances of its clauses which is unsatisfiable as propositional logic formulas

- Each ground instance of an atom can be treated as a separate propositional variable
- *E.g.*, $P(Mary), P(Fred), P(Teresa)$ can each be assigned either true or false
- But what about $P(x)$?

18.4.1 **Herbrand Interpretation**

- Herbrand's Theorem effectively says we don't need to consider the infinite number of possible interpretations
- We can consider only the Herbrand Interpretation of the formula
- Given a formula f, the **Herbrand Interpretation** of f maps every symbol in f to itself
- *I.e.*, instead of having to worry about what each symbol might mean, we only consider the symbols themselves
- Our domain of interpretation is just the symbols of the formula

18.4.2 Herbrand Universe

To find the **Herbrand universe** H for a set S of clauses:

- If c is a constant occurring in a clause in S, H contains c
- If no constants occur in any clause in S, add a new constant symbol to H
- if f is a function symbol of arity n occurring in any clause in S, and H contains $t_1, \cdots, t_n$, add $f(t_1, \cdots, t_n)$ to H
- Repeat *ad infinitum*

If S contains *any* function symbols, then H is infinite

18.4.3 Herbrand Base

The **Herbrand base** B of a set S of clauses is defined as follows

- If P is a predicate symbol appearing in $t_1, \cdots, t_n$ are members of the Herbrand universe of S, then $P(t_1, \cdots, t_n)$ is in B
- Nothing else is in B

The Herbrand base is the set of all “propositions” we need to find truth assignments for

If the Herbrand universe is infinite, so is the Herbrand base

18.4.4 Example

- Fido is a dog
  All hungry dogs bark
  Fido is hungry
  Therefore, Fido barks

- $D(F)$
  $\forall x(H(x) \land D(x) \rightarrow B(x))$
  $H(F)$
  $\therefore B(F)$

- Herbrand universe $= \{F\}$

- Herbrand base $= \{D(F), H(F), B(F)\}$

- Only instances of $\forall x(H(x) \land D(x) \rightarrow B(x))$ from the Herbrand base

- That's just $H(F) \land D(F) \rightarrow B(F))$

$\boxed{/x}$ Example (2)

- Now it’s all just propositional logic — easy!
- $D(F)$
  $H(F) \wedge D(F) \rightarrow B(F)$
  $H(F)$
  $\therefore B(F)$
- This is argument is valid
- So by Herbrand’s theorem, the original predicate logic formula was valid

18.4.5 On To Resolution

- Not always so easy
- Usually the Herbrand universe and base are infinite
- Sadly, Herbrand died aged 23 before he could solve this problem
- Solution to this problem would wait until Robinson's resolution principle

18.4.6 **Summary**

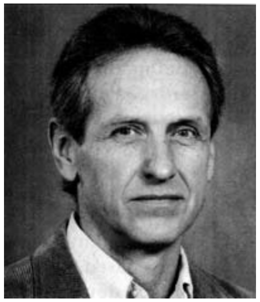
- *Interpretation* of a formula is what each of its symbols mean
- *Domain of interpretation* is the set of things involved in interpretation
- A formula is *valid* if it is true in *all* interpretations
- Herbrand's theorem says we only need to consider *Herbrand interpretation*, which interprets each symbol as itself
- *Herbrand universe* is the set of all terms that can be made from functions and constants of the formula

$\boxed{\forall x}$ 19 Resolution in Predicate Logic

- **Aim:** Use resolution to show that a Predicate Logic formula is unsatisfiable
- **Application:** Proving that arguments which are expressible in predicate logic are correct
- **Problem:** Resolution only defined for Propositional Logic so far
- **Strategy:** Adapt Propositional techniques to Predicate Logic

History

- Resolution: J. Alan Robinson, 1965
  - Computerized theorem prover
- Logic Programming: Robert Kowalski, 1974
  - A program is a conjunction of implications
  - Each implication gives one way a statement can be true
  - Execution is finding ways a statement can be true

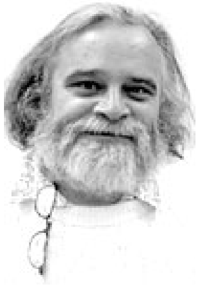


History (2)

- Prolog: Alain Colmerauer, 1973
  - Designed for natural language understanding

- Prolog compiler: David H. D. Warren, 1977
  - Quite efficient
  - Modern Prolog syntax follows this implementation

- Constraint Logic Programming: Jaffar and Lassez, 1987
  - Generalizes Logic Programming
  - Developed at University of Melbourne



19.0.7 **Pre-Processing for Predicate Logic**

We extend the algorithm for converting a formula to conjunctive normal form to enable us to deal with variables and quantifiers.

For each formula:

1. Convert to Prenex Normal Form (CNF with all quantifiers at the start of the formula).
2. Eliminate existential quantifiers using Skolemisation.
3. Drop the universal quantifiers and convert to clausal form.

19.0.8 Prenex Normal Form Algorithm

Stage 1 Use $A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$ to eliminate ↔

Stage 2 Use $A \rightarrow B \equiv \neg A \vee B$ to eliminate →

Stage 3 Use

a) De Morgan's Laws

b) $\neg \forall x A(x) \equiv \exists x \neg A(x)$

c) $\neg \exists x A(x) \equiv \forall x \neg A(x)$

to push ¬ immediately before atomic wff

Simplify formula using $\neg\neg A \equiv A$

Stage 4 Rename bound variables if necessary

Stage 5 Use the equivalences on (Kelly page 192) to bring all quantifiers to the left

Stage 6 "Distribute" ∧, ∨

19.0.9 Example

Convert to Prenex Normal Form:

$$\forall x\Big(A(x) \to B(x)\Big) \leftrightarrow \exists x Q(x)$$

1. $\Big(\forall x\Big(A(x) \to B(x)\Big) \boxed{\to} \exists x Q(x)\Big) \boxed{\wedge} \Big(\exists x Q(x) \boxed{\to} \forall x\Big(A(x) \to B(x)\Big)\Big)$

2. $\Big(\boxed{\neg} \forall x\Big(\boxed{\neg} A(x) \boxed{\vee} B(x)\Big) \boxed{\vee} \exists x Q(x)\Big) \wedge \Big(\boxed{\neg} \exists x Q(x) \boxed{\vee} \forall x\Big(\boxed{\neg} A(x) \boxed{\vee} B(x)\Big)\Big)$

3a. $\Big(\boxed{\exists x \neg} \Big(\neg A(x) \vee B(x)\Big) \vee \exists x Q(x)\Big) \wedge \Big(\boxed{\forall x \neg} Q(x) \vee \forall x\Big(\neg A(x) \vee B(x)\Big)\Big)$

3b. $\Big(\exists x\Big(\boxed{\neg} \neg A(x) \boxed{\wedge} \boxed{\neg} B(x)\Big) \vee \exists x Q(x)\Big) \wedge \Big(\forall x \neg Q(x) \vee \forall x\Big(\neg A(x) \vee B(x)\Big)\Big)$

4. $\Big(\exists \boxed{w} \Big(A(\boxed{w}) \wedge \neg B(\boxed{w})\Big) \vee \exists x Q(x)\Big) \wedge \Big(\forall \boxed{y} \neg Q(\boxed{y}) \vee \forall \boxed{z} \Big(\neg A(\boxed{z}) \vee B(\boxed{z})\Big)\Big)$

5a. $\boxed{\exists w \exists x} \Big(\Big(A(w) \wedge \neg B(w)\Big) \vee Q(x)\Big) \wedge \boxed{\forall y \forall z} \Big(\neg Q(y) \vee \Big(\neg A(z) \vee B(z)\Big)\Big)$

5b. $\boxed{\exists w \exists x \forall y \forall z} \Big(\Big((A(w) \wedge \neg B(w)) \vee Q(x)\Big) \wedge \Big(\neg Q(y) \vee \neg A(z) \vee B(z)\Big)\Big)$

6. $\exists w \exists x \forall y \forall z \Big(\boxed{\Big(A(w) \vee Q(x)\Big) \wedge \Big(\neg B(w) \vee Q(x)\Big)} \wedge \Big(\neg Q(y) \vee \neg A(z) \vee B(z)\Big)\Big)$

$\boxed{\forall x}$ Exercise

Convert the following formula to prenex normal form:

$$\forall x(P(x) \to ((\exists y Q(x, y)) \land (\exists y R(x, y))))$$

19.0.10 Skolemisation

- Suggested by Thoralf Skolem
- **Aim:** Get rid of ∃ quantifiers
- **How:** Introduce *new* constants and functions, *not already existing* in the formula, to replace variables corresponding to existentially quantified variables. Result is the *Skolemisation* of the formula.
- **Definition:**
  - ∃ outside ∀: replace by new constant.
    $\exists x \forall y A(x,y)$ becomes $\forall y A(c,y)$.
  - ∃ inside ∀: replace by new function with arguments all variables in outside ∀s.
    $\forall y \exists x \forall z \exists w A(x,y,z,w) \wedge B(y,w)$ becomes
    $\forall y \forall z A(f_1(y), y, z, f_2(y,z)) \wedge B(y, f_2(y,z))$

19.0.11 **Intuition**

- If $\exists x \forall y P(x, y)$ then there is some x that satisfies $P(x, y)$ for every y
- But we don't know *which* x
- So we make up a name: we create a new constant and define it to be some constant satisfying P
- New constant will represent be the same thing as some old constant, but that's OK
- *E.g.*, $P(x, y)$ means $y = x + y$
- Convert to $\forall y P(c, y)$
- c happens to be 0, but we don't need to know that

Intuition (2)

- If $\forall y \exists x P(x,y)$ then for each y there is some x that satisfies $P(x,y)$
- Now the x that satisfies $P(x,y)$ depends on y: it's a function of y
- So now we make up a function: for any y, we say that $f(y)$ satisfies $P(f(y),y)$
- *E.g.*, $P(x,y)$ means $x > y$
- convert to $\forall y P(f(y),y)$
- $f(x)$ could be $x+1$, but we don't care

19.0.12 Important Result

Theorem. Let A be a formula and A' its Skolemisation. Then A is satisfiable iff A' is satisfiable.

- A and A' are not (in general) equivalent, since there can be symbols in A' not occurring in A.
- So the Skolemised formula (no $\exists$ quantifiers) can be used in resolution.
- Prolog clauses do not allow existentially quantified variables, so there is no Skolemisation to be done! (Fortunate, since we generally don't want two distinct constants representing the same thing in Prolog)

19.0.13 Clausal Form

To find the clausal form of a formula:

1. Convert to Prenex normal form and then Skolemise.
2. Drop all $\forall$ quantifiers.
3. Convert to clausal form as for Propositional Logic.

- Once in Prenex normal form, all variables are universally quantified and at the front of each clause
- Order of same quantifiers is unimportant
- Drop them, then consider all variables implicitly universally quantified
- Simplifies resolution algorithm

Exercise

Skolemize the following formula, then put it into clausal form:

$$\forall y \exists x (P(y) \vee \neg Q(x, y))$$

An Example

Use resolution to prove that the following argument is valid.

All hungry animals are caterpillars.
All caterpillars have 42 legs.
Edward is a hungry caterpillar.
Therefore, Edward has 42 legs.

Translation to Logic:

$$\forall x(H(x) \to C(x)) \qquad H(\text{Edward}) \land C(\text{Edward})$$

$$\forall x(C(x) \to L(x)) \qquad \therefore L(\text{Edward})$$

Prenex Normal Form

$$\begin{array}{ll} \forall x(H(x) \rightarrow C(x)) & \qquad H(\text{Edward}) \wedge C(\text{Edward}) \\ \forall x(C(x) \rightarrow L(x)) & \qquad \therefore L(\text{Edward}) \end{array}$$

Negate conclusion and convert to Prenex Normal Form:

$$\begin{array}{rrcl} & \forall x(\neg H(x) \vee C(x)) & \wedge & H(\text{Edward}) \\ \wedge & \forall x(\neg C(x) \vee L(x)) & \wedge & C(\text{Edward}) \\ & & \wedge & \neg L(\text{Edward}) \end{array}$$

$\boxed{\forall x}$ Clausal form

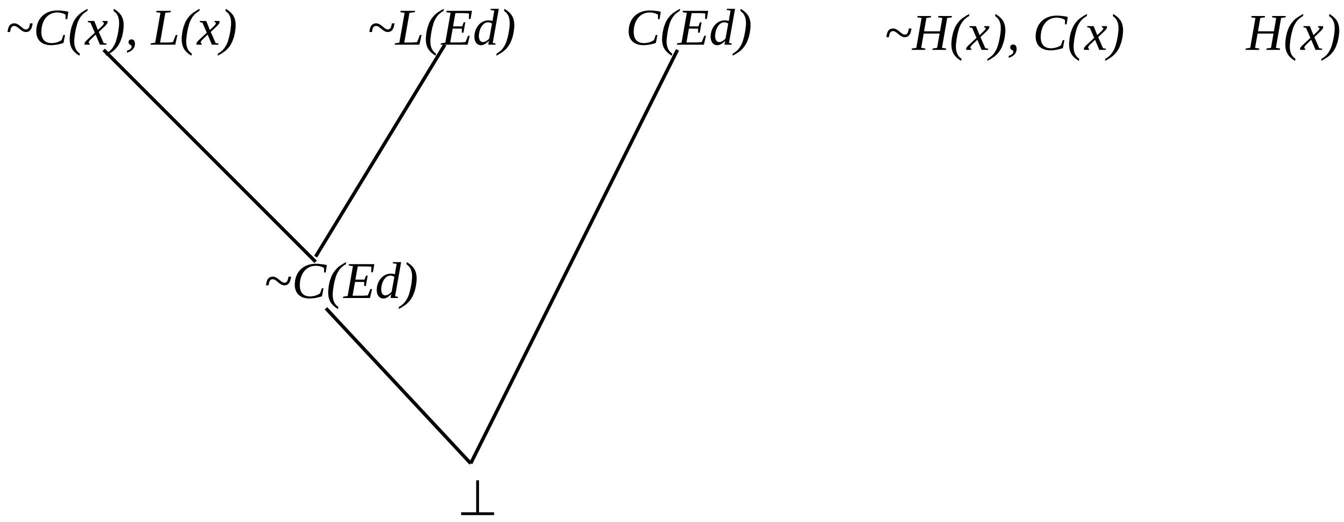
$$
\begin{array}{rrcl}
 & \forall x(\neg H(x) \vee C(x)) & \wedge & H(\mathrm{Edward}) \\
\wedge & \forall x(\neg C(x) \vee L(x)) & \wedge & C(\mathrm{Edward}) \\
 & & \wedge & \neg L(\mathrm{Edward})
\end{array}
$$

Set of sets representation:

$$
\left\{
\begin{array}{ll}
\{\neg H(x), C(x)\}, & \{H(\mathrm{Edward})\}, \\
\{\neg C(x), L(x)\}, & \{C(\mathrm{Edward})\}, \\
 & \{\neg L(\mathrm{Edward})\}
\end{array}
\right\}
$$

Resolution Refutation

$$\left\{ \begin{array}{ll} \{\neg H(x), C(x)\}, & \{H(\text{Edward})\}, \\ \{\neg C(x), L(x)\}, & \{C(\text{Edward})\}, \\ & \{\neg L(\text{Edward})\} \end{array} \right\}$$



Just one refutation, there are *many*.

Why $C(Ed)$ on second line? See in next lecture

19.0.14 Summary: Herbrand Approach

Problems

- set of ground clauses may be infinite
- no adequate means to guide the search for refutation
- validity problem for predicate logic is undecidable (problem with *any* method)

20 Resolution with Unification

How did Robinson solve the problem of detecting unsatisfiability in predicate logic, and how did that give rise to logic programming?

- Robinson's Approach
- Unification
- Resolvents
- Resolution Theorems
- Horn Clauses and Prolog
- Summary: Robinson's Approach

20.1 Robinson’s Approach

- **Robinson’s Theorem** (1965) A set of clauses is unsatisfiable iff it has a resolution refutation, where unification is used to match complementary literals.

- Avoid generating (possibly infinitely many) ground instances of clauses.

- Still have search control problem but greatly reduced.

20.2 Unification

- A **substitution** is a finite set of replacements of variables by terms, i.e. a set σ of the form $\{t_1/x_1, t_2/x_2, \cdots, t_n/x_n\}$, where the x_i are variables and the t_i are terms. If A is a formula, $A\sigma$ is the result of **simultaneously** making the substitutions t_i for each occurrence of x_i.

  e.g. if $A = \{P(x), Q(y, y, b)\}$, $\sigma = \{h(y)/x, a/y, c/z\}$ then $A\sigma = \{P(h(y)), Q(a, a, b)\}$
  Note similar but not the same as a valuation.

- A **unifier** of two literals L_1 and L_2 is a substitution σ such that $L_1\sigma = L_2\sigma$.

- L_1 and L_2 are **unifiable** if there exists a unifier for L_1 and L_2

Unification (2)

- A **most general unifier** (**m.g.u.**) for L_1 and L_2 is a substitution σ such that σ is a unifier for L_1 and L_2 and every other unifier τ of L_1 and L_2 can be expressed as $\sigma\theta$ for some substitution θ.

  ($A\sigma\theta$ is the composition of σ and θ applied to A, *i.e.* performing σ first, then θ).

- Extend to sets of literals $S = \{L_1, \ldots, L_n\}$ in the obvious way. σ_1 unifies L_1 and L_2, σ_2 unifies $L_1\sigma_1$ and L_2, . . . , σ_{n-1} unifies $L_1\sigma_1 \cdots \sigma_{n-2}$ and L_n. The unifier of S is $\sigma_1 \cdots \sigma_{n-1}$.

- **Lemma.** If a set S is unifiable, then it has a most general unifier.

- **Lemma.** A most general unifier for a set S (if it exists) is unique up to renaming of variables.

20.2.1 Examples

- $\{P(x,a), P(b,c)\}$ is not unifiable

- $\{P(f(x),y), P(a,w)\}$ is not unifiable.

- $\{P(x,c), P(b,c)\}$ is unifiable by $\{b/x\}$.

- $\{P(f(x),y), P(f(a),w)\}$ is unifiable by $\sigma = \{a/x, w/y\}$, $\tau = \{a/x, a/y, a/w\}$, $\upsilon = \{a/x, b/y, b/w\}$
  Note that σ is an m.g.u. and $\tau = \sigma\theta$ where $\theta = \cdots$

- $\{P(x), P(f(x))\}$ is not unifiable. (c.f. occur check!)

20.2.2 **Unification Algorithm**

1. $T_1 = L_1$; $T_2 = L_2$; $\sigma_0 = \{\}$; $i = 0$

2. If T_1 is identical to T_2, terminate with σ_i as the most general unifier.

3. Find the leftmost position where T_1 and T_2 differ. If the two terms at this position are a variable v_i and a term t_i such that v_i does not occur in t_i (the *occurs check*), then

   $$\sigma_{i+1} = \sigma_i\{t_i/v_i\}; \quad T_1 = T_1\{t_i/v_i\}; \quad T_2 = T_2\{t_i/v_i\}$$

   Otherwise, terminate as S is not unifiable.

4. $i = i + 1$; resume from step 2.

$\boxed{/x}$ Unification Algorithm (2)

- $T_1 = L_1 = f(\textbf{x}, g(x)) \qquad T_2 = L_2 = f(\textbf{h(y)}, g(h(z)))$

  $\sigma_1 = \{h(y)/x\}$
  $T_1 = f(h(y), g(h(\textbf{y}))) \qquad T_2 = f(h(y), g(h(\textbf{z})))$

  $\sigma_2 = \{h(y)/x, y/z\}$
  $T_1 = f(h(y), g(h(y))) \qquad T_2 = f(h(y), g(h(y)))$
  i.e. σ_2 is an m.g.u.

20.2.3 Results on Unification

- **Theorem.** The unification algorithm is correct.
- **Remark.** The occur check can be exponential in the size of the original terms, so Prolog simply omits this part of the algorithm.
- **Question.** What does this mean for Prolog's correctness?
- **Question.** What happens when you type the following queries to Prolog?

```
?- X = f(X).
?- X = f(X), Y = f(Y), X=Y.
```

Exercise

Find the *most general unifier* of the following pairs of terms, if possible. (a, b are constants, x, y, z are variables.)

1. $f(x, y, f(x, y))$ $\qquad$ $f(b, z, z)$
2. $f(f(x, y), y)$ $\qquad$ $f(z, g(a))$

20.3 Resolvents

- Recall: For Propositional Logic:
  - Two literals are complementary if one is L and the other is $\neg L$.
  - The *resolvent* of two clauses containing complementary literals L, $\neg L$ is their union omitting L and $\neg L$.
- For Predicate Logic:
  - Two literals L and $\neg L'$ are *complementary* if $\{L, L'\}$ is unifiable.
  - Let σ be an m.g.u. of complementary literals $\{L, L'\}$ with L a literal in C_1 and $\neg L'$ a literal in C_2. Then the *resolvent* of C_1 and C_2 is the union of $C_1\sigma$ and $C_2\sigma$ omitting $L\sigma$ and $\neg L'\sigma$.

20.4 Resolution Theorems

Resolution Principle

If D is a resolvent of clauses C_1 and C_2, then $C_1 \wedge C_2 \models D$.

Resolution Theorem

A clause set S is unsatisfiable iff $\bot \in R^*(S)$, where $R^*(S)$ is the set of *all* clauses obtainable after a finite number of resolution steps, starting with S.

20.4.1 Example: The Miserable Tadpole

every shark eats a tadpole

$\forall x(S(x) \rightarrow \exists y(T(y) \land E(x,y))) \qquad \{\neg S(x), T(f(x))\}, \{\neg S(x), E(x, f(x))\}$

all large white fish are sharks

$\forall x(W(x) \rightarrow S(x)) \qquad \{\neg W(x), S(x)\}$

colin is a large white fish living in deep water

$W(colin) \land D(colin) \qquad \{W(colin)\}, \{D(colin)\}$

any tadpole eaten by a deep water fish is miserable

$\forall z(T(z) \land \exists y(D(y) \land E(y,z) \rightarrow M(z)) \qquad \{\neg T(z), \neg D(y), \neg E(y,z), M(z)\}$

negation of: some tadpoles are miserable

$\neg\exists z(T(z) \land M(z)) \qquad \{\neg T(z), \neg M(z)\}$

$\boxed{/x}$

20.4.2 Refutation

C1	$\neg S(x),\ T(f(x))$	C2	$\neg S(x),\ E(x, f(x))$
C3	$\neg W(x),\ S(x)$	C4	$W(c)$
C5	$D(c)$	C6	$\neg T(z),\ \neg D(y),\ \neg E(y, z),\ M(z)$
C7	$\neg T(z),\ \neg M(z)$		

C8	$S(c)$	C4, C3 $\{c/x\}$
C9	$T(f(c))$	C8, C1 $\{c/x\}$
C10	$E(c, f(c))$	C8, C2 $\{c/x\}$
C11	$\neg D(y),\ \neg E(y, f(c)),\ M(f(c))$	C9, C6 $\{f(c)/z\}$
C12	$\neg D(c),\ M(f(c))$	C10, C11 $\{c/y\}$
C13	$M(f(c))$	C5, C12
C14	$\neg T(f(c))$	C7, C13 $\{f(c)/z\}$
C15	$\perp$	C9, C14

So there is a miserable tadpole – the one eaten by colin.

20.5 Horn Clauses and Prolog

- Horn clause = clause with at most one positive literal.
  e.g. $\{p(x), \neg q(x,y), \neg r(x,y)\}$
  represents $p(x) \vee \neg q(x,y) \vee \neg r(x,y)$
  rewritten as $p(x) \vee \neg(q(x,y) \wedge r(x,y))$
  rewritten as $p(x) \leftarrow (q(x,y) \wedge r(x,y))$
  or, in Prolog notation `p(X) :- q(X, Y), r(X, Y).`

- SLD-resolution is Robinson's approach restricted to linear input resolution.

20.5.1 Example

- Suppose C is the clause set
  $\{\{q(x), \neg p(x), \neg s(x)\}, \{p(x), \neg r(x)\}, \{r(a)\}, \{r(b)\}, \{s(b)\}\}$
  Does $C \models q(b)$?

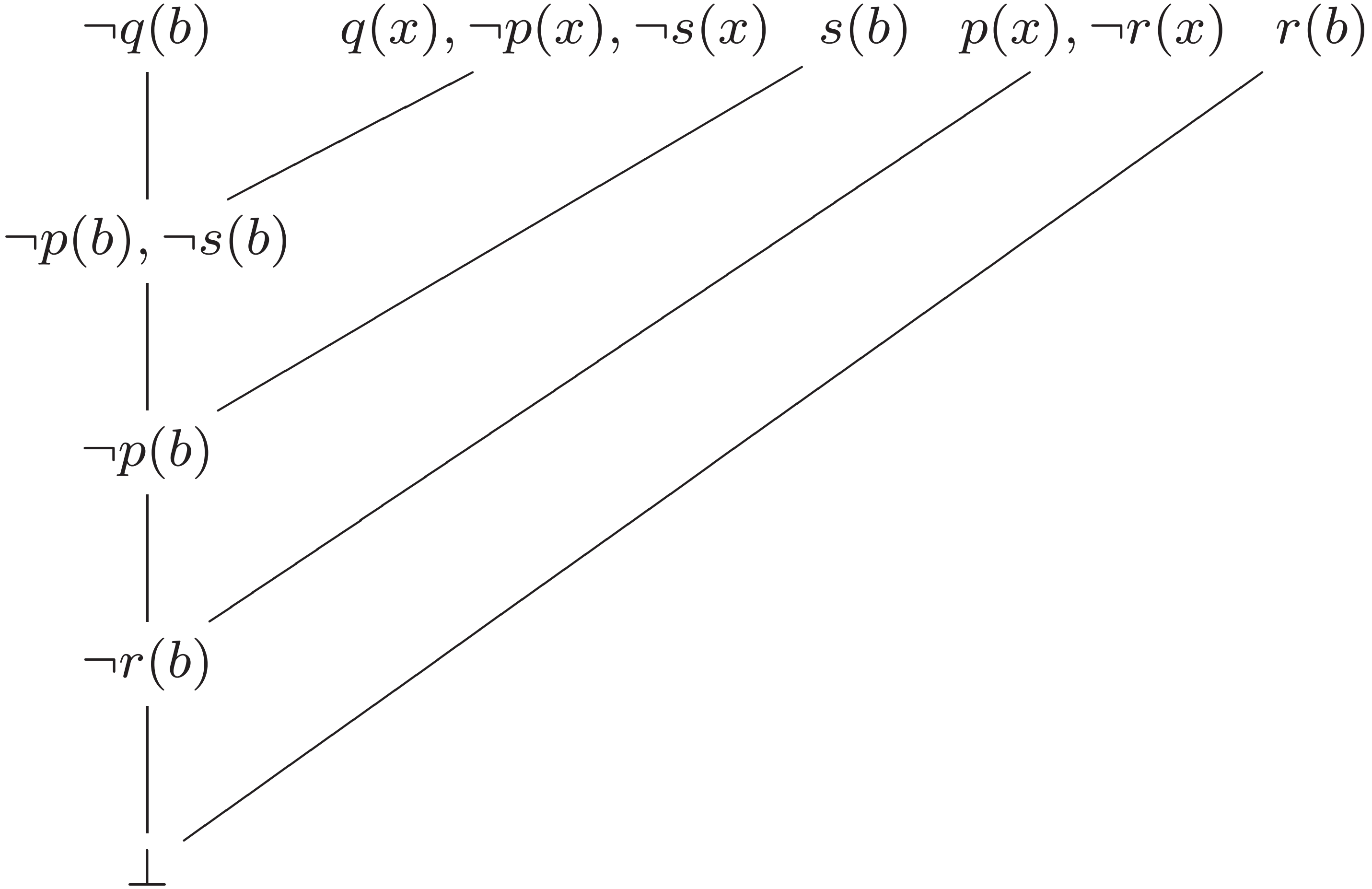
- check for unsatisfiability of $C \cup \{\neg q(b)\}$, i.e.

  1. $\{q(x), \neg p(x), \neg s(x)\}$
  2. $\{p(x), \neg r(x)\}$
  3. $\{r(a)\}$
  4. $\{r(b)\}$
  5. $\{s(b)\}$
  6. $\{\neg q(b)\}$

- and (in a few steps of linear resolution) you derive $\perp$

20.5.2 Linear Resolution Diagram

Example, continued using linear input resolution.



$\neg q(b)$ $\quad q(x), \neg p(x), \neg s(x)$ $\quad s(b)$ $\quad p(x), \neg r(x)$ $\quad r(b)$

$\neg p(b), \neg s(b)$

$\neg p(b)$

$\neg r(b)$

$\bot$

$\boxed{/x}$ Exercise

Draw a resolution diagram refuting this set of clauses:

$\{P(a,b)\}$ $\{\neg P(x,y), Q(x,y)\}$ $\{\neg P(x,y), Q(y,x)\}$ $\{\neg Q(b,a)\}$

20.5.3 (Abstract) Interpreter for Prolog

```
Input: A query $Q$ and a logic program $P$
Output: yes if $Q$ ‘implied’ by $P$, no otherwise
  Initialise current goal set to $Q$;
  While the current goal is not empty do
    Choose $G$ from the current goal;
    Choose instance of a rule $G'$ :- $B_1, \ldots, B_n$ from $P$;
      renaming all variables to new symbols giving $G''$ :- $B'_1$, . . . , $B$
      such that $G$ and $G''$ are unifiable, with mgu $\sigma$
    (if no such rule exists, exit while loop)
    Replace $G$ by $B'_1\sigma$, . . . , $B'_n\sigma$ in current goal set;
  If current goal set is empty,
    output yes;
  else output no;
```

20.5.4 **Prolog in Prolog**

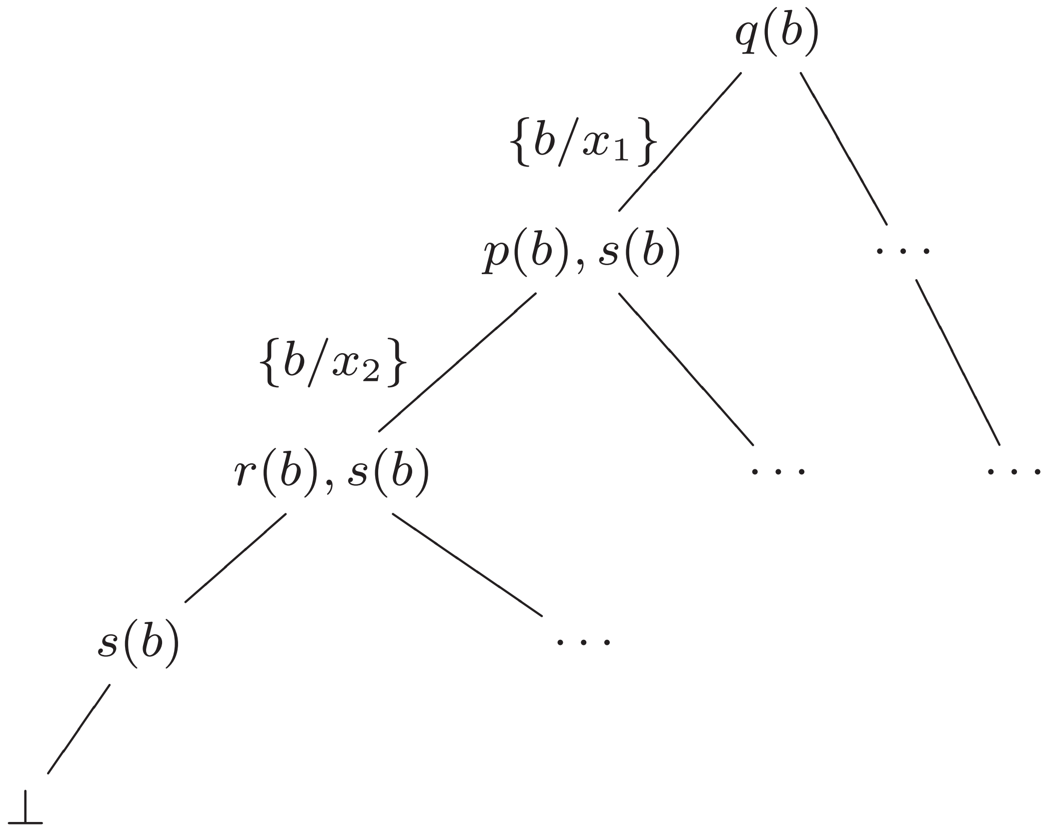
A basic Prolog interpreter in Prolog:

```
solve(true).
solve((G1,G2)) :-
        solve(G1),
        solve(G2).
solve(Goal) :-
        clause(Goal, Body),
        solve(Body).
```

Needs to be expanded to support built-ins, disjunction, if-then-else, *etc*.

NB: Prolog takes care of backtracking and creating copies of clauses with fresh variables for us!

20.5.5 SLD tree



$q(x)$:- $p(x), s(x)$.
$p(x)$:- $r(x)$.
$r(a)$.
$r(b)$.
$s(b)$.

- branching represents choice points (revisited on backtracking)
- x_1, x_2 are copies of x from original program

20.5.6 **Summary: Robinson's Approach**

- based on resolution with unification of complementary literals

- sound and complete inference method

- still need search control information to define an implementation

- Prolog uses this method restricted to Horn clauses and linear input resolution (plus rule and literal selection strategy)

21 Semantics of Logic Programs

It is useful to be able to formally characterize what a logic program means. This allows us to formally analyze programs.

1. The Meaning of “Meaning”

2. The Meaning of a Logic Program

3. Finding Meaning

4. Approximating Meaning

5. Groundness Analysis

21.1 The Meaning of “Meaning”

What does this C function “mean?”

```
int sq(int n)
{
    return n * n;
}
```

What about this?

```
int sq(int n)
{
    int i, sum;
    for (i=0, sum=0; i<n; ++i) {
        sum += n;
    }
    return sum;
}
```

The Meaning of “Meaning” (2)

What does this C function mean?

```
void hi(void)
{
    printf("hello, world!\n");
}
```

or this:

```
int count;
void inc(void)
{
    count++;
}
```

Meaning of a C function must include what it *does* as well as what it *returns*

The Meaning of “Meaning” (3)

Meaning of a program component gives understanding of that component

Meaning of Haskell function

```
fact  :: Integer -> Integer
fact n = product [1..n]
```

is (mathematical) factorial function:

$$\{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 6, 4 \mapsto 24, \ldots\}$$

Note: meaning is infinite set

21.2 The Meaning of a Logic Program

Meaning of logic program is what it makes true

Ignore Input/Output, assert/retract, *etc*.

Meaning of a program can be shown as a set of unit clauses (facts)

This program *is* its meaning:

```
capital(tas, hobart).   capital(vic, melbourne).
capital(nsw, sydney).   capital(sa, adelaide).
capital(act, canberra). capital(qld, brisbane).
capital(nt, darwin).    capital(wa, perth).
```

The Meaning of a Logic Program (2)

```
parent(alice, harriet).  parent(alice, george).
parent(bob, harriet).    parent(bob, george).
parent(harriet, laura).  parent(harriet, ken).

grandparent(C, G) :-
        parent(C, P),
        parent(P, G).
```

Meaning is:

```
parent(alice, harriet).    parent(alice, george).
parent(bob, harriet).      parent(bob, george).
parent(harriet, laura).    parent(harriet, ken).

grandparent(alice, laura). grandparent(alice, ken).
grandparent(bob, laura).   grandparent(bob, ken).
```

The Meaning of a Logic Program (3)

Meaning of a recursive predicate is also a set of clauses

E.g., what is the meaning of:

```
num(0).
num(s(N)) :- num(N).
```

`num`/1 holds for numbers written in successor notation, so meaning is:

```
num(0).
num(s(0)).
num(s(s(0))).
num(s(s(s(0)))).
...
```

21.3 Finding Meaning

Meaning of program P can be found using *immediate consequence* function T_P

$$T_P(I) = \{H\theta \mid (H :- G_1, \ldots G_n) \in P \ \wedge \ G_1\theta \in I \ \wedge \ \ldots G_n\theta \in I\}$$

Here θ is any substitution that unifies G_1, $\ldots G_n$ with elements of I and grounds all variables in H

Take $T_P^2(I) = T_P(T_P(I))$, $T_P^3(I) = T_P(T_P(T_P(I)))$, *etc*.

Meaning of program P is $T_P^\infty(\emptyset)$

Example

Take program $P = \{num(0), num(s(N)) : -num(N)\}$

$T_P^1(\emptyset) = \{num(0)\}$
(because body of clause is empty)

$T_P^2(\emptyset) = \{num(0), num(s(0))\}$

$T_P^3(\emptyset) = \{num(0), num(s(0)), num(s(s(0)))\}$

$T_P^\infty(\emptyset) =$
$\{num(0), num(s(0)), num(s(s(0))), num(s(s(s(0)))), \ldots\}$

Meaning of most programs is infinite

Exercise

Give the semantics of the following Prolog program:

```
p(a,b).
p(b,c).
p(c,d).
q(X,Y) :- p(X,Y).
q(X,Y) :- p(Y,X).
```

21.4 Why Study Semantics?

- Programming (thinking about your code)
- Verification (is it correct)
- Debugging (the answer is wrong, but why)
- Implementation
- Language design
- Philosophy

Hundreds of papers have been written on the semantics of logic programs, many concerning negation

21.4.1 **Verification and Debugging**

Suppose each clause/predicate of a program is true according to my intended interpretation

Then everything computed by the program (the meaning, the set of logical consequences) is true

Correctness can be *verified* just by reasoning about individual components

If the program computes something false, one of clauses/predicates must be false

Declarative debugging systems use the intended interpretation to find which one

21.5 **Approximating Meaning**

Can also *approximate* meaning of a program: gives *some* information about the program

Approximation can be made finite: can actually be computed

Usual approach: replace data in program by an abstraction, then compute meaning

Many interesting program properties are Boolean: use propositional logic

E.g., parity (even/odd): abstract `0` by $true$

Abstract `s(X)` by $\neg x$

Approximating Meaning (2)

Abstracted `num` predicate:

$$\begin{array}{ll} num(true) & num(0) \\ num(\neg x) \leftarrow num(x) & num(s(X)) \leftarrow num(X) \end{array}$$

$T_P^1(\emptyset) = \{num(true)\}$
$T_P^2(\emptyset) = \{num(true), num(false)\}$
$T_P^3(\emptyset) = \{num(true), num(false)\}$

Can stop here: further repetitions will not add anything

Result is finite

Says that both even and odd numbers satisfy `num/1`

Approximating Meaning (3)

Program:

```
plus(0, Y, Y).
plus(s(X), Y, s(Z)) :- plus(X, Y, Z).
```

Abstracted version:

$plus(true, y, y)$ $\quad$ y can be either $true$ or $false$

$plus(\neg x, y, \neg z) \leftarrow plus(x, y, z)$

$$T_P^1(\emptyset) = \{plus(true, true, true), plus(true, false, false)\}$$

$$T_P^2(\emptyset) = \left\{ \begin{array}{l} plus(true, true, true), plus(true, false, false), \\ plus(false, true, false), plus(false, false, true) \end{array} \right\}$$

$$T_P^3(\emptyset) = \left\{ \begin{array}{l} plus(true, true, true), plus(true, false, false), \\ plus(false, true, false), plus(false, false, true) \end{array} \right\}$$

Approximating Meaning (4)

This result is better shown as a table:

+	even	odd
even	even	odd
odd	odd	even

Even such a simple analysis is able to discover interesting properties of programs

Exercise

Use parity analysis to determine the possible parities of the arguments of `p/1` defined as:

```
p(0,s(0)).
p(s(X), s(s(Y))) :- p(X, Y).
```

21.6 Groundness Analysis

A more useful property for optimization of Prolog code is groundness

Again, a Boolean property

```
        append([], Y, Y).
        append([U|X], Y, [U|Z]) :- append(X, Y, Z).
```

Abstracted version:

$$append(true, y, y)$$
$$append(u \wedge x, y, u \wedge z) \leftarrow append(x, y, z)$$

In second clause, u can be $true$ or $false$, so:

$$append(true, y, y)$$
$$append(x, y, z) \leftarrow append(x, y, z)$$
$$append(false, y, false) \leftarrow append(x, y, z)$$

Groundness Analysis (2)

$$append(true, y, y)$$
$$append(x, y, z) \leftarrow append(x, y, z)$$
$$append(false, y, false) \leftarrow append(x, y, z)$$

$$T_P^1(\emptyset) = \{append(true, true, true), append(true, false, false)\}$$

$$T_P^2(\emptyset) = \left\{ \begin{array}{l} append(true, true, true), append(true, false, false), \\ append(false, true, false), append(false, false, false) \end{array} \right\}$$

$$T_P^2(\emptyset) = \left\{ \begin{array}{l} append(true, true, true), append(true, false, false), \\ append(false, true, false), append(false, false, false) \end{array} \right\}$$

Groundness Analysis (3)

A truth table will help us understand this:

x	y	z	$append(x, y, z) \in T_P^\infty(\emptyset)$
true	true	true	true
true	true	false	false
true	false	true	false
true	false	false	true
false	true	true	false
false	true	false	true
false	false	true	false
false	false	false	true

We see when $append(x, y, z)$ is true, $(x \wedge y) \leftrightarrow z$

Groundness Analysis (4)

So after a call `append(X,Y,Z)`, `X` and `Y` will be ground iff `Z` is

We can use this result in analyzing predicates that call append

```
rev([], []).
rev([U|X], Y) :-
        rev(X, Z),
        append(Z, [U], Y).
```

Abstracted version:

$$rev(true, true)$$

$$rev(u \wedge x, y) \leftarrow (rev(x, z) \wedge append(z, u, y))$$

Groundness Analysis (5)

Using what we've learned about append:

$$rev(true, true)$$
$$rev(u \wedge x, y) \leftarrow (rev(x, z) \wedge ((z \wedge u) \leftrightarrow y))$$

Since u can be either $true$ or $false$:

$$rev(true, true)$$
$$rev(x, y) \leftarrow (rev(x, z) \wedge (z \leftrightarrow y))$$
$$rev(false, y) \leftarrow (rev(x, z) \wedge (false \leftrightarrow y))$$

Simplifying biimplications:

$$rev(true, true)$$
$$rev(x, y) \leftarrow rev(x, y)$$
$$rev(false, false) \leftarrow rev(x, z)$$

$\boxed{/x}$

Groundness Analysis (6)

$$rev(true, true)$$
$$rev(x, y) \leftarrow rev(x, y)$$
$$rev(false, false) \leftarrow rev(x, z)$$

$$T_P^1(\emptyset) = \{rev(true, true)\}$$
$$T_P^2(\emptyset) = \{rev(true, true), rev(false, false)\}$$
$$T_P^3(\emptyset) = \{rev(true, true), rev(false, false)\}$$

Groundness Analysis (7)

So after `rev(X,Y)`, `X` is ground iff `Y` is: $x \leftrightarrow y$

This analysis gives quite precise results

Many other such abstractions have been proposed

This is an area of active research

22 Finite State Machines

What is a program? What is an algorithm? How simple can we make them?

1. Introduction
2. Finite State Machines
3. FSMs in Prolog
4. Formally
5. NFAs
6. Power

22.1 Introduction

There are many programming languages, many programming paradigms

All have a few things in common; programs:

- Are finite
- Take input
- Produce output
- Make decisions based on current input and possibly earlier input

To facilitate recalling information about earlier inputs, most languages allow *state* of computation to be changed

22.2 **Finite State Machines**

Simplest mechanism we can define to do this always has a current state, which is one of a finite number of discrete states

Each input, together with current state, determines next state and any output

The state captures what is important about the history of the computation

This is a *finite state machine* (*FSM*) or *finite automaton*

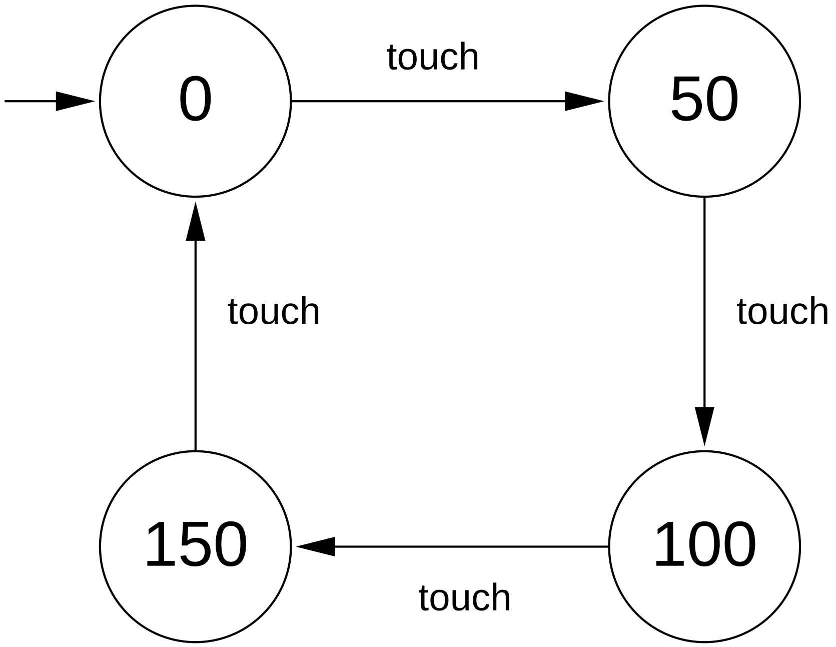
Finite State Machines (2)

A touch light is an example of a finite state machine

1. Light is initially off;
2. when touched, the 50 watt filament is lit;
3. when touched again, the 50 watt filament is extinguished and the 100 watt filament is lit;
4. when touched again, both filaments are lit
5. when touched again, both filaments are extinguished (go back to step 1)

22.2.1 State Diagrams

We can depict machine as a *state diagram* or *state transition diagram*

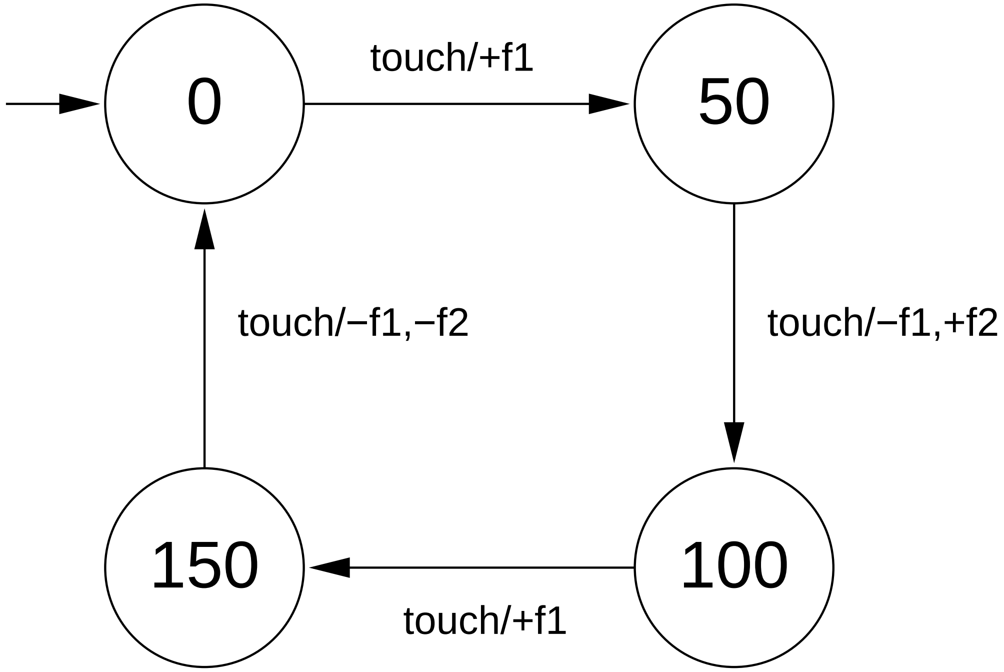


Circles represent states; arrows represent transitions between states in response to input

Initial state is indicated by sourceless arrow

22.2.2 Output

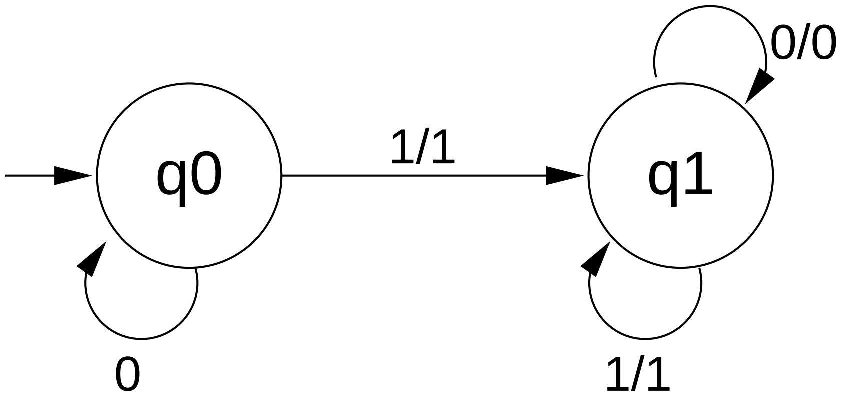
FSMs can produce output



Here $+f1$ means that filament 1 should be turned on, $-f2$ means filament 2 should be turned off, *etc*.

22.2.3 Example

This FSM copies its input of binary numbers to its output, stripping leading 0s:



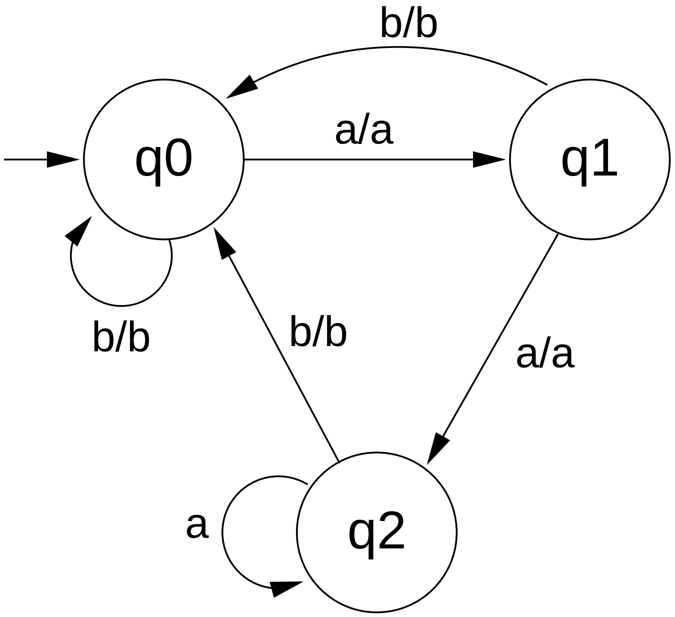
The transition from the initial state for 0 leaves it in that state without any output

States capture the history of the computation

Think of state $q0$ as meaning “no 1s seen yet” and $q1$ as “some 1s seen”

Exercise

What output does the following FSM produce with input $aabbaaabbb$?



More generally, describe what it does.

22.2.4 **Accepting States**

Many problems amount to simply determining if the input is “acceptable” — does it satisfy some criterion

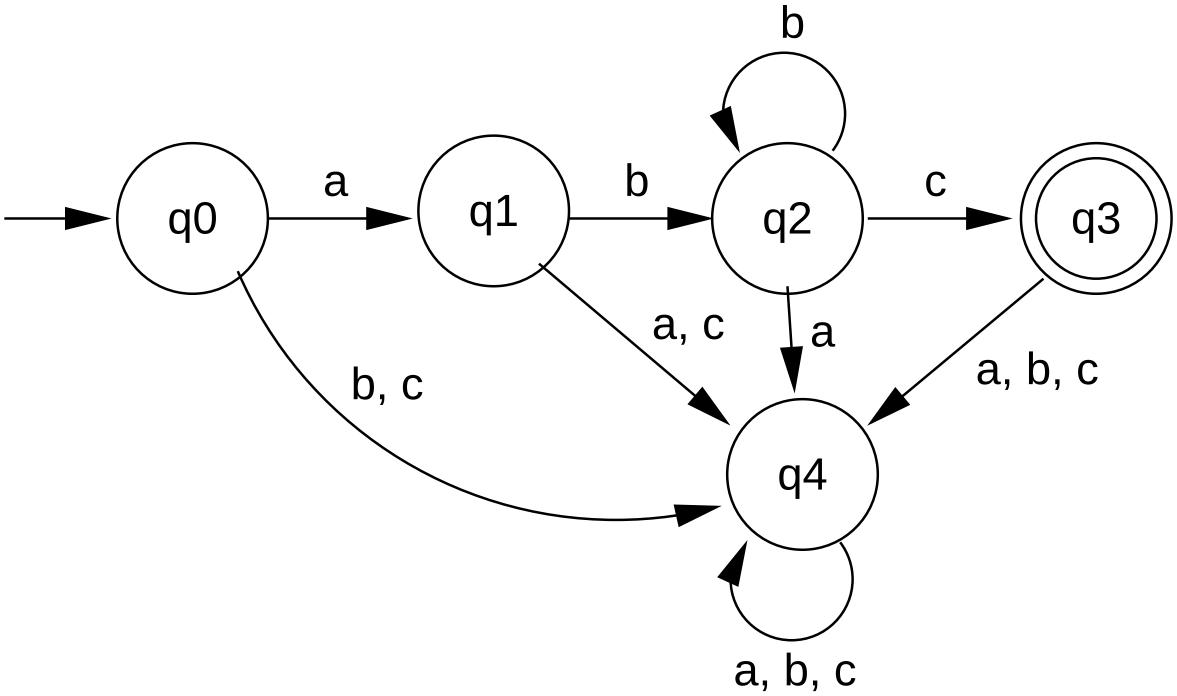
Double circle around state indicates *accepting state*

When input runs out, if machine is in an accepting state, it accepts that input string; otherwise it doesn’t

Can have as many accepting states as needed

∞ Accepting States (2)

Determine if the input is one `a` followed by one or more `b`s followed by a single `c`:



No output needed here, just *accept* or *reject*

Machine can do both: have output and accept or reject

22.3 FSMs in Prolog

To simulate FSMs in Prolog, we can specify state transition by a predicate `delta`$(State0, Input, State)$ which says that when in $State0$ we receive $Input$ as input, we transition into $State$

Also specify initial and accepting states

```
delta(q0, a, q1).  delta(q0, b, q4).  delta(q0, c, q4).
delta(q1, a, q4).  delta(q1, b, q2).  delta(q1, c, q4).
delta(q2, a, q4).  delta(q2, b, q2).  delta(q2, c, q3).
delta(q3, a, q4).  delta(q3, b, q4).  delta(q3, c, q4).
delta(q4, a, q4).  delta(q4, b, q4).  delta(q4, c, q4).

initial(q0).

accepting(q3).
```

FSMs in Prolog (2)

Code to similate an FSM is simple:

```
recognize(Input) :-
        initial(State0),
        run(Input, State0, State),
        accepting(State).

run([], State, State).
run([I|Is], State0, State) :-
        delta(State0, I, State1),
        run(Is, State1, State).
```

Exercise

Design an FSM to recognize strings containing an even number of as. The input can include both as and bs.

22.4 Formally

What does it take to define a FSM?

Q a set of states

Σ a set of input symbols, called the *alphabet*

δ specifies the state transitions

q_0 the initial state

F the set of accepting states

(assuming we do not need output).

So an FSM is a 5-tuple:

$$\langle Q, \Sigma, \delta, q_0, F \rangle$$

22.4.1 δ

State transition is specified by a *function* δ

Given any state and any input, δ yields the state to transition into

$$\delta : (Q \times \Sigma) \rightarrow Q$$

Can specify δ as a table (as we did for Prolog implementation):

state s	input i	$\delta(s,i)$
q0	a	q1
q0	b	q4
q0	c	q4
q1	a	q4
⋮	⋮	⋮

22.5 **NFAs**

What if state transition were specified by a *relation* rather than a *function*?

$$\delta \subseteq Q \times \Sigma \times Q$$

When δ is a function, only one next state given current state and input

When δ is a relation, may be more than one next state

Such a FSM is called a *Nondeterministic Finite Automaton* or *NFA*

If δ is a function: *Deterministic Finite Automaton* or *DFA*

An NFA accepts an input string if *some* selection of the state transitions specified by δ leads to an accepting state

22.5.1 λ transitions

NFAs also often allow λ transitions to be specified

A λ transition is a spontaneous transition — a transition in response to no input at all

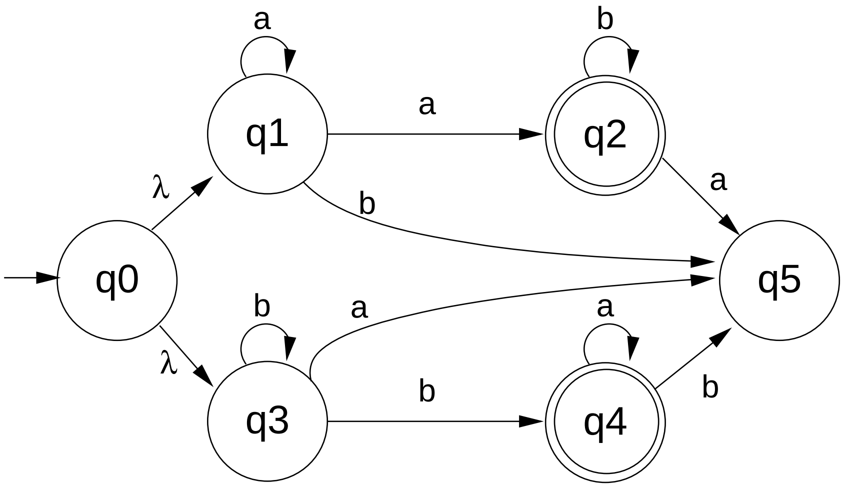
Specify this formally by pretending the input alphabet contains an extra symbol λ, and there can be as many λs as you like between any two real symbols

Then, formally δ becomes a relation

$$\delta \subseteq Q \times (\Sigma \cup \{\lambda\}) \times Q$$

22.5.2 Example

This NFA accepts all input that is either some `a`s optionally followed by some `b`s or some `b`s optionally followed by some `a`s:

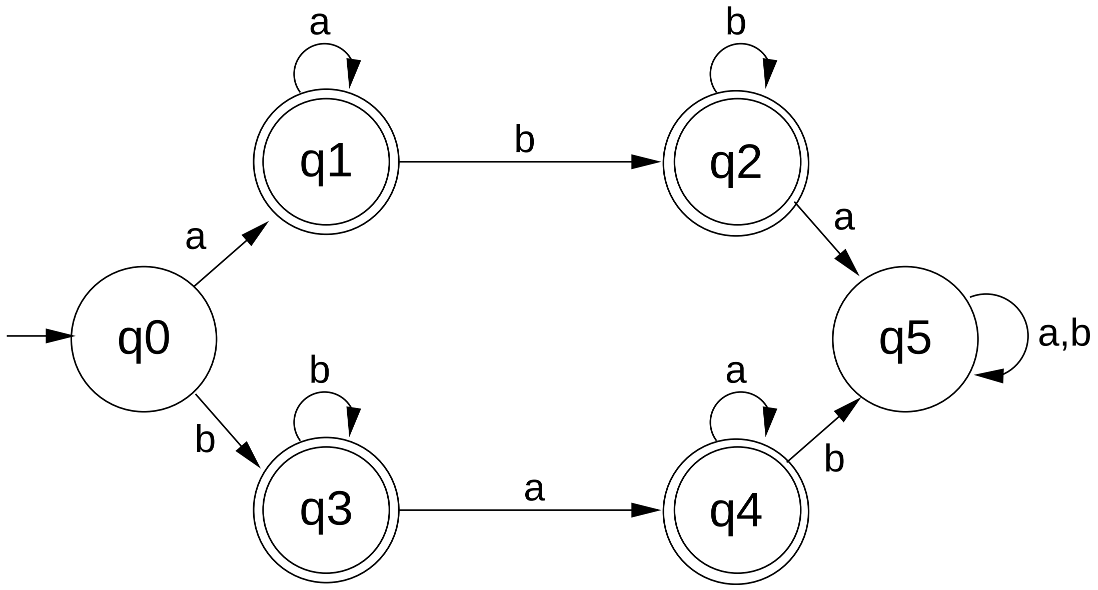


Example (2)

Important result: any NFA can be converted into an equivalent DFA

Intuition: state in new DFA corresponds to *set* of states in NFA

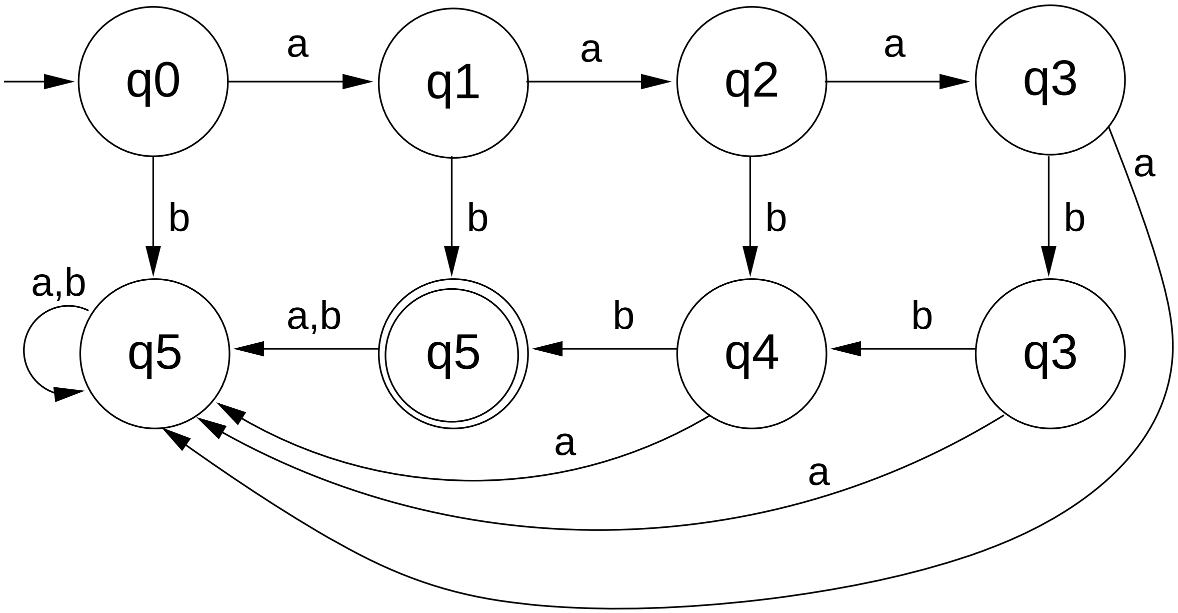
This is an equivalent DFA to the previous NFA:



22.6 **Power**

FSMs cannot do everything real programs can do

FSMs can recognize `ab` or `aabb` or `aaabbb`:



This regular structure can be extended to recognize $\mathtt{a}^n\mathtt{b}^n$ for n up to any fixed limit

But it's not possible to build a FSM to recognize $\mathtt{a}^n\mathtt{b}^n$ for *any* n

23 Turing Machines

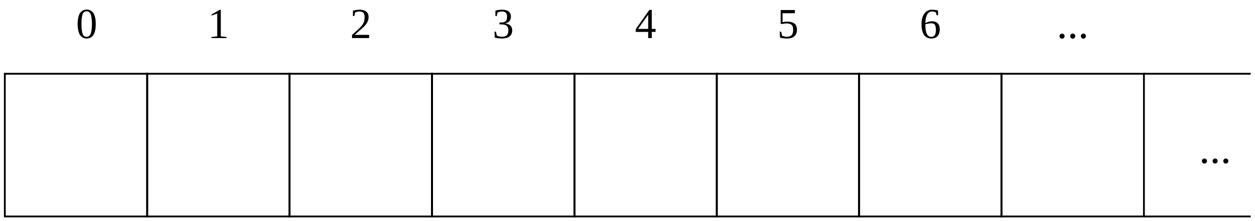
1. Turing Machines

2. Nondeterministic Turing Machines

23.1 Turing Machines

- Introduced by Alan Turing
- Abstract machine with many features of computer
- Designed *before* the stored program computer!
- No memory limit!
- An infinite tape, and a tape reader, that reads one symbol at a time.

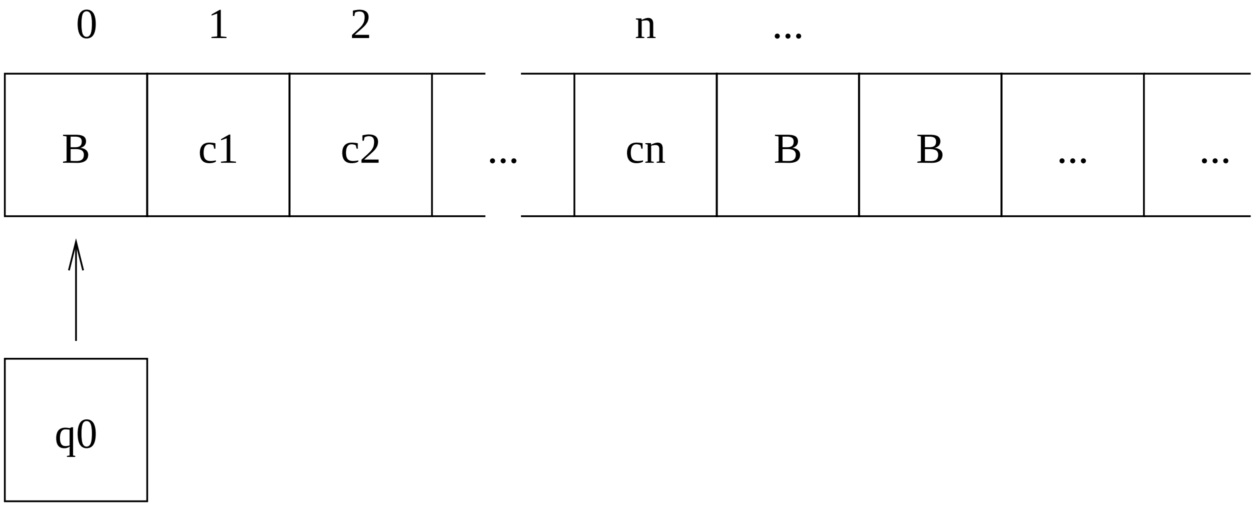
Turing Machines (2)



A quintuple $M = (Q, \Sigma, \Gamma, \delta, q0)$

- A finite set of states Q.
- A finite set Γ, the *tape alphabet*, which includes blank B
- A subset $\Sigma \subseteq \Gamma - \{B\}$ called the *input alphabet*
- A partial *transition function* $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
- A *start state* $q0 \in Q$

23.1.1 Turing Machines: Initial



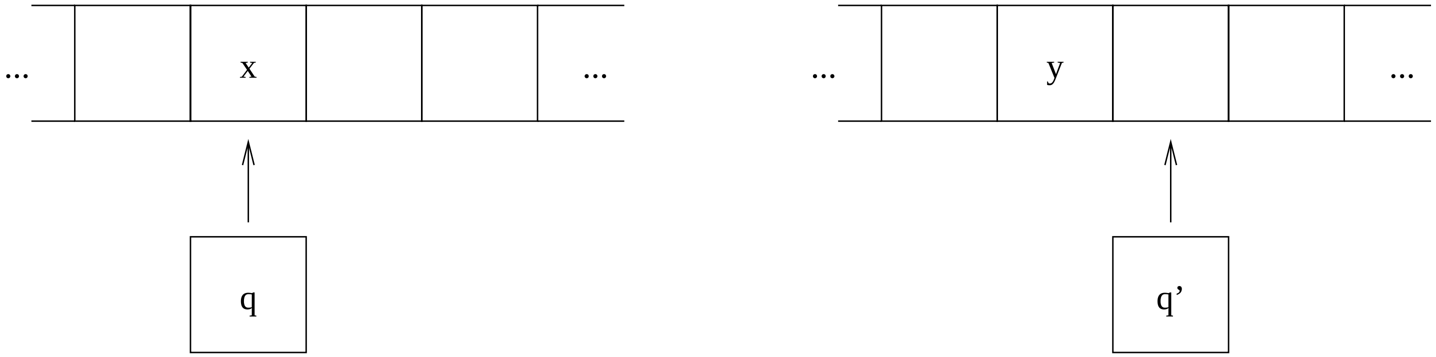
0	1	2		n	...			
B	c1	c2	...	cn	B	B	...	...

q0

Computation begins with

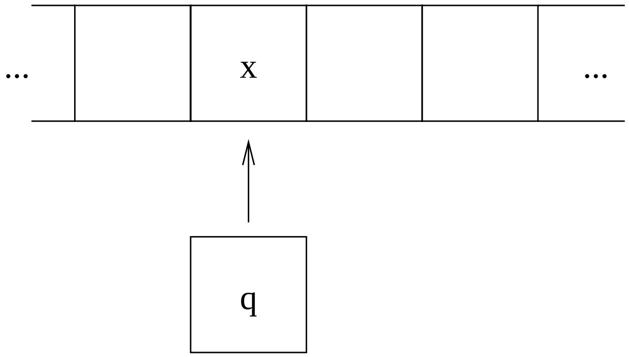
- An *input* string $s = c_1 \cdots c_n$ in Σ^* on the tape from position 1 to n
- The remaining tape blank (filled with B)
- Initial state $q0$

⌧ 23.1.2 Turing Machines: Transition



- Machine in state q reading tape symbol $x \in \Gamma$
- $\delta(q, x) = [q', y, d]$
- Transitions have three parts:
  - Change the state to q'
  - Write symbol y on square scanned by tape head
  - Move head left or right (depending on direction d)

⊠ 23.1.3 Turing Machines: Final



- Machine *halts* if in state q reading tape symbol $x \in \Gamma$ there is no $\delta(q, x) = [q', y, d]$

- Remember δ is a partial function.

- *Output* is the result remaining on the tape.

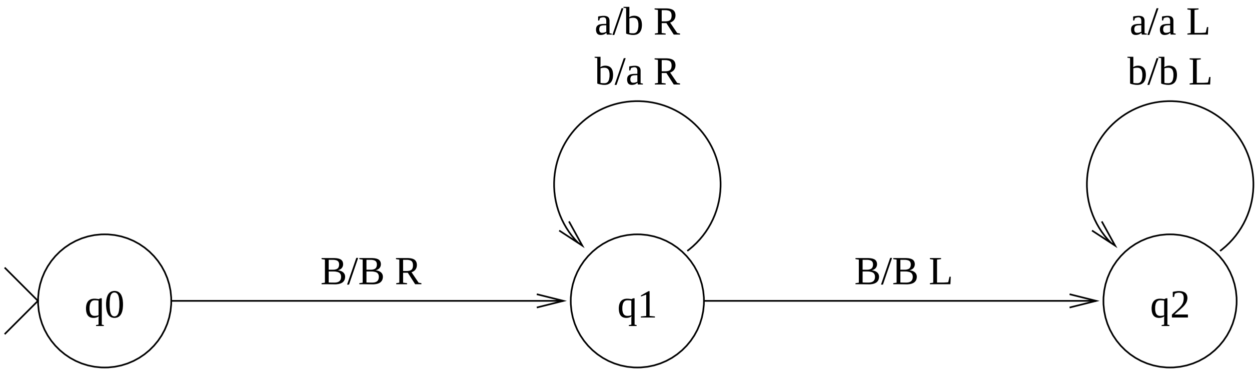
- Machine *halts abnormally* if the head moves off the left end of the tape.

23.1.4 Turing Machines: Example

Change all a's to b's and vice versa.

δ	B	a	b
$q0$	$q1, B, R$		
$q1$	$q2, B, L$	$q1, b, R$	$q1, a, R$
$q2$		$q2, a, L$	$q2, b, L$

Represented as a state machine

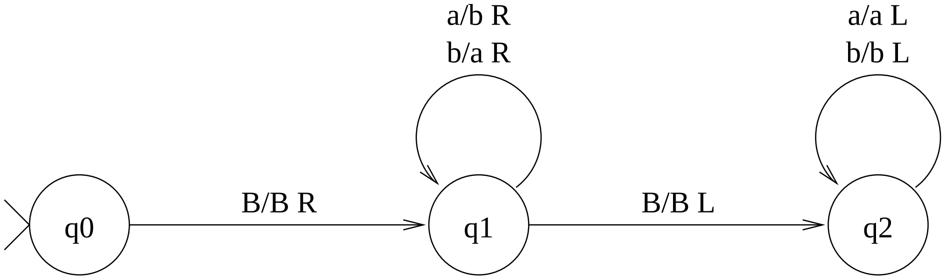


23.1.5 Turing Machines: Traces

- A *configuration* uq_ivB
  - represents the tape $uvBBB\dots$ (v includes rightmost non blank)
  - shows tape head over first symbol in v
  - shows machine state q_i
- notation $uq_ivB \vdash xq_jyB$ indicates a transition from configuration uq_ivB to xq_jyB.
- $uq_ivB \vdash^* xq_jyB$ means result of any finite number of steps

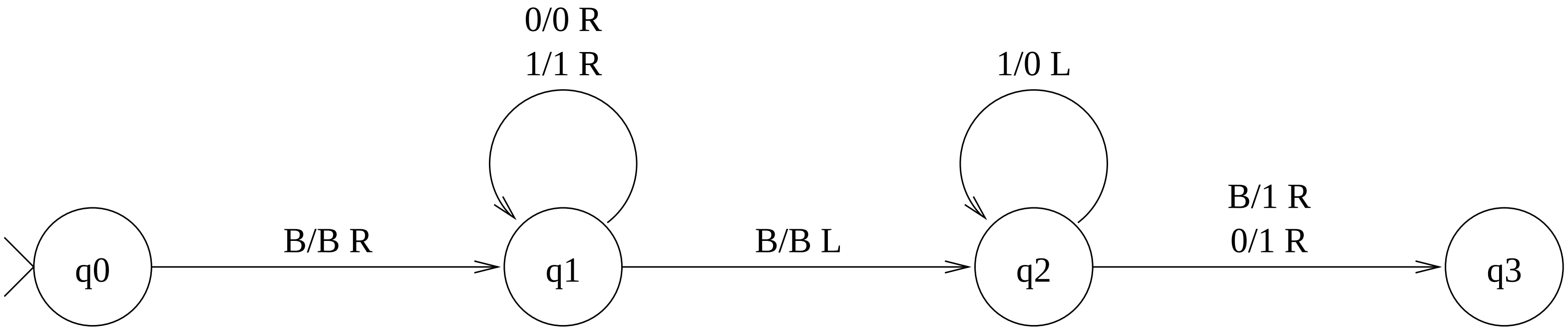
23.1.6 Turing Machines: Example Trace

$$\begin{aligned}
& q_0 BabaB \\
\vdash\ & Bq_1 abaB \\
\vdash\ & Bbq_1 baB \\
\vdash\ & Bbaq_1 aB \\
\vdash\ & Bbabq_1 B \\
\vdash\ & Bbaq_2 bB \\
\vdash\ & Bbq_2 abB \\
\vdash\ & Bq_2 babB \\
\vdash\ & q_2 BbabB
\end{aligned}$$



23.1.7 Turing Machines: Another Example

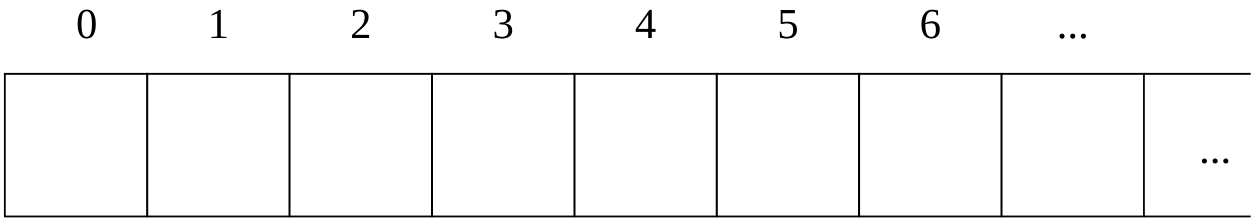
Machine to add 1 to a binary number



$$\begin{aligned} & q_0B001B \\ \vdash\ & Bq_1001B \\ \vdash\ & B0q_101B \\ \vdash\ & B00q_11B \\ \vdash\ & B001q_1B \\ \vdash\ & B00q_21B \\ \vdash\ & B0q_200B \\ \vdash\ & B01q_30B \end{aligned}$$

$$\begin{aligned} & q_0B11B \\ \vdash\ & Bq_111B \\ \vdash\ & B1q_11B \\ \vdash\ & B11q_1B \\ \vdash\ & B1q_21B \\ \vdash\ & Bq_210B \\ \vdash\ & q_2B00B \\ \vdash\ & 1q_300B \end{aligned}$$

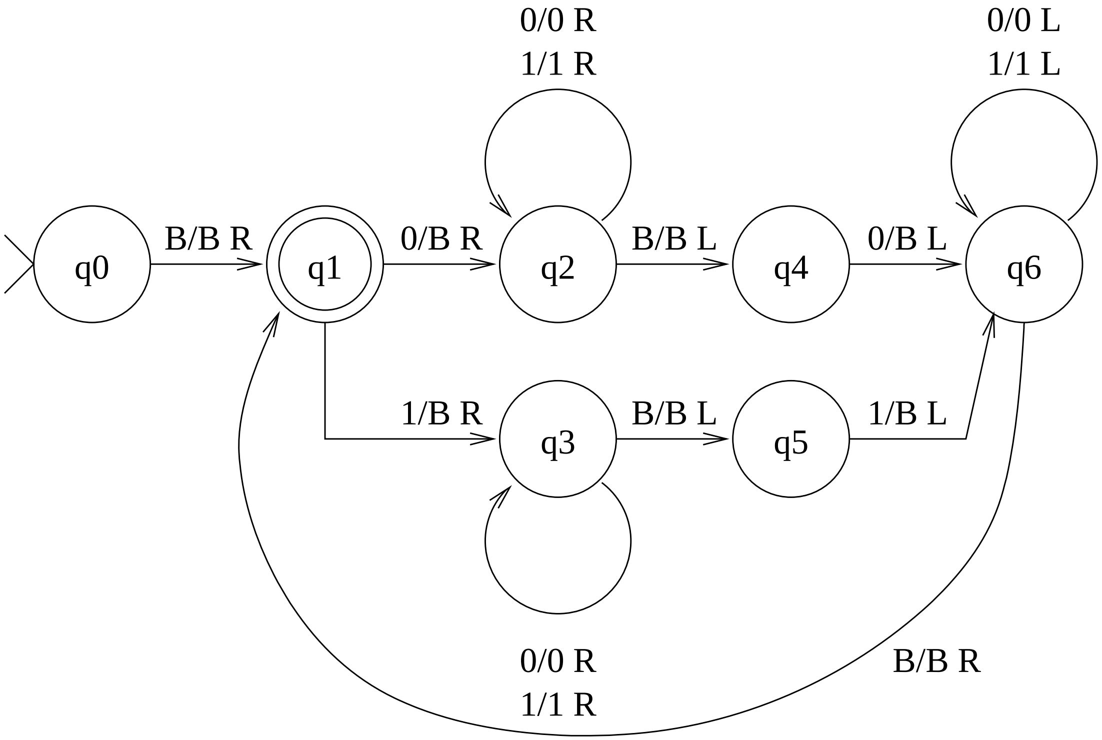
23.1.8 Turing Machines as Acceptors



A sextuple $M = (Q, \Sigma, \Gamma, \delta, q0, F)$

- F is the set of final states.
- If M halts in a state $q \in F$ then the input is accepted
- If M halts in a state $q \notin F$ the input is rejected.
- $L(M)$ is the set of strings accepted by M.

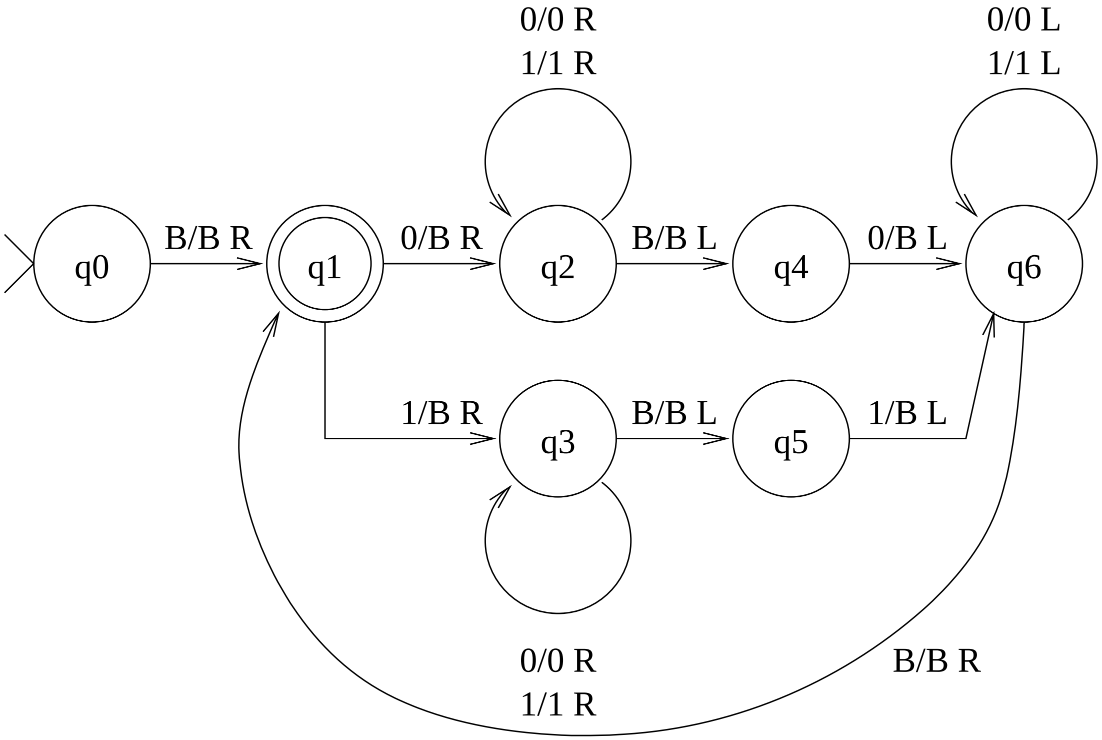
Turing Machines as Acceptors (2)



A machine that accepts a string which is an even length palindrome of 0's and 1's. e.g $00, 0110, 0001001000 \in L(M)$

⊠ Exercise

Modify the even length palindrome machine to accept odd length palindromes too e.g. 101



23.1.9 Turing Machines in Prolog

- given `delta(State,Sym,NewState,NewSym,LR)`
- (maybe) given `accepting(State)` for accepting states
- We can write a Turing machine simulator in Prolog
- So Prolog can do anything a Turing machine can

```
delta(q0, ’B’, q1, ’B’, right).
delta(q1, ’B’, q2, ’B’, left).
delta(q1, a, q1, b, right).
delta(q1, b, q1, a, right).
delta(q2, a, q2, a, left).
delta(q2, b, q2, b, left).

initial(q0).
```

Main Code

```
run(Input, Output, State) :-
        initial(State0),
        tape_list_position(Tape0, Input, 0),
        run(State0, Tape0, State, Tape),
        tape_list_position(Tape, Output, _).

recognize(Input) :-
        run(Input, _, State),
        accepting(State).
```

State Transition

```
run(State0, Tape0, State, Tape) :-
        (   step(State0, Tape0, State1, Tape1) ->
                run(State1, Tape1, State, Tape)
        ;   State = State0,
            Tape = Tape0
        ).

step(State0, Tape0, State, Tape) :-
        replace_tape_symbol(Tape0, Sym0, Tape1, Sym1),
        delta(State0, Sym0, State, Sym1, Direction),
        move_tape(Direction, Tape1, Tape).
```

Handling the Tape

```
tape_list_position(tape(Left, Right), List, Pos) :-
        length(Left, Pos),
        reverse(Left, Left1),
        append(Left1, Right, List).

replace_tape_symbol(tape(Left0,Right0), Sym0, tape(Left0,Right), Sym) :-
        replace_next_of_infinite_tape(Right0, Sym0, Right, Sym).

move_tape(left,  tape([Lsym|Left],Right), tape(Left,[Lsym|Right])).
move_tape(right, tape(Left,Right), tape([Rsym|Left],Right1)) :-
        replace_next_of_infinite_tape(Right, Rsym, [_|Right1], Rsym).

replace_next_of_infinite_tape([Sym0|Rest], Sym0, [Sym|Rest], Sym).
replace_next_of_infinite_tape([], ’B’, [Sym], Sym).
```

23.2 Nondeterministic Turing Machines

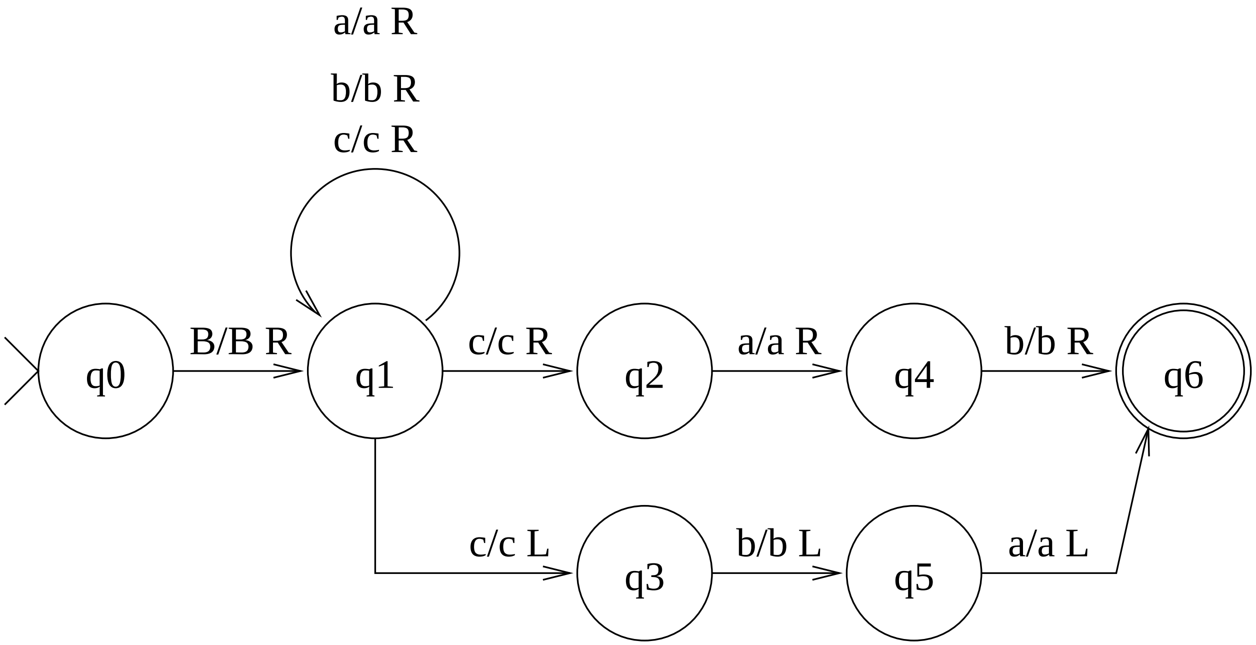
- δ maps to a subset of $Q \times \Gamma \times \{L, R\}$ rather than just one (or zero)

- Computation chooses which one to take?

- A non-deterministic Turing machine M accepts input s if some possible computation accepts s.

- **Important result:** Any language accepted by a non-deterministic Turing machine can be accepted by a deterministic Turing machine

Example



A machine which accepts strings of a's, b's and c's where there is a c followed by or preceded by ab.

Example (2)

Trace1	Trace2	Trace3
$q_0BacabB$	$q_0BacabB$	$q_0BacabB$
$\vdash Bq_1acabB$	$\vdash Bq_1acabB$	$\vdash Bq_1acabB$
$\vdash Baq_1cabB$	$\vdash Baq_1cabB$	$\vdash Baq_1cabB$
$\vdash Bacq_1abB$	$\vdash Bacq_2abB$	$\vdash Bq_3acabB$
$\vdash Bacaq_1bB$	$\vdash Bacaq_4bB$	
$\vdash Bacabq_1B$	$\vdash Bacabq_6B$	

First and third do not accept, but second does

That's good enough: the machine accepts

24 Undecidability

Are there some programs that cannot be written?

How many kinds of infinity are there?

24.1 Church-Turing Thesis

There is an effective procedure to solve a decision problem P if, and only if, there is a Turing machine that answers yes on inputs $p \in P$ and no for $p \notin P$.

- All computation can be performed by Turing machines?

- Not a *theorem* since we have no formal definition of an effective procedure!

- More like the *definition* of effective procedure!

Extended Church-Turing thesis *A decision problem P is partially solvable if, and only if, there is a Turing machine thats answers yes on inputs $p \in P$.*

24.1.1 Evidence for Church-Turing Thesis

- Every effective procedure in
  - Partial recursive functions
  - Lambda calculus
  - Predicate logic
  - Register machines

  can be *simulated* by Turing machine.
- All known effective procedures can be transformed to Turing machines

24.2 An undecidable problem

- The Halting Problem
  - Given an arbitrary Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0)$
  - An input $s \in \Sigma^*$
  - Does M halt on input s ?
- For particular Turing machines we may be able to determine that they halt.
- But not for *all possible machines*

24.2.1 Prove the Halting Problem Undecidable

- Suppose to the contrary
- There is a Turing machine H which accepts description of Turing machine M and input w and accepts M and w if machine M halts on input w.
- Assume $\Sigma = \{0, 1\}$ and $\Gamma = \{0, 1, B\}$, and states $\{q_0, \ldots, q_n\}$ and q_0 is the start state.
- Representing machine M (its δ function)

Symbol	0	1	B	q_0	q_1	$\cdots$	q_i	$\cdots$	L	R
Encoding	1	11	111	1	11	$\cdots$	1^{i+1}	$\cdots$	1	11

- $\delta(q_i, x) = [q_j, y, d]$ as string

$$\texttt{code}(q_i)0\texttt{code}(x)0\texttt{code}(q_j)0\texttt{code}(y)0\texttt{code}(d)$$

Prove the Halting Problem Undecidable (2)

- Represent δ function by **00** between two individual transitions.
- Beginning and end represented by *000*.
- **Example**

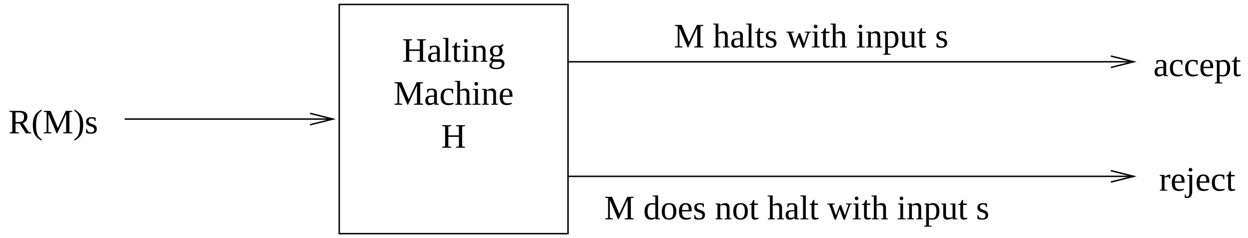
$\delta(q_0, B) = [q_1, B, R]$	101110110111011
$\delta(q_1, 0) = [q_0, 0, L]$	1101010101
$\delta(q_1, 1) = [q_2, 1, R]$	110110111011011
$\delta(q_2, 1) = [q_0, 1, L]$	1110110101101

- M is represented as $R(M)$ by

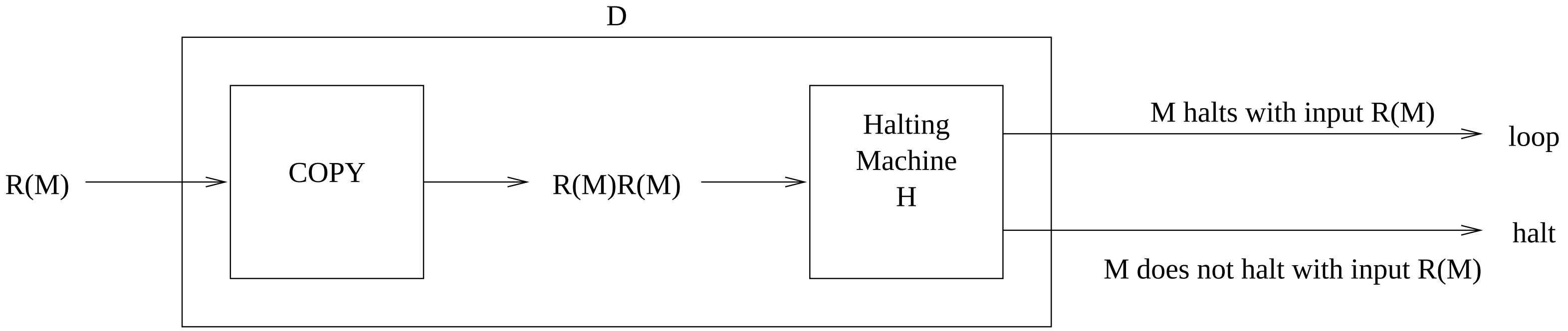
*000*101110110111011**00**1101010101**00**110110111011011**00**1110110101101*000*

Prove the Halting Problem Undecidable (3)

- Machine H takes a description of machine M, $R(M)$ and input s and accepts the input if M halts on s, and otherwise halts and does not accept.

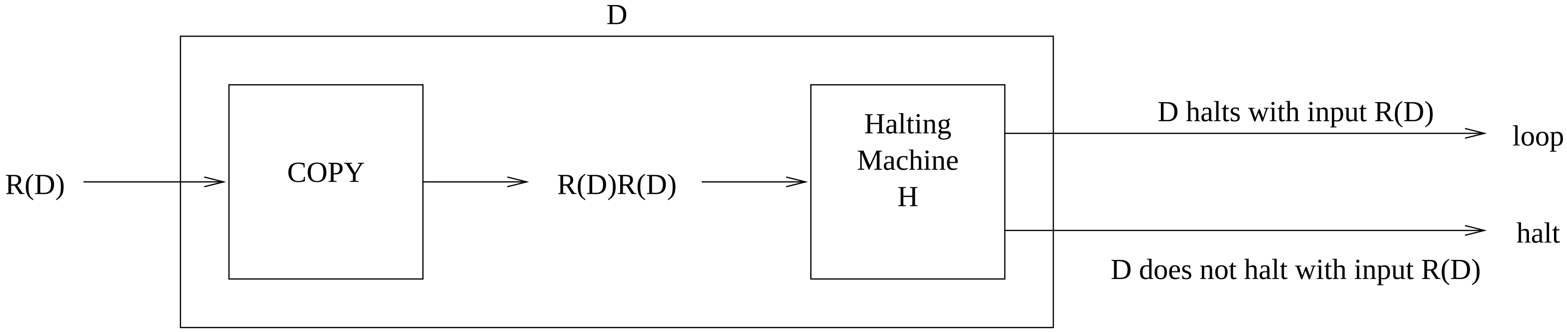


- Make a new machine D that copies its input and then applies H, and loops if H accepts, otherwise halts.



Prove the Halting Problem Undecidable (4)

- Execute D on the input $R(D)$



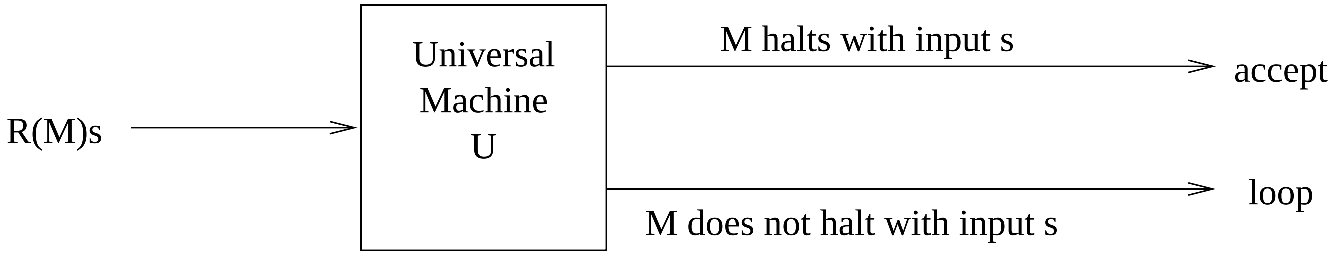
- **Contradiction** if D halts on input $R(D)$ then D loops!
- *Hence* The language L_H of strings $R(M)s$ such that machine M halts on input s is not *decidable*

∞ Exercise

Suppose Prolog had a builtin predicate `halt(Goal)` that succeeds if `Goal` would eventually terminate, and fails otherwise. Write a Prolog predicate `contra`/0 that calls `halt`/1 in such a way that `halt`/1 cannot possibly work.

24.3 Universal Machines

- We can build a Universal Turing Machine that simulates any Turing machine:



R(M)s

Universal Machine U

M halts with input s

accept

M does not halt with input s

loop

- The language L_H of strings $R(M)s$ such that machine M halts on input s is *semidecidable*

24.3.1 **A Universal Machine U (3 tape machine)**

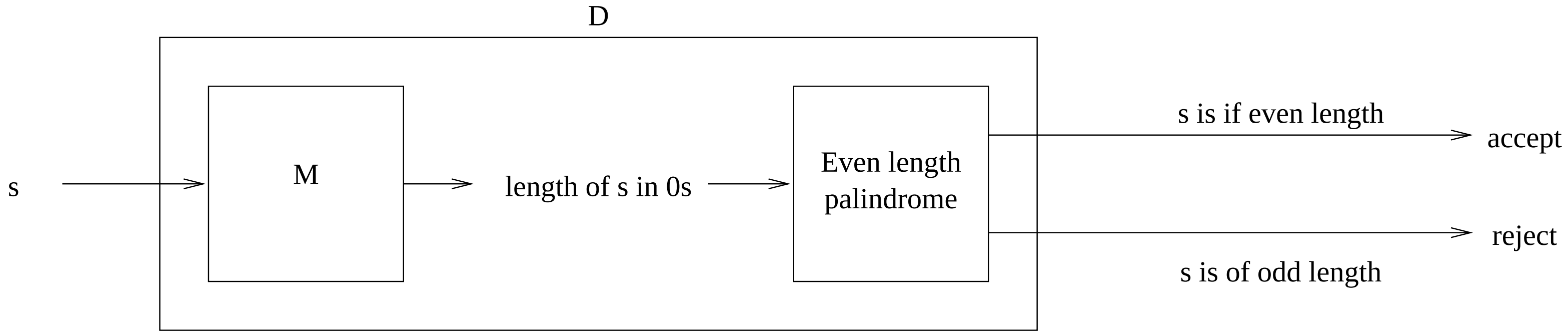
- If input is not of form $R(M)s$ move indefinitely right (loop)
- write string s on tape 3
- write 1 encoding state q_0 on tape 2
- simulate M on tape 3:
  - scan tape 1 for a transition for the state encoded on tape 2 q_i acting on the symbol x under the head on tape 3
  - if no transition, U halts
  - otherwise $\delta(q_i, x) = [q_j, y, d]$
    - write new state $\texttt{code}(q_j)$ on tape 2
    - write symbol y to tape 3
    - move tape head on tape 3 in direction d

24.4 Reducability

- A decision problem P is (Turing) *reducible* to a problem P' if:
  - there is a Turing machine M which maps input p to p' such that $p \in P$ iff $p' \in P'$

- *Idea* translate new problem P to old problem P'

- Example: we can reduce the problem of detemining if a string is even length to detecting if a string is an even length palindrome
  - M maps input replacing all 1s by 0s



D

s

M

length of s in 0s

Even length palindrome

s is if even length

accept

reject

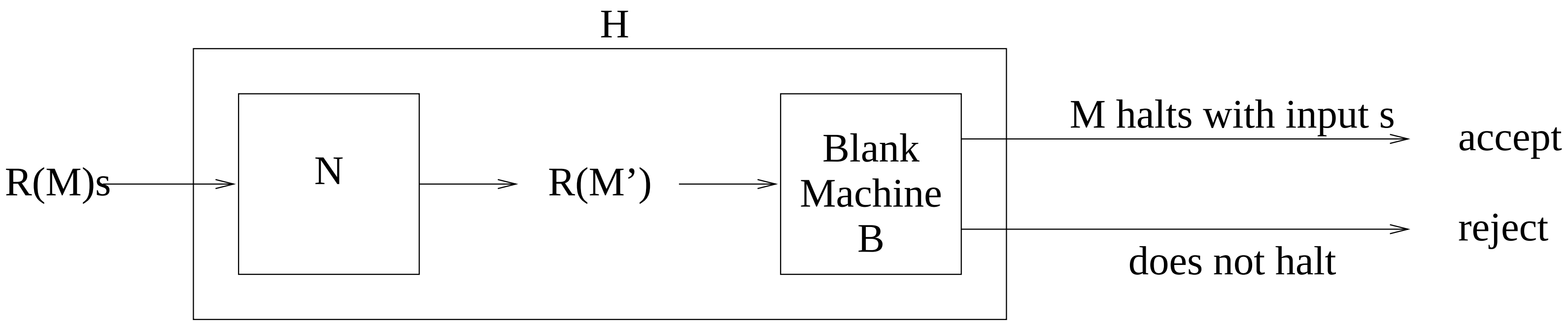
s is of odd length

24.4.1 **Another Undecidable Problem**

- The problem to determine if a Turing machine halts on blank input is undecidable
- Assume contrary, machine B answers problem
- create the machine H from B, that is reduce halting problem to blank tape problem
- Build machine N which takes input $R(M)s$ and creates output $R(M')$ where M' is a machine which:
  - takes a blank tape as input
  - writes s on tape
  - then executes machine M on tape

Another Undecidable Problem (2)

- The machine N provides a *reduction* of the halting problem to the halting on blank tape problem.

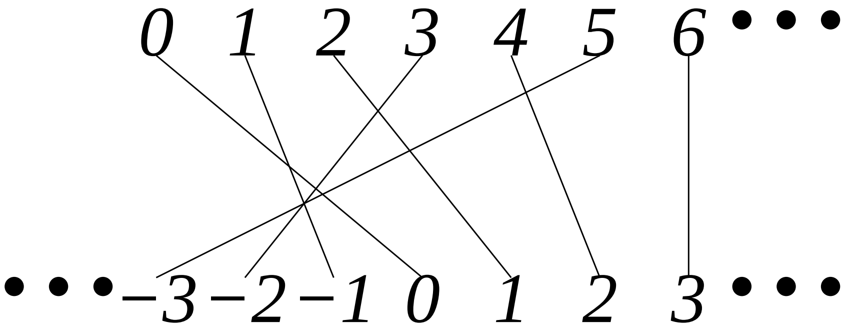


24.5 Countability

- There are infinitely many natural numbers $\{0, 1, 2, \ldots\}$
- Are there more integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ than natural numbers?
- Two sets have the same *cardinality* (size) if we can match each element of one set to a different element of the other, and vice versa
- A set that is either finite or has the same cardinality as the natural numbers is *countable*
- Are all sets countable?

∞ 24.5.1 **Integers**

- The integers are countable:



- Match nonnegative integers p with natural number $2p$
- Match negative integers n with natural number $-2n-1$

24.5.2 Rationals

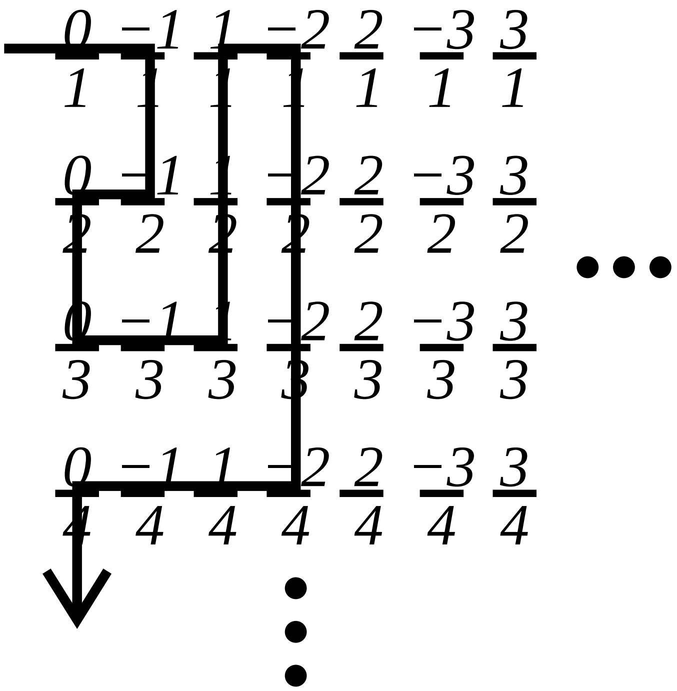
- Rational numbers are numbers that can be expressed as n/d for integers n and d

- The rational numbers are countable

All rationals appear in this:

$$\begin{array}{ccccccccc}
\frac{0}{1} & \frac{-1}{1} & \frac{1}{1} & \frac{-2}{1} & \frac{2}{1} & \frac{-3}{1} & \frac{3}{1} & & \\
\frac{0}{2} & \frac{-1}{2} & \frac{1}{2} & \frac{-2}{2} & \frac{2}{2} & \frac{-3}{2} & \frac{3}{2} & & \cdots \\
\frac{0}{3} & \frac{-1}{3} & \frac{1}{3} & \frac{-2}{3} & \frac{2}{3} & \frac{-3}{3} & \frac{3}{3} & & \\
\frac{0}{4} & \frac{-1}{4} & \frac{1}{4} & \frac{-2}{4} & \frac{2}{4} & \frac{-3}{4} & \frac{3}{4} & & \\
 & & & \vdots & & & & &
\end{array}$$

Can match rationals with naturals:



$$\begin{array}{ccccccccc}
\frac{0}{1} & \frac{-1}{1} & \frac{1}{1} & \frac{-2}{1} & \frac{2}{1} & \frac{-3}{1} & \frac{3}{1} & & \\
\frac{0}{2} & \frac{-1}{2} & \frac{1}{2} & \frac{-2}{2} & \frac{2}{2} & \frac{-3}{2} & \frac{3}{2} & & \cdots \\
\frac{0}{3} & \frac{-1}{3} & \frac{1}{3} & \frac{-2}{3} & \frac{2}{3} & \frac{-3}{3} & \frac{3}{3} & & \\
\frac{0}{4} & \frac{-1}{4} & \frac{1}{4} & \frac{-2}{4} & \frac{2}{4} & \frac{-3}{4} & \frac{3}{4} & & \\
 & & & \vdots & & & & &
\end{array}$$

24.5.3 **Reals**

- Real numbers include rationals and irrational numbers
- Irrationals include e, π, $\sqrt{2}$, *etc*.
- The real numbers are *not* countable
- Proof by Georg Cantor (diagonal argument)
- Assume reals are countable
- Then reals between 0 and 1 are countable

24.5.4 **Reals are Uncountable**

- Then we can arrange *all* reals in $[0,1]$ like this:

$$\begin{array}{ll} 0 & 0.D_0^0 D_1^0 D_2^0 D_3^0 \cdots \\ 1 & 0.D_1^1 D_1^1 D_2^1 D_3^1 \cdots \\ 2 & 0.D_2^2 D_1^2 D_2^2 D_3^2 \cdots \\ 3 & 0.D_3^3 D_1^3 D_2^3 D_3^3 \cdots \end{array} \quad \vdots$$

  Each D_m^n is the m^{th} decimal digit of the n^{th} real number

- Create a real number $X = 0.X_0X_1X_2X_3\cdots$ where each X_i is 2 if D_i^i is 1, or 1 otherwise

- X is not in our set of real numbers (it is different from each in at least one digit)

- Contradiction!

Exercise

Is the set of integer intervals, *e.g.* from 1 to 10, or from 0 to 99, or from -78 to -42, countable? Why or why not?

25 Complexity

1. What is Complexity?
2. Rates of Growth
3. Tractable Problems
4. NP Problems

25.1 What is Complexity?

- Performance of an *algorithm* = resources required for its execution
  - Time
  - Space
- Complexity of a *problem* = resources required for "best" possible algorithm
  - there may be no universal best algorithm
  - special cases may have better algorithms
- Examples
  - Sorting
  - Searching

What is Complexity? (2)

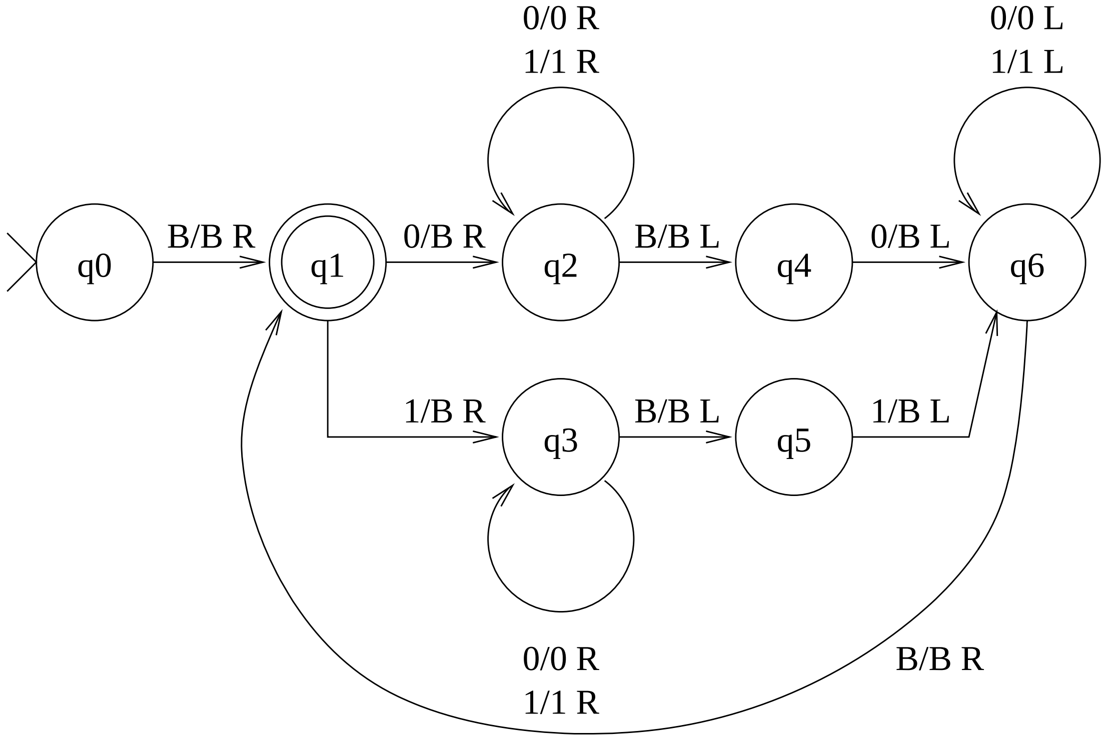
- Require a *formal* definition of computation
  - Turing Machine
  - Partial recursive functions
  - Lambda calculus
  - Predicate Logic
  - Register Machines
- Reason about “programs” in these formalisms
- Restrict to *decision* problems
  - A decision problem asks if input $p \in P$
  - It “decides” whether p is in the set P, or alternatively “defines” set P

What is Complexity? (3)

- *Time complexity* of a Turing Machine M:
  - The (worst case) number of transitions steps required to process input string of length n
- *Space complexity* of a Turing Machine:
  - The (worst case) maximum length of the tape required to process input string of length n
- Time complexity is always greater than space complexity (why?).
- Note this is complexity of an *algorithm* not complexity of a *problem*

25.1.1 Complexity Example

- Recall the Turing Machine that accepts even length palindromes



- Worst cases are: when we succeed: input $s = uu^R$, or when we fail at last step: input $s = u0u^R$ or $u1u^R$

Complexity Example (2)

$$\begin{array}{rrrr}
q_0BB & q_0B0B & q_0B00B & q_0B111B \\
\vdash Bq_1B & \vdash Bq_10B & \vdash Bq_100B & \vdash Bq_1111B \\
accept & \vdash BBq_2B & \vdash BBq_20B & \vdash BBq_311B \\
 & \vdash Bq_5BB & \vdash BB0q_2B & \vdash BB1q_31B \\
 & reject & \vdash BBq_40B & \vdash BB11q_3B \\
 & & \vdash Bq_6BBB & \vdash BB1q_51B \\
 & & \vdash BBq_1BB & \vdash BBq_61BB \\
 & & accept & \vdash Bq_6B1BB \\
 & & & \vdash BBq_11BB \\
 & & & \vdash BBBq_3BB \\
 & & & \vdash BBq_5BBB \\
 & & & reject
\end{array}$$

Complexity Example (3)

- Worst case steps

length of input	transition steps
0	1
1	3
2	6
3	10
$\vdots$	$\vdots$
n	$\Sigma_{i=1}^{n+1} = (n+2)(n+1)/2$

25.1.2 **Different execution machines**

- Some forms of execution machine offer better complexity
- Random Access Machine (RAM) (closer to modern computer)
  - constant time access to all memory locations by index

  -
    ```
    for (i = 1; i <= n/2; i++)
        if (s[i] != s[n+1-i]) return FALSE;
    return TRUE;
    ```
  - This version takes n steps.

Different execution machines (2)

- the complexity of a problem P is the complexity of the best algorithm for P (for a given class of execution machine)
  - $(n+2)(n+1)/2$ for Turing machine
  - n for RAM

25.2 Rates of Growth

n	0	5	10	25	50	100	1000	10000
$20n + 500$	500	600	700	1000	1500	2500	20500	200500
n^2	0	25	100	625	2500	10000	1000000	100000000
$n^2 + 2n + 5$	5	40	125	680	2605	10205	1002005	100020005
$n^2/(n^2 + 2n + 5)$	0	.625	.800	.919	.960	.980	.998	.9999

- Contribution of n^2 to $n^2 + 2n + 5$ is almost all as n increases
- **key**: only the fastest growing term is really important in determining rates of growth.

25.2.1 Big O Notation

- A function $f(n)$ is said to be *order* $g(n)$ if there is a positive constant c and positive integer n_0 such that

$$f(n) \le c \times g(n), \quad \forall n \ge n_0$$

- Write $f = O(g)$ if f is of order g.

- **Example** $f(n) = n^2 + 2n + 5,\ g(n) = n^2$ then $f = O(g)$

$$\begin{aligned} f(n) &= n^2 + 2n + 5 \\ &\le n^2 + 2n^2 + 5n^2 \text{ for } n \ge 1 \\ &= 8n^2 \\ &= 8 \times g(n) \end{aligned}$$

25.2.2 **Big O Hierarchy**

constant	$O(1)$	hash table lookup
logarithmic	$O(log(n))$	binary search tree lookup
linear	$O(n)$	list lookup
n log n	$O(nlog(n))$	sorting
quadratic	$O(n^2)$	
cubic	$O(n^3)$	solving linear equations
polynomial	$O(n^r)$ for some r	
exponential	$O(b^n)$ for some b	deciding propositional logic
factorial	$O(n!)$	

25.3 Tractable Problems

	$log_2(n)$	n	n^2	n^3	2^n	$n!$
5	2	5	25	125	32	120
10	3	10	100	1000	1024	3628800
20	4	20	400	8000	1048576	2.4×10^{18}
50	5	50	2500	125000	1.1×10^{15}	3.0×10^{64}
100	6	100	10000	1000000	1.2×10^{30}	$> 10^{157}$
200	7	200	40000	8000000	1.6×10^{60}	$> 10^{374}$

- A problem is *polynomial complexity* if it is $O(n^r)$ for some fixed r
- Problems which are not polynomial complexity are *intractable*
- Cannot be realistically solved for even medium instances
- But we **really** want to solve these problems!

25.4 Nondeterministic Polynomial Time

- A problem is *non-deterministic* polynomial time if there is a *non-deterministic* Turing machine M that solves the problem of size n in at most a polynomial number of transitions n^r for some fixed r.

- Alternatively problem is non-deterministic polynomial time if:
  - Guessing stage (non-deterministically guess an answer)
  - Checking stage (in polynomial time, check the guess)

25.4.1 **SAT**

Example: *propositional satisfiability*

Given a set of propositional clauses, guess a valuation for all the truth values. Evaluate F under this valuation. If $true$ answer yes! Given a propositional formula in clausal form

- Nondeterministically guess a truth assignment for each variable
- Check whether it makes the clauses true
  - For each clause, is there a true literal in the clause
- $O(n)$ where n is the size of the formula

25.4.2 P and NP

- A problem is in the class P if it is (deterministic) polynomial time
- A problem is in the class NP if it is non-deterministic polynomial time
- $P \subseteq NP$
- $P = NP$ one of the most important problems in Computer Science. Everyone believes that $P \neq NP$!

25.4.3 A problem in P

- Solution of propositional horn clauses is $O(n)$
- Algorithm
  - For each clause $x \leftarrow x_0, \ldots x_n$ set count $= n$
  - place all clauses with count 0 in queue
  - Remove first from queue, set lhs to $true$
  - Subtract 1 from the count of all clauses that contain lhs
  - Add new clauses with count 0 to queue.
  - continue until queue empty

25.4.4 P and NP and reducibility

- If p is (polynomial time) reducible to q and $q \in P$, then $p \in P$

- If p is (polynomial time) reducible to q and $q \in NP$, then $p \in NP$

- A problem p is NP-hard if every $q \in NP$ can be reduced to p.

- A problem p is NP-complete if p is NP-hard and $p \in NP$

- Propositional satisfiability is the most famous NP-complete problem.

25.4.5 **Proving NP completeness**

- If we can reduce problem p to q and p is NP-complete, then q is NP-hard.
- 3SAT.
  - A 3SAT problem is a SAT problem where each clause has at most 3 literals.
  - We can reduce SAT to 3SAT by adding new variables.
    - Long clause $l_1 \vee l_2 \vee l_3 \vee l_4$ becomes
    - $(l_1 \vee l_2 \vee y) \wedge (\neg y \vee l_3 \vee l_4)$
    - Any solution of the conjunction satisfies the original clause
    - Any solution of the original clause can be extended to satisfy the conjunction (choose the right value for y)

25.4.6 NP completeness

- Thousands of problems have been shown to be NP complete!
- edge cover, Hamiltonian path, graph coloring
- scheduling, rostering, register allocation
- Constraint programming concentrates on solving just these kind of problems

25.5 Summary

- Church Turing thesis: effective procedures
- Turing Machines
- Halting problem is *undecidable*
- Validity problem for predicate logic is *semidecidable*
- Satisfiability/Validity problem for propositional logic (SAT) is *NP-complete*
- Tractable and Intractable problems
- Solution of predicate Horn clauses is *undecidable*
- Solution of propositional Horn clauses is $O(n)$

26 Review

Cannot cover the whole semester in one lecture, but we'll try. . . .

1. Propositional Logic

2. Predicate Logic

3. Automata

26.1 Propositional Logic

A *proposition* is statement that can be either true or false, *e.g.* “interest rates are rising”

Assign propositional variable to simple propositions for convenience, *e.g.* R for “interest rates are rising”

Combine propositions using *connectives*:

$\neg$	(negation, not)
$\wedge$	(conjunction, and)
$\vee$	(disjunction, or)
$\rightarrow$	(material implication, if)
$\leftrightarrow$	(biimplication, iff)

Exercise: From English

$$\begin{aligned} R &= \text{Interest rates rising} \\ S &= \text{Share prices falling} \\ U &= \text{Unemployment rising} \end{aligned}$$

1. If interest rates rise, share prices fall
2. Interest rates rise if share prices fall (Careful!)
3. Interest rates are rising, but unemployment is not
4. Unemployment rises exactly when interest rates do
5. If unemployment rises, then share prices fall if interest rates rise

26.1.1 **Truth Tables**

Truth tables have a row for each combination of values of propositional variables, column for each formula of interest

A	B	$A \wedge B$	$A \vee B$	$\neg A$	$A \rightarrow B$	$A \leftrightarrow B$
T	T	T	T	F	T	T
T	F	F	T	F	F	F
F	T	F	T	T	T	F
F	F	F	F	T	T	T

If all truth table rows for a formula are true, formula is a *tautology*; if all are false, formula is *unsatisfiable*

A *model* of a formula is an *assignment* of truth values to variables that makes it true (truth table row)

26.1.2 **Arguments**

Argument is a sequence of propositions ending in a conclusion, *e.g.*:

> When interest rates rise, share prices fall. Interest rates are rising. Therefore, share prices will fall.

Conclusion is claimed to follow from premises

To check correctness, see if conjunction of premises imply conclusion, *e.g.* $((R \rightarrow S) \wedge R) \rightarrow S$

R	S	$R \rightarrow S$	$(R \rightarrow S) \wedge R$	$((R \rightarrow S) \wedge R) \rightarrow S$
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

All rows of truth table are true, so argument is *valid*

26.1.3 Better Way

Usually faster to demonstrate argument validity by contradiction: assume it is invalid, and show a contradiction

Simple technique can help with the bookkeeping: write out the formula and write T or F under each variable and connective as you determine them, propagating variable values

$$((R \rightarrow S) \land R) \rightarrow S$$

T X F T T F F

So the initial assumption (that the argument was invalid) was wrong; the argument is valid

26.1.4 Axiomatic System (AL)

Axiom schemas — allow any wff for A, B, and C

Ax1 $A \rightarrow (B \rightarrow A)$

Ax2 $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

Ax3 $(\neg A \rightarrow \neg B) \rightarrow (B \rightarrow A)$

One rule of deduction, or *inference rule*: *modus ponens*: from A and $A \rightarrow B$ infer B

AL system is:

1. *Consistent*: $\vdash A$ iff $\nvdash \neg A$
2. *Sound*: $\vdash A$ implies $\models A$
3. *Complete*: $\models A$ implies $\vdash A$
4. *Decidable*: there is an algorithm to always decide if $\vdash A$

Exercise: Complete the proof

$$\{A \to B, B \to C\} \vdash A \to C$$

1	$B \to C$	Hyp
2	$(B \to C) \to (A \to (B \to C))$	Ax1
3	**(a)**	MP 1,2
4	$(A \to (B \to C)) \to ((A \to B) \to (A \to C))$	Ax2
5	$(A \to B) \to (A \to C)$	**(b)**
6	**(c)**	Hyp
7	$A \to C$	MP 5,6

(a)

(b)

(c)

Exercise: Normal Forms

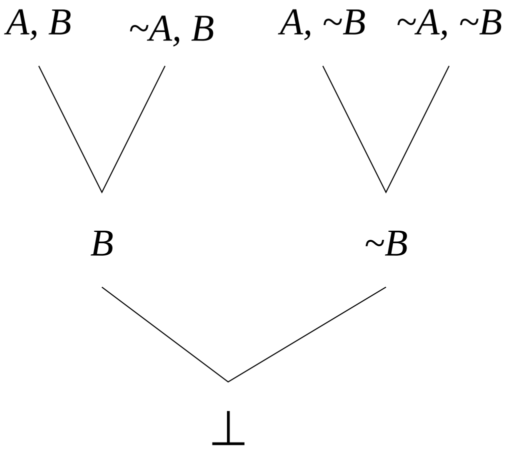
- A *literal* is a basic proposition or the negation of one
- *Conjunctive normal form (CNF)*: conjunction of 1 or more disjunctions of 1 or more literals
- *Disjunctive normal form (DNF):* the converse
- Exercise: which normal form are these?
  $(A \wedge \neg B) \vee (C \wedge D)$
  $(A \wedge B)$
  $\neg(A \vee B)$
  $(\neg A \vee B) \wedge C$
  $(\neg A \vee B) \vee C$
- Convert to normal form: replace $\leftrightarrow$ and $\rightarrow$ using only $\wedge, \vee, \neg$; push $\neg$ in; remove double $\neg$; distribute $\vee$ over $\wedge$ and $\wedge$ over $\vee$

26.1.5 Resolution

- Clausal form: CNF written as set of sets
- Complementary literals: *e.g.* A and $\neg A$
- Resolvent of 2 clauses with complementary literals is union of 2 clauses with complementary literals removed
- *E.g.* resolvent of $\{A, \neg B, C\}$ and $\{A, B, \neg D\}$ is $\{A, C, \neg D\}$
- Horn clause has at most 1 positive literal

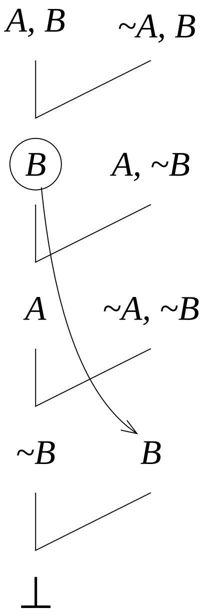
Resolution (2)

Resolution



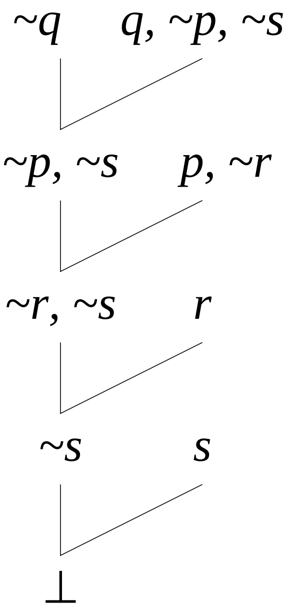
Resolve any 2 clauses

Linear Resolution



Resolve result of last step with any other clause

Linear Input Resolution



Resolve result of last step with clause from program

All are sound. All are complete, except L.I. Resolution: complete only for Horn clauses

26.2 **Predicate Logic**

- Variables stand for *anything*
- Predicates describe properties of individuals or relationships among individuals — “parameterized propositions”
- *Interpretation* limits *domain of discourse* and gives meaning of predicates, functions, and constants
- Truth value of formula usually depends on interpretation
- Formula A is *valid*, written $\models A$, if A is true in *every* interpretation; *satisfiable* if true in some interpretation

26.2.1 Quantifiers

- $\forall x F$ means F is true for *every* x in the domain of discourse
- $\exists x F$ means F is true for *some* x in the domain
- $\forall x \exists y F$ *not* the same as $\exists y \forall x F$
- $\forall x \forall y F$ *is* the same as $\forall y \forall x F$
- $\forall x \neg F$ *not* the same as $\neg \forall x F$
- $\forall x \neg F$ *is* the same as $\neg \exists x F$
- Quantifier applies to single immediately following formula, *e.g.* in $\forall x P(x) \wedge Q(x)$, the quantifier only applies to $P(x)$

26.2.2 From English

- Translating from English to predicate logic, be careful of order of quantifiers, and negation, parentheses

- Translate “every P does Q” to $\forall x(P(x) \rightarrow Q(x))$

- Translate “some Ps do Q” to $\exists x(P(x) \wedge Q(x))$

- Note usually use $\rightarrow$ with $\forall$ and $\wedge$ with $\exists$

Exercise: To and from English

$B(x)$	x is a book
$P(x)$	x is a person
$O(x,y)$	x is older than y
$A(x,y)$	x is the author of y
$R(x,y)$	x is a reader of y

- Everyone is a book
- Everyone reads books
- $\forall x(\exists y A(x,y) \to \exists y(B(y) \land R(x,y)))$
- Every author is older than the books they write

26.2.3 Normal Forms

- *Prenex normal form* is like conjunctive normal form with all quantifiers at the left
- Convert to PNF like CNF, but $\neg\forall x F \Longrightarrow \exists x \neg F$ and $\neg\exists x F \Longrightarrow \forall x \neg F$
- When multiple quantifiers use same variable, rename all but one to use new variables
- Once $\leftrightarrow$ and $\rightarrow$ are gone and all $\neg$ are innermost, just move all quantifiers in order to front of formula

26.2.4 Clausal Form

- Clausal form is PNF with no existential quantifiers
- Skolemisation eliminates existential quantifiers
- Replace existentially quantified variables by *new* function of all variables *universally* quantified in outer scope
- Function of no variables is a constant
- *E.g.*, $\forall x \forall y \exists z P(x, y, z) \Longrightarrow \forall x \forall y P(x, y, f(x, y))$ and $\exists x \forall y P(x, y) \Longrightarrow \forall y P(c, y)$
- Clausal form usually written as set of sets

26.2.5 Resolution

- *Herbrand interpretation*: interpretation mapping every constant, function and predicate symbol to that symbol (identity interpretation)
- *Herbrand Universe*: set of all ground terms that can be created from the function and constant symbols of the program
- *Herbrand:* Set of clauses is unsatisfiable iff a finite subset of ground instances is
- *Robinson:* Resolution is sound and complete for predicate logic
- Resolution same as for propositional logic, except that *unification* is used to match complementary literals

26.2.6 Unification

- *Substitution* specifies terms to substitute for some variables in a term or atom; other variables are left unchanged

- *E.g.*, given substitution $\sigma = \{a/x, f(x)/y\}$ and term $T = g(x, y, z)$, $T\sigma = g(a, f(x), z)$ (note new x not replaced by a)

- Simplified algorithm to unify a and b:

  1. If a is a variable not appearing anywhere in b, add a substitution b/a
  2. If b is a variable not appearing anywhere in a, add a substitution a/b
  3. If both are terms with the same function and arity, recursively unify the arguments pairwise
  4. otherwise, fail

Exercise: Resolution Proof

Use resolution to prove the following set of clauses is unsatisfiable:

$$\left\{\{A(c,v,v)\},\{A(f(u,x),y,f(u,z)),\neg A(x,y,z)\},\{\neg A(f(a,c),f(a,c),v)\}\right\}$$

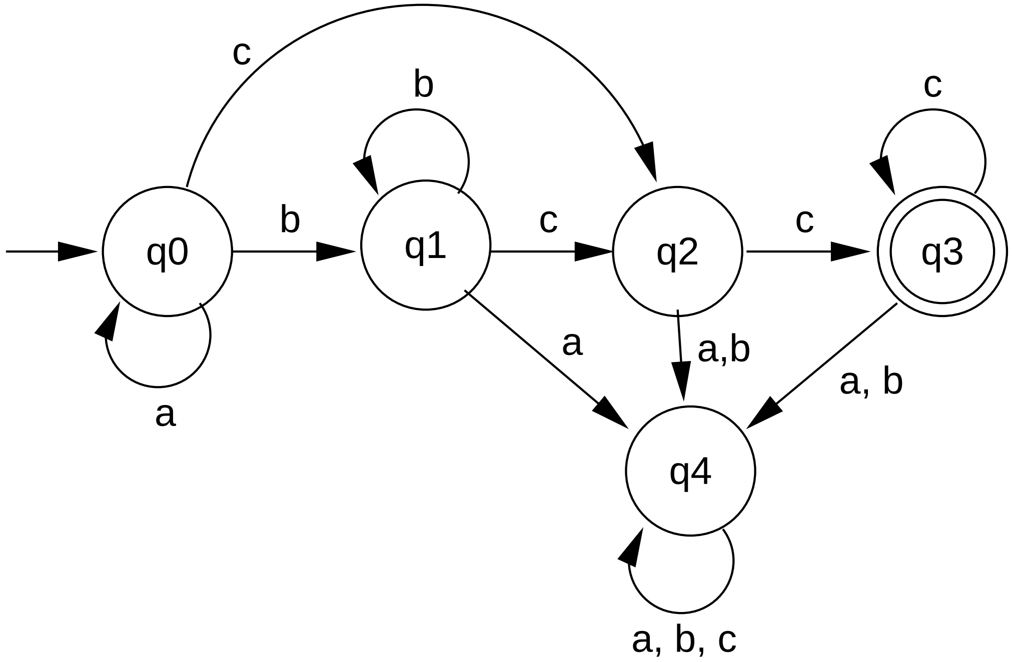
26.3 **Automata**

- Finite Automata (Finite State Machine) handles a sequence of input symbols from *input alphabet* set Σ
- Machine's reaction to input symbol depends on current *state*, an element of Q
- Reaction to input specified by function $\delta : Q \times \Sigma \rightarrow Q$
- Machine must also specify *initial state* q_0
- Machine may act as a *recognizer*; then must also specify a set of states F; when input runs out, if state of machine $\in F$ then input is *accepted* by machine
- Machine may produce output; then δ may also specify an output symbol to produce for given state and input
- *Nondeterministic Finite Automaton (NFA)* is FA where δ is a relation rather than a function; Any NFA can be rewritten as a DFA

Exercise

Often drawn as diagram with circles representing states and arcs labeled by input symbols representing transitions (δ function)

What input does this FSM accept?



26.3.1 **Pushdown Automata**

- Pushdown automata are like FAs augmented with unlimited stack
- Each transition may consume symbol from input, push symbol, or pop (and check) symbol
- Extend FA with set Γ of symbols allowed on stack; δ maps state, input symbol, and stack top to new state and stack top
- Usually interested in *nondeterministic* pushdown automata: δ is a relation
- *Cannot* rewrite every NPDA to equivalent DPDA
- Pushdown automata can recognize languages FAs cannot, eg $a^n b^n$
- NPDAs can recognize *regular languages* (*cf* regular expressions, as used in many Unix tools)

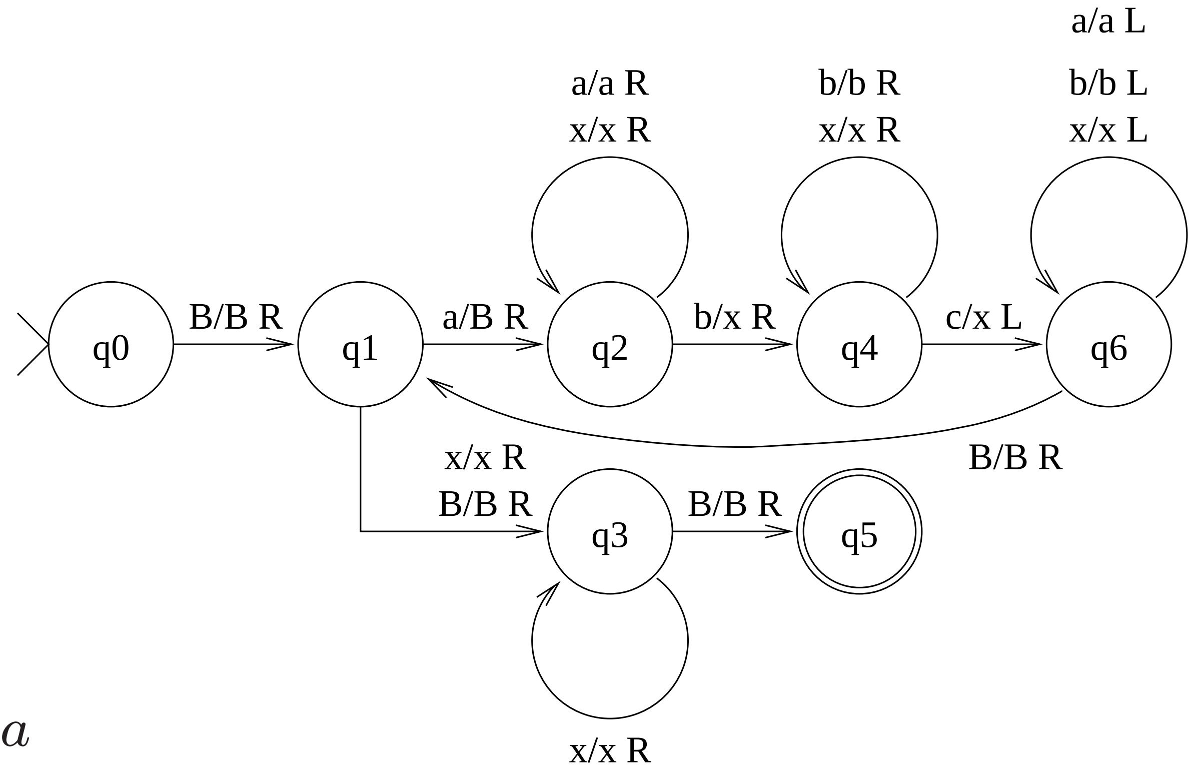
26.3.2 Turing Machines

- Turing machines replace the stack with an infinite *tape*
- Input is provided on the tape, output is written there, and tape can be used for intermediate results
- Tape always allows special blank symbol B; after finite input, tape is all Bs
- Γ is *tape alphabet*; $\Sigma \cup \{B\} \subseteq \Gamma$
- At each transition, tape is moved either left (L) or right (R) one symbol
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
- δ is a *partial* function; when δ has no value for current state and tape symbol, machine halts

Exercise

Which of these inputs does the following machine accept?

More generally, what inputs will the machine accept?



- *abaca*
- *abc*
- *aabbcccc*
- *aaabbbccc*

26.3.3 **Undecidability**

- *Church-Turing Thesis*: there is an *effective procedure* to decide a problem iff a Turing machine can do so
- Cannot be proved, but has stood the test of time
- Halting problem (will a turing machine M ever halt if given input s) is *undecidable* — can never be implemented on any TM
- Halting problem is *semidecidable* — can built TM SH that will accept when M will halt for input s; if it won't, SH will not terminate
- Problem P is *reducible* to Q if input for P can be rewritten to input for Q
- If P is reducible to Q and Q is decidable, so is P; if P is undecidable, so is Q
- Satisfiability of *predicate* logic formulae is undecidable

26.3.4 Countability

- Two set are same *cardinality* if some function maps every element of one to a unique element of the other
- A set is countable if it has same cardinality as the natural numbers
- Integers are countable: *e.g.*, map even naturals n to $n/2$ and odd to $-(n+1)/2$
- Rational numbers are countable
- Real numbers are not: Cantor diagonal argument

26.3.5 Complexity

- *Complexity* measures worst case time or space usage of algorithm depending on size of input
- Complexity of *problem* measures complexity of *best possible* algorithm
- Algorithm of "order $f(n)$" $O(f(n))$ means $f(n)$ dominates complexity of algorithm
- Complexity depends on computation model: *e.g.*, RAM can test palindrome in $O(n)$, TM $O(n^2)$
- Problem is in NP if some Nondeterministic TM can solve problem in polynomial time ($O(n^c)$ for some c)
- Problem is NP-hard if every problem in NP can be reduced to it in polynomial time; NP-complete if in NP and NP-hard
- Many problems are NP-complete, including SAT:

determining satisfiability of *propositional* logic formulae