Lecture 1 – Artificial Intelligence

Webpage:  http://www.aass.oru.se/~mbl/ai

Lecturer: Mathias Broxvall
phone: 30 1438
email: mathias.broxvall@tech.oru.se
room: T2232

Course assistant: Marco Gritti
email: marco.gritti@tech.oru.se
room: T1212
Literature

Course book:

Alison Cawsey, The essence of Artificial Intelligence (Akademibokhandeln).

Recommended reading:

(Lisp)
2. Steele, *Common Lisp The Language*, 1990

(AI)
Preliminary course contents

week 44: Introduction to course and AI.
week 45: LISP programming: Recursion, higher order functions, lambda expressions etc.
week 46: General problem solving, search, expert systems.
week 47: Planning, the STRIPS planner, Agents, Structured knowledge representation.
week 48: Propositional/Predicate Logic.
week 49: Natural language processing, Machine learning.
week 50: Machine learning. Preparation for exam. AI research at AASS.

Lab 1: LISP, week 44 – 45
Lab 2: Search, week 46
Lab 3: Expert Systems, week 47
Lab 4: Agents, week 48 - 50
What is Artificial Intelligence?

**Artificial**: Not natural or real, made by the art of man

**Intelligence**: 1) The power of perceiving, learning, understanding and knowing; mental ability. 2) News, information.

(\textit{Oxford Advanced Learners Dictionary of Current English})

Artificial Intelligence is the design and study of computer programs that behave intelligently. (Dean, Allen.)

Artificial Intelligence is the study of how to make computers do things at which, at the moment, people do better. (Rich)

Artificial Intelligence is the study of mental faculties through the use of computational models. (Charmiak, McDermont)

The branch of computer science that is concerned with the automation of intelligent behavior. (Luger, Stubblefield)

Develop Intelligent programs that can handle unexpected situations and allow for a better communication. (Sandewall)

... Artificial Intelligence is the art of making computers work the way they do in the movies. (Unknown)
Philosophical ideas about AI

Can machines think?

The Turing test:
First described by Alan Turing, 1950.
If an impartial judge communicating with a human and/or with a computer both attempting to pass as a human and he cannot see the difference between them, then the machine passes the turing test.

The Chinese room:

The Loebner price competition:
The Loebner Prize is an annual competition that awards prizes to the Chatterbot considered the most humanlike for that year. The format of the competition is much like that of a standard Turing test.

(wikipedia)
AI as a broad field involving research in numerous other areas.
Weak AI vs. Strong AI

Weak AI: Machines can be made to act as if they were intelligent.
Ex: Eliza, A.L.I.C.E, expertsystems etc.

Strong AI: Machines that act intelligently have real, conscious minds.
- human-like AI
  Ex: the movie A.I. etc.
- non-human-like
  Ex: HAL in the movie 2001
Symbol systems vs. Connectionistic AI

**Symbol systems / classical AI:**
- Symbolic manipulation of abstract concepts.
- Example: Logic, expert systems, planning, etc.

**Connectionistic AI:**
- Inspired by eg. the brain, evolution, etc.
- Example: Neural networks, genetic algorithms, etc.
Some of the topics in Artificial Intelligence

Automated Reasoning
Agents
Combinatorial search
Computer vision
Expert system
Genetic programming / Genetic algorithms
Knowledge representation
Machine learning
Machine planning
Neural networks
Natural language processing
Non-monotonic Reasoning
Program synthesis
Planning
Robotics
Artificial life
Distributed artificial intelligence
Some systems using Artificial Intelligence

Language Translation
Supervisory Systems
Automated personal assistants
Intelligent information retrieval
Robots
Autonomous vehicles
Expert systems
Diagnosis

Computer games

Practical example – playing games
Tic-Tac-Toe

- 2 players: O - X
- take turns in filling a square
- first move cannot be square 5
- winner = three in a row (vertical, horizontal, diagonal)
"Solution" to the Tic-Tac-Toe game

Just create a table with the best winning move for every possible game state!

Tic-Tac-Toe - program 1

- Takes a lot of space
- A lot of work specifying all the 19 683 entries
- Errors?
- No generalisation possible
Another ”solution”

Create an *ad hoc* algorithm for playing.

Blank = 2, X = 3, O = 5

Make2: return center if it is empty, otherwise a non empty non corner.

Posswin: return the square that constitutes a winning move. (if product is 18 then X can win, if product is 50 then 0 can win).
Tic-Tac-Toe - program 2

1. Go(1)

2. if Board[5] = ' ' then Go(5) else Go(1)

3. if Board[9] = ' ' then Go(9) else Go(3)

4. if Posswin(x) then Go(Posswin(x)) else Go(Make2)

5. if Posswin(x) then Go(Posswin(x)) elseif Posswin(o) then Go(Posswin(o))
   elseif Board[7] = ' ' then Go(7) else Go(3)

6. if Posswin(o) then Go(Posswin(o)) elseif Posswin(x) then Go(Posswin(x))
   elseif Go(Make2)

7. if Posswin(x) then Go(Posswin(x)) elseif Posswin(o) then Go(Posswin(o))
   elseif Go(Anywhere)

8. if Posswin(o) then Go(Posswin(o)) elseif Posswin(x) then Go(Posswin(x))
   elseif Go(Anywhere)
Comments

+ More space efficient
+ Easier to understand and change

- Not as fast
- Total strategy has to be figured out in advance
- Not generalizable
The best solution – search

Basic idea – try every possible sequence of moves, assuming that your opponent is a genius. Don't take the moves allowing him to win!

Comments:

- Requires much more time.
+ Can be extended to more complex games.
+ Can be augmented with knowledge about game playing. Eg "heuristics".
LISP – LISt Processing

LISP is:

- The second oldest highlevel (computer) language in widespread use today.
- The most popular language within the field of Artificial Intelligence.
- The most hated language by computer science students today.
LISP

Developed 1958 by John McCarthy at MIT lab as abstract notion for *recursive functions*.

Builds heavily on *linked lists* as the primitive datatype. Manipulating primarily *symbols*.

Trivial syntax: every program is a list of instructions.

Metaprogramming – programs that write or modify other programs.

Two major dialects today: Common Lisp and Scheme.

Many different implementations of Common Lisp exists. Eg:
- CMU/CL – a free implementation from CMU. See http://www.cons.org/cmucl/
- CLISP – Another (slightly simpler) free implementation. See http://clisp.cons.org/
  **Allegro Common Lisp**, A common commercial implementation. See: http://www.franz.com/
### Programming paradigms

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow chart (&quot;goto programming&quot;)</td>
<td>Basic, Fortran</td>
</tr>
<tr>
<td>Structured programming</td>
<td>C, Pascal</td>
</tr>
<tr>
<td>Object-oriented programming</td>
<td>Smalltalk, C++, CLOS</td>
</tr>
<tr>
<td>Process programming</td>
<td>Ada</td>
</tr>
<tr>
<td>Functional programming</td>
<td>pure LISP, ML</td>
</tr>
<tr>
<td>Rulebased programming</td>
<td>Prolog, Expertsystems</td>
</tr>
</tbody>
</table>

LISP is usually associated to *functional programming* although it can do most of these paradigms.
LISP properties

- Symbol manipulation, lists. Example:
  a, b, foo, mathias,
  (+ a b) (car (color red) (owner mathias)) ()
  (1 2 (3 a b) (4 (5)))

- Interpreted
  (can also compile code)

- Incremental program development

- Data and programs

- Efficient garbage collections
LISP is *interactive*

- User writes expression at prompt
- LISP system interprets expression
- LISP prints result

Example:

```lisp
> (+ 1 2)
=> 3
> (defun inc(x) (+ 1 x))
=> INC
> (inc 3)
=> 4
```

This is the *READ–EVAL–PRINT loop*

Realy usefull for interactive development and debugging. Eg. rewriting the code while the system is running!
Data and Programs

Data and programs have the same form.

- Programs can be input data for a program
  (Programs can modify themselves)

- Define your own language on top of LISP
  (Embedded language)

- Functions are data objects
  (Functions can be parameters to other functions)
S-expressions in LISP

- Atoms:
  - a b c foo hi hello

- Numbers
  - 1 2 3 47 0.1 11.0

- Recursive definition:

  since: 1 2 3 a b c are S-expressions
  therefore: (1 2 3 a b) is an S-expression
  therefore: (1 2 (a b) 3) is an S-expression
Forms

A form is an *S-expression* that can be *evaluated* (computed).
Let *S* be any S-expression that we want to evaluate:

- If *S* is a number the the result of evaluating *S* is that number.
- If *S* is a variable name (atomic symbol) than the result of evaluating *S* is the (current) content of that variable. If no such variable exists -> error.
- If *S* is a *special form* – treat specially.
- If *S* is a list then the first element of the list is considered a function name and the rest of the elements are evaluated to get the arguments. The result of evaluating *S* is the result of applying the function to the given arguments.

> 10
=> 10
> (+ 2 5)
=> 7
> (* 2 (+ 3 4))
=> 14
Lecture 2 - LISP programming

Ealier:

General about AI, The Chinese room, Turing test, Loebner competition etc.
(Chapter 1 in *The Essence of AI*)
Basic LISP programming, functions
(Online LISP material (eg. the *LISP primer*) or books (eg. *Haraldsson*))

Now:

LISP programming: specialforms, conditionals,. recursion, higher order functions, lambda expressions etc.
(Online LISP material (eg. the *LISP primer*) or books (eg. *Haraldsson*))
Special forms

Some lisp expressions are treated in a non standard way. These are called *special forms*. Through the use of *macro's* it's possible to define new *special forms*.

Special form: DEFUN

```
(defun name (parameter1 ... parameterN)
  function-body
)
```

Creates a new function with given name and parameters. When called the variables `parameter1 ... parametersN` are bound to the given arguments and the expression(s) `function-body` are evaluated and returned.

Special form: SETQ

```
(setq name expression)
```

Sets the value of variable `name` to the result of executing `expression`. 
Special forms

Special form: QUOTE

(quote expression)
'expression

Returns expression without evaluating it. Example:

> (quote (+ 2 3))
=> (+ 2 3)
> (quote X)
=> X
> (quote (+ 1 X))
=> (+ 1 X)
> 'X
=> X
> '(+ 1 (* 2 X))
=> (+ 1 (* 2 X))

Special variable: NIL

nil
()

This is a special variable meaning the empty list.
Manipulating lists

(list expression1 ... expressionN) Create a new list contain the given arguments.
(cons head body) Appends the first argument head to the given list body.
(car L) (first L) Returns the first element of the given list.
(cdr L) (rest L) Gives all but the first element of a list.

(second L) (third L) ... Returns second/third etc. element of L.
(cadr L) Same as (car (cdr L))
(caddr L) Same as (car (cdr (cdr L)))
(cdadr L) ... Same as (cdr (car (cdr L)))
(append L1 L2) Appends the two lists together. Note difference to cons!

(member X L) True if X is a member of L
Functions for testing datatype

(symbolp X) True iff X is a symbol
(numberp X) True iff X is a number
(listp X) True iff X is a list
(consp X) True iff X is a list of at least one element.
(atom X) True iff X is an atom (symbol, number, ..., not a list)
(null X) True iff X is the empty list (NIL)

(eq X Y) True iff X and Y are the same object (special case for lists)
(equal X Y) True iff X and Y consists of the same object (also lists)
Conditionals

(if condition t-expr f-expr) Evaluates condition and if true (not NIL) evaluate and return t-expr otherwise evaluate and return f-expr.

(when condition t-expr) Evaluates condition and if true evaluate and return t-expr, otherwise return nil.

(cond Evaluates pred-1 and if true evaluates and returns expr-1, otherwise continues with next test. Return NIL if no test is satisfied. Use special variable T for the always-true case.
   (pred-1 expr-1)
   ...
   (pred-N expr-N)
 )

(and expr-1 ... expr-N) Evaluates expr-1 through expr-N. Stops and returns false if any result is false, otherwise returns last result.

(or expr-1 ... expr-N) Evaluates expr-1 through expr-N. Stops and returns true if any result is true, otherwise returns NIL.

(not expr-1) Returns true iff expr-1 evaluates to false
Manipulating numbers

(+ x1 ... xN)  Gives the sum of x1 ... xN
(- x1 ... xN)  Gives x1 minus sum of x2 ... xN

(* x1 ... xN)  Gives the product of x1 ... xN
(/ x1 ... xN)  Gives x1 divided by sum of x2 ... xN

(< x1 ... xN)  True iff x1 ... xN monotonically increasing.
(<= x1 ... xN) True iff x1 ... xN monotonically non decreasing.
(> x1 ... xN)  Monotonically decreasing.
(>= x1 ... xN) Monotonically non increasing.
(= x1 ... xN)  All the same.
(/= x1 ... xN) All different
Some other functions:

(eval expr) Evaluates it's argument (given by evaluating expr)

(write expr) Writes it's argument to stdout

(read) Wait for input from stdin and returns the expression read.

(format t string arg0 .. argN) Prints string to stdout substituting some special characters for the (evaluated) arg0 ... argN. For instance:

~a Print next argument
~% Print a newline

Example of using format:

> (format t "1 + 1 is ~a says LISP~%Right?~%" (+ 1 1))
gives:
1 + 1 is 2 says LISP Right?
Exercises

1) Write a function "absolut" that takes a number and returns the absolute value of it. You may not use the built in function abs in lisp. Example:
   (absolut 3)
   => 3
   (absolut -7)
   => 7

2) Write a function my-sum that takes a list of two numbers and returns the sum of these two numbers.
   Example:
   (my-sum '(1 2))
   => 3
   (my-sum '(10 10))
   => 10

3) Write a function "efternamn" that takes a symbol denoting a name. If the name is Mathias then it should return the symbol Broxvall, if the name is Ingmar then it should return Stenmark and otherwise it should return unknown.

4) Solve exercise 3 without using the "if" specialform.
Lecture 3

Previously:
- S-expressions, forms, evaluating forms
- Special forms
- Arithmetic functions
- List manipulation functions (first, rest, cons, append, ...)
- Conditionals (if, cond, and, or, while, ...)
- Creating functions (defun)

Today
- Recursion
- Recursion
- Recursion
- ...

Artificial Intelligence
HT2005
Trace – seeing what's happening

One useful way of debugging lisp programs is to watch all the procedure calls. You can do this with the special form `trace`. Example.

```lisp
> (defun inc(X) (+ 1 X))
> (defun dec(X) (- X 1))
> (defun foo(X Y)
    (* (inc X) (dec Y)))
> (trace inc dec foo) <- --- turns on tracing these functions
> (foo 2 3)
  0: (FOO 6 4)
    1: (INC 6)
    1: INC returned 7
    1: (DEC 4)
    1: DEC returned 3
  0: FOO returned 21
=> 21

> (untrace)
```
Example of recursion

If we want to implement the faculty operator.

- \(1! = 1\)
- \(N! = N \times (N-1)!\)

We can do this like this:

```lisp
(defun fac (N)
  (if (= N 1)
      1
      (* N (fac (- N 1)))
  )
)
```

> (fac 1)
=> 1
> (fac 3)
=> 6
> (fac 5)
=> 120
Tracing the faculty function

How does this recursion work? Let's trace it and see!

> (trace fac + -)
> (fac 4)
0:  (FAC 4)
  1:  (- 4 1)
    1:  - returned 3
    1:  (FAC 3)
      2:  (- 3 1)
      2:  - returned 2
      2:  (FAC 2)
        3:  (- 2 1)
        3:  - returned 1
        3:  (FAC 1)
        3:  FAC returned 1
        3:  (* 2 1)
        3:  * returned 2
        2:  FAC returned 2
        2:  (* 3 2)
        2:  * returned 6
      1:  FAC returned 6
      1:  (* 4 6)
    1:  * returned 24
0:  FAC returned 24
General idea of recursion

- Make the program work for a base case (e.g., N=1, the empty list, ...)
- For more advanced cases, decompose into simpler cases and call yourself (e.g., (* N (fac (1-N))))

This is similar to the mathematical idea of:

**PROOF BY INDUCTION**

You can use proof by induction to prove correctness of the program. E.g., prove that your "fac" program computes the factorial. Example:

Base case: For N=1 our program clearly returns N! which is 1
Inductive case: Assume that our program computes N! for all N < some k. If we call our program with N=k+1 then our program returns (k+1) * fac(k) = (k+1) * k! = (k+1)!. Thus our program computes the factorial function also up to k+1

Ergo, our program computes the factorial function for arbitrary numbers...
Recursion as a general form of iteration

Assume that we want to test if a list contains any numbers? Formulate problem recursively:

- The empty list never contains any numbers.
- If the first element of a list is a number, then the list contains (at least) one number.
- Otherwise, the list contains numbers if the rest of the list contains numbers.

Or formulated as LISP code:

```lisp
(defun contains-number (L)
  (cond
   ((null L) nil)
   ((numberp (first L)) T)
   (T (contains-number (rest L)))))
```

Test it:

```
> (contains-number '(r 2 d 2))
=> T
> (contains-number '(a c d c))
=> nil
```
Tracing our recursive function

> (trace contains-number)

> (contains-number '(r 2 d 2))
  0: (CONTAINS-NUMBER (R 2 D 2))
    1: (CONTAINS-NUMBER (2 D 2))
      1: CONTAINS-NUMBER returned T
  0: CONTAINS-NUMBER returned T
=> T

> (contains-number '(a c d c))
  0: (CONTAINS-NUMBER (A C D C))
    1: (CONTAINS-NUMBER (C D C))
      2: (CONTAINS-NUMBER (D C))
        3: (CONTAINS-NUMBER (C))
          4: (CONTAINS-NUMBER NIL)
          4: CONTAINS-NUMBER returned NIL
            3: CONTAINS-NUMBER returned NIL
              2: CONTAINS-NUMBER returned NIL
                1: CONTAINS-NUMBER returned NIL
                  0: CONTAINS-NUMBER returned NIL
        NIL
Example 2 – Summing all numbers in a list

Problem: Compute the sum of the numbers in a given list.

Recursive definition:

- The sum of the empty list is zero
- The sum of a list \((x_0 \ldots x_N)\) is \(x_0\) plus the sum of \((x_1 \ldots x_N)\)

LISP solution:

```
(defun sum (L)
  (if (null L)
      0
      (+ (first L) (sum (rest L)))))
```

Testing it:

```
> (sum '())
0
> (sum '(1 2 3))
6
```
Tracing example 2

> (trace sum +)
> (sum '(1 2 3))
0: (SUM (1 2 3))
  1: (SUM (2 3))
  2: (SUM (3))
    3: (SUM NIL)
    3: SUM returned 0
    3: (+ 3 0)
    3: + returned 3
  2: SUM returned 3
  2: (+ 2 3)
  2: + returned 5
1: SUM returned 5
1: (+ 1 5)
1: + returned 6
0: SUM returned 6
6
Template for sequential processing

General template for sequential processing of all elements in a list:
  • What should the result be for the empty list? \textit{'start-value'}
  • How do you combine the results with each other? \textit{'op'}

\begin{verbatim}
(defun fn (L)
  (if (endp L)
      'start-value'
      (op (first L) (fn (rest L)))
    ))
\end{verbatim}

or

\begin{verbatim}
(defun fn (L)
  (cond ((endp L) 'start-value')
        ('other-predicate' 'expression')
        (T ('op' (first L) (fn (rest L))))
      ))
\end{verbatim}
Template for sequential processing – example 1

Remove all instances of element X from a list.
• Base case: for the empty list we return the empty list
• If X is the first element of the list, just return the recursive result on the rest.
• Otherwise, append the first element of the list to the recursive result of the rest.

(defun remove (X L)
  (cond
    ((endp L) nil)
    ((equal X (first L)) (remove X (rest L)))
    (T (cons (first L) (remove X (rest L))))
  ))

Testing it:
> (remove 42 '())
NIL
> (remove 1 '(1 2 3))
(2 3)
> (remove 2 '(1 2 3))
(1 3)
Exercise

1. Create a function `add-one` that takes a list of numbers and adds one to each number and creates a new list of the results. Eg:

   (add-one '(1 2 3))
   => (2 3 4)

2. Create a function `my-sum` that takes two lists of numbers which are equally long and creates a new list whose elements are the sum of the two corresponding elements from the original lists. Eg:

   (my-sum '(1 2) '(2 4))
   => (3 6)

   (my-sum '(1 2 3) '(3 2 1))
   => (4 4 4)

3. Write a `my-remove` function that given a list containing sublists of numbers, it removes all sublists with sum > 0.

   (my-remove '( (0 -1) (-2 -3 +3) (-5 +5 +1) )

=> ( (0 -1) (-2 -3 +3) )
Lots of recursion

Problem: We have a list containing sublists of numbers, remove all sublists with sum > 0.
Solution: Define two functions – one computing the sum of lists and one doing the removing.

```lisp
(defun sum-all (L)
  (if (null L)
      0
      (+ (first L) (sum-all (rest L)))
  ))

(defun my-remove (L)
  (cond
   ((null L) NIL)
   ((> (sum-all (first L)) 0) (my-remove (rest L)))
   (T (cons (first L) (my-remove (rest L))))
  ))

Example:
> (my-remove '((-1 1) (0 1) (2 3) (2 1 -4)))
((-1 1) (2 1 -4))
```
Double recursion

Example: We want to compute the sum of all numbers in a list, including those inside sublists (and subsublists etc.) of the original list.

Recursive formulation:

• The sum of the empty list is zero.
• The sum of a list beginning with a number is that number plus the sum of rest.
• The sum of a list beginning with a sublist is the sum of the sublist plus the sum of the rest.

(defun sum-all (L)
  (cond
   ((null L) 0)
   ((listp (first L))
    (+ (sum-all (first L)) (sum-all (rest L))))
   (T
    (+ (first L) (sum-all (rest L))))))
Tracing a double recursive function

> (trace sum-all +)
> (sum-all '(1 (1 1)))
  0: (SUM-ALL (1 (1 1)))
    1: (SUM-ALL ((1 1)))
      2: (SUM-ALL (1 1))
        3: (SUM-ALL (1))
          4: (SUM-ALL NIL)
          4: SUM-ALL returned 0
          4: (+ 1 0)
          4: + returned 1
          3: SUM-ALL returned 1
        3: (+ 1 1)
        3: + returned 2
      2: SUM-ALL returned 2
      2: (SUM-ALL NIL)
      2: SUM-ALL returned 0
      2: (+ 2 0)
      2: + returned 2
    1: SUM-ALL returned 2
    1: (+ 1 2)
    1: + returned 3
  0: SUM-ALL returned 3
3
Template for double recursion

General template for double recursion:

- What should the result be for the empty list?  
  'start-value'
- How do you combine the results with each other?  
  'op'

```
(defun fn (L)
  (cond
    ((endp L) 'start-value')
    ((listp L) ('op' (fn (first L)) (fn (rest L))))
    (T ('op' (first L) (fn (rest L))))
  ))
```
Auxiliary functions

Sometimes you need extra "temporary" values for your functions. Define a wrapper function calling an auxiliary function with the extra parameters. Example define a function printing every item of a list on one new line with an index in front of it.

(defun my.pretty-print (L)
   (my.pretty-print* L 0))
(defun my.pretty-print* (L index)
   (if (null L)
       (format t "END.~%")
       (progn
           (format t "~a.~t~a~%" index (first L))
           (my.pretty-print* (rest L) (1+ index))
       )))
Testing my.pretty-print

> (my.pretty-print '(foo (boo 42) fum (fie bum)))
  0. FOO
  1. (BOO 42)
  2. FUM
  3. (FIE BUM)
END.
NIL

> (trace my.pretty-print my.pretty-print*)
> (my.pretty-print '(foo boo))
  0: (MY-PRETTY-PRINT (FOO BOO))
      1: (MY-PRETTY-PRINT* (FOO BOO) 0)
  0. FOO
      2: (MY-PRETTY-PRINT* (BOO) 1)
  1. BOO
      3: (MY-PRETTY-PRINT* NIL 2)
END.
      3: MY-PRETTY-PRINT* returned NIL
      2: MY-PRETTY-PRINT* returned NIL
      1: MY-PRETTY-PRINT* returned NIL
      0: MY-PRETTY-PRINT returned NIL
NIL
>
Basic LISP exercises

1. Write a function \texttt{twice(L)} that takes a list makes it twice as long by copying everything to the end of it. Eg:
\[
\texttt{(twice '(a b c)) => '(a b c a b c)}
\]

2. Assume that we have database of employees defined as follows:
\[
\texttt{(setf employee-database}
\]
\[
\texttt{'((kalle office-worker 10000))}
\]
\[
\texttt{(erik technician 15000)}
\]
\[
\texttt{(anna office-worker 15000)}
\]
\[
\texttt{...}
\]
\[
\texttt{)}
\]
Write a function that iterates over the database and computes the sum of all the salaries. Write a function that sums the salaries of all office-workers.

3. Write a double recursive function \texttt{my-reverse} that takes a list and reverses the order of all the elements within it. Elements within sublists should also be reversed.
Lecture 4

Previously:
- Recursion
- Recursion
- Double recursion

Today
- Variables
- Higher order functions
- ...

Artificial Intelligence
HT2005
Local variables

Special form:

(let ( (variable1 expression1)
    (variable2 expression2)
    ...
    (variableN expressionN) )
  body
)

Evaluates expression1 .. expressionN and assigns result to variable1 .. variableN. After this body is evaluated (using eg. variable1 .. variableN) and returned as result of whole let statement.

(let* ((variable1 expression1)
    ...
    (variableN expressionN) )
  body
)

Same as above with exception that variable-i can be used in expression-j for any j > i.
Example of local variables

We want to compute the distance between two points given as \((x_1 \ y_1)\) and \((x_2 \ y_2)\)

;; Computes the distance between point P1 and P2
(defun distance (P1 P2)
  (let ((delta-x (- (first P1) (first P2)))
        (delta-y (- (second P1) (second P2))))
    (sqrt (+ (* delta-x delta-x)
             (* delta-y delta-y))))
)
Global variables:
To create a new global variable use defvar. To modify global variables use setq or setf.

(defvar variable init-value) Creates a new global variable with given initial value (optional!)
(setq variable value) Sets the value of a global or local variable
(setf variable value) General version of setq. Also work on content of structures eg:
(setq (car L) 42)
Functions as data

Functions in lisp can be accessed as data.

```lisp
> #'foo
#<Interpreted Function FOO {486337C1}>
```

And manipulated just like data...

```lisp
> (cons #'foo '(1 2 3))
(#<Interpreted Function FOO {486337C1} > 1 2 3)
```

If you store it in a variable, you can later call it

```lisp
> (defun foo (fn)
   (funcall fn 1 2 3))
> (foo #'list)
(1 2 3)
> (foo #'+)
6
```

You can even create new functions on the fly

```lisp
> (foo (lambda (x y z) (+ x (* y z))))
7
```
Higher order functions

(lambda (arg1 ... argN) body)  Creates a new function taking argument arg1 ... argN and evaluating body with these arguments when called. *Inherits* all other variables from where the function is created.

#'function-name
name  Returns the function corresponding to given name

(funcall function-object arg1 ... argN)  Evaluates the given function object using given arguments.
Higher order functions – example zero

Write a function `my-iter` that iterates over a list of numbers and combine them using a given operator. Eg:

```
(my-iter #'+ '(1 2 3 4))
=> 10
```

```
(my-iter #'* '(1 2 3 4))
=> 24
```

Solution:

```
(defun my-iter (fn L)
  (if (endp L)
    ()
    (funcall fn (first L) (my-iter fn (rest L))))
```

We call this function `my-filter` a *higher order function* since it accepts functions as arguments.
Higher order functions – example 1

Write a function my-filter that returns the elements of a list that satisfy a test given as another function.

> (my-filter (lambda (X) (< X 6)) '(1 2 7 3))
(1 2 3)

(defun my-filter (fn L)
  (cond
   ((null L) '())
   ((funcall fn (first L))
    (cons (first L) (my-filter fn (rest L))))
   (T
    (my-filter fn (rest L))))
)
Higher order functions – example 2

Write a function that computes the sum of the square of all integers in the range N ... M

\[
\text{sum-of-squares} \ (N \ M) =
\begin{cases}
0 & \text{if } N > M \\
\text{sum-of-squares} \ (1 + N) \ M + \text{square} \ N & \text{otherwise}
\end{cases}
\]

Write a function that computes the sum of \( \sin(i) \) for all integers \( i \) in the range N ... M

\[
\text{sum-of-sin} \ (N \ M) =
\begin{cases}
0 & \text{if } N > M \\
\sin \ N + \text{sum-of-sin} \ (1 + N) \ M & \text{otherwise}
\end{cases}
\]

Write a function that computes the sum of \( f(i) \) for all integers \( i \) in the range N ... M

\[
\text{sum-of-fn} \ (fn \ N \ M) =
\begin{cases}
0 & \text{if } N > M \\
n(f(n)) + \text{sum-of-fn} \ fn \ (1 + N) \ M & \text{otherwise}
\end{cases}
\]
Some built-in *higher order functions*

\[
\begin{align*}
\text{mapcar } \text{fn } L & \quad \text{Applies fn to each element of L and cons'es the result} \\
\text{sort } L \text{ fn} & \quad \text{Sorts L using the given precedence operator}
\end{align*}
\]

\[
\begin{align*}
> \text{ (mapcar } \text{'1+ } \text{'(1 2 3) } \\
& \quad \text{(2 3 4)} \\
> \text{ (mapcar } \text{'list } \text{'(a b c) } \\
& \quad \text{((a) (b) (c))}
\end{align*}
\]

\[
\begin{align*}
> \text{ (sort } \text{'(1 6 4 7) 's<} \\
& \quad \text{(1 4 6 7)} \\
> \text{ (sort } \text{ '((1 3) (0 2) (5 1)) (lambda } \text{(X Y) } \text{(first X) (first Y))} \\
& \quad \text{((0 2) (1 3) (5 1))} \\
> \text{ (sort } \text{ '((1 3) (0 2) (5 1)) (lambda } \text{(X Y) } \text{(second X) (second Y))} \\
& \quad \text{((5 1) (0 2) (1 3))}
\end{align*}
\]
Exercises

1. Write a higher order function **my-iter** that takes a function F and a list L and applies F to every atom in L or inside any sublist within L. E.g:

   > (my-iter #'print '(a (b) c ((d)))))
   a
   b
   c
   d
   => nil

2. Write a function **compose** that takes two functions F and G of **arity 1** and returns a new function that is F(G(X))

3. Rewrite the function **compose** so that it takes a list of functions F1 .. Fn all of arity 1 and returns a new function that is F1(F2(F3(... Fn(X) ...)))
Lecture 5

Previously:
  • Recursion, double recursion, ...
  • Local /global variables
  • Higher order functions, ...

Today:
  • Representing and storing data
  • Cons-cells, dotted lists, circular lists, ...
  • Destructive functions
Association lists

If we want to store and lookup a set of facts we can do so with *association lists*. An association list is a just list of (key value) pairs. Eg:

```
'((shape round) (colour red) (area 42))
```

We insert "facts" by adding (key value) pairs to the beginning of the list and/or substituting current values in the list. Eg. Inserting the fact that *name* is *ball-3* gives

```
'((name ball-3) (shape round) (colour red) (area 42))
```

Functions to do this automatically:

- (acons key value a-list) Adds (key value) to the front of a-list
- (assoc key a-list) Returns (key value) from a-list if possible
Property lists

LISP has a built in database associating (key value) pairs to every symbol in lisp.

> (get 'ball-3 'shape)  
NIL
> (setf (get 'ball-3 'shape) 'round)  
> (setf (get 'box-1 'shape) 'square)  
> (get 'ball-3 'shape)  
ROUND
> (get 'box-1 'shape)  
SQUARE

We can even apply get to symbols returned by arbitrary expressions (eg. from variables):
> (defvar X 'ball-3)  
> (get X 'shape)  
ROUND

(symbol-plist symbol) Returns all keys and values defined for this symbol
(gensym) Creates a fresh symbol.
Abstract datatypes

Idea: We decide to use lisp lists and other constructs to store data in a structured way and give functions to create and access this data. The data is never accessed in any otherways. Eg:

(defun create-coordinate (X Y) (list 'coordinate X Y))
(defun coordinate-x (C) (second C))
(defun coordinate-y (C) (third C))

;; Computes the length of a coordinate vector
(defun my-length (C)
  (sqrt (+ (* (coordinate-x C) (coordinate-x C)) (* (coordinate-y C) (coordinate-y C)))))

Advantages:
- Easier to modify program – if we decide to change how data is stored we only modify the accessor functions.
- Easier to understand what the program is doing.
- Clean interface with the user.
- Less risk for bugs.
Data abstraction example

Assume that we want to have a database of information about students and teachers and that we store the information as follows:

('student name (courses))
('teacher name)
('course name teacher points)

**Without abstraction:**

(setq BO (list 'teacher "Bo")
(setq MATHIAS (list 'teacher "Mathias")
(setq MATH (list 'course "math" MATHIAS 3))
(setq AI (list 'course "ai" MATHIAS 5))
(setq NILS (list 'student "Nils" (list MATH AI)))

**Changing teacher in the math course**

(setf (third MATH) BO)

(defun name-all-teachers-of-student (student)
  (name-all-teachers-of-student* (third student)))
(defun name-all-teachers-of-student* (X)
  (if (null X)
    ()
    (cons (second (third X))
      (name-all-teachers-of-student* (rest X)))))

(name-teachers-of-student NILS)
=> ("Bo" "Mathias")
Data abstraction

Create functions that interface with the data and the rest of the program:

- **Constructors** – creates new instances of the data
- **Accessors** – Extracts data
- **Modifiers** – Modifies data
- **Recognisers** – Checks if a piece of data is of the given type

```lisp
;; Ex. Constructor
(defun make-course (name points)
  (list 'course name points))

;; Ex. Accessor
(defun course-name (course) (second course))
(defun course-points (course) (third course))

;; Ex. Recogniser
(defun is-course (X) (and (consp X) (equal (first X) 'course)))

;; Ex. Modifier
(defun course-name! (course new-name) (setf (second course) new-name))
(defun course-points! (course new-points) (setf (third course) new-points))

;; Use it!
(defun name-teachers-of-student (student)
  (mapcar (lambda (course) (teacher-name (course-teacher course)))
          (student-courses student)))
```
Cons cells

LISP lists consists of cons cells – ie. LISP lists are singly linked lists.

The list '(a b c) are represented by three conscells pointing to the symbols a,b,c and the nil object.

The cons operator creates a new conscell pointing to it's two arguments.

The car returns the first pointer, the cdr the second pointer.

You can change the contents of a cons-cell by doing setf on it:

```
(setf L 'a (b c))
(setf (cdr L) '(d e))
L => (a d e)
```
Dotted pairs

You can create things that are not lists by doing cons on arbitrary objects. These non-lists are printed using a *dotted* notation.

```lisp
> (cons '(a b) 'c)
((A B) . C)
```

You can also enter these expressions by hand:

```
'((a b) . c)
```

Or create them using `setf`.

```lisp
(setf L '(1 2 3))
(setf (car L) '(a b))
(setf (cdr L) 'c)
```
Binary trees: Conscells can define binary trees

;; Do op1 on every leaf and combine results
;; using op2
(defun fn (op1 op2 tree)
  (if (atom tree)
      (funcall op1 tree)
      (funcall op2
                   (fn op1 op2 (car tree))
                   (fn op1 op2 (cdr tree))
      )))

(defun fn #'+ #'cons '((1 . 2) . 3))
=> ((2 . 3) . 4)

(fn #'+ #'cons '((1 . 2)
         . 3))
=> ((2 . 3) . 4)
Circular lists

Sometimes we can (unintentionally?) create *circular* lists.

> (setq L '(a b c))
> (setq (cdr (cdr (cdr L))) L)
(A B C A B C A B C A B C A B C A B C A B C A B C A B C A B C A B C A B C A B C A B C A B C A B C A B C ...)

> (setq *print-circle* t)
#1=(A B C . #1#)
Destructive functions

Functions which are modifying their arguments (assuming that the arguments are lists, structures or something else that can be modified) are called destructive.

```lisp
(defun my-inc (L)
  (if (null L)
    ()
    (progn
      (setf (car L) (1+ (car L)))
      (my-inc (rest L))
      L)
  ))
```

Some built in functions, eg. sort, nconc etc. , are destructive.

```lisp
> (setf my-list '(2 1 4 3))
> (sort my-list #'<)
(1 2 3 4)
> my-list
(2 3 4)

> (setf A '(a b))
> (nconc A '(c d))
(a b c d)
> A
(a b c d)
```
Structures

Structures are used for implementing data abstraction. They are provided in the lisp language together with functions to:

- Create instances
- Modify instances
- Recognize instances
- Extract information from an instance
;; Define a structure
(defstruct student name courses)

;; Create instance of structure
(setf kalle (make-student))

;; Modify instances
(setf (student-name kalle) "Karl Eriksson")

;; Access fields
(student-name kalle)

;; Recognize a student
(student-p kalle)

;; Create a student with given values
(make-student :name "Kalle")

;; Define students with default values
(defstruct student (name "John Doe") ...)

Artificial Intelligence
HT2005
Iterative functions in LISP

LISP can do iterative programming through eg. the dotimes, dolist and do constructs.

(dotimes (variable times [result])
  
  body1
  ...
  bodyN)

First evaluates times giving a non negative integer and then evaluates body1 ... bodyN repeatedly with variable assigned the value 0 through times-1.

Finally, evaluates result and returns the result.

(dolist (variable list [result])
  
  body1
  ...
  bodyN)

Evaluates list once and performs repeatedly body1... bodyN with variable iterating over the elements of list.

Finally, evaluates and returns result.
General iteration constructs

\[
(\text{do } ((\text{variable-1 } \text{initial-value-1} \ [\text{step-1}]) \\
\ldots \\
(\text{variable-N } \text{initial-value-N} \ [\text{step-N}]) ) \\
(\text{end-expr } \text{result-expr}) \\
\text{body1} \\
\ldots \\
\text{bodyN})
\]

A general iterative construct equivalent to a general \textit{for loop} in C.

1. \textit{variable-1} ... \textit{variable-N} are assigned the result of evaluating \textit{initial-value-1} ... \textit{initial-value-N}.
2. \textit{end-expr} is evaluated, if true \textit{result-expr} is evaluated and returned. (finished)
3. \textit{body1} ... \textit{bodyN} is evaluates (using eg. \textit{variable-1} ... \textit{variable-N})
4. \textit{step-1} ... \textit{step-N} is evaluated and assigned to \textit{variable-1} ... \textit{variable-N}
5. Goto 2.
The LOOP construct

Another iterative lisp construct is the loop construct.

(loop
  body1
  ...
  bodyN)

Evaluates body1 ... bodyN repeatedly for ever.

(return expression)

Returns from a loop construct with given value

Example:

(let ((i 0))
  (loop
    (do-something
     (when (> i 10)
       (return nil))
    )))

;; Finished after 10 loops
Optional parameters

Functions can be defined to take a variable amount of parameters using *optional* parameters, *keyword* parameters and *rest* parameters.

Optional parameters have a *default value* used if that parameter is not specified.

```
(defun foo (x &optional (y 0) (z 0))
  (+ x y z))
```

```
(foo 1)
=> 1
(foo 1 2)
=> 3
(foo 1 2 3)
=> 6
```

Note. You can also use optional parameters in lambda expressions and/or macros.
Keyword parameters

Keyword parameters are a way of having many optional parameters that can be named during function invocation. Gives eg. more readable code when many (optional) parameters.

```
(defun foo (X &key (value 42) (increment 0) (multiplier 1))
  (cons X (* (+ value increment) multiplier)))

(foo 0)
=> (0 42)
(foo 0 :value 11)
=> (0 11)
(foo 'kalle :increment 2)
=> (kalle 44)
(foo 'kalle :value 0 :increment 11 :multiplier 2)
=> (kalle 22)
```

You can also have a `&rest` variable which gets a list of all remaining parameters.

```
(defun foo(&rest args) args)
(foo 1 2 3)
=> (1 2 3)
(foo)
=> nil
```
Macros

Create special forms using macros

(defmacro name arguments body)

The result of evaluating a macro is a piece of code which should be evaluated inplace of the macro.
Week 46

Revised lecture content:

Monday w46: More LISP
Thursday w46: General problem solving and search
Monday w47: Expertsystems

Reading instructions:

*Before* thursday w46: Chapter 4 in Cawsey
*Before* Monday w47: Chapter 3 in Cawsey

Lab2:

Now available on the homepage.
Solve the mapcolouring problem using search.
Read it before going to the lab!
Brief introduction to Complexity theory

Computational complexity is the study of the difficulty of solving computational problems, in terms of the required computational resources, such as time and space (memory).

Note: It does not calculate in general the actual time it takes to solve a problem in a specific computer, but it gives an expectation of how resource consuming finding a solution is as the input gets bigger. We generally express this by an ordo expression.
Complexity of an algorithm

The complexity of an algorithm, given an input of length $n$, is the number of steps needed, in the worst case, to terminate the algorithm. It is indicated by $O(f(n))$, where $f(n)$ is a function that has $n$ as parameter.

Example: a function takes time linear to its size has the complexity $O(n)$.

Complexity of a problem

The complexity of the best algorithm found until now to solve the problem.
Categorization of complexity of problems

**polynomial**: they can be solved in polynomial time with respect to the size of the input.
Examples:

- *constant time* $O(1)$ - for instance retrieving a value from an hash table
- *linear time* $O(n)$: for instance checking if an element is present in an unsorted list
- *logarithmic time* $O(log(n))$: for instance checking if an element is present in a sorted list
- $O(n*log(n))$: for instance sorting a list

**exponential**: they can be solved in exponential time with respect to the size of the input.
Examples:

- $O(2^n)$: for instance satisfiability problem - checking if a boolean formula is true for at least one assignment of values to the variables
Classes of complexity

P: they can be solved in polynomial time with respect to the size of the input.

NP: P plus all problems for which the best algorithm found up to now is exponential, but the solution can be verified in polynomial time.

NP-complete: sub-class of NP of specially important (and difficult) problems. If one could find an algorithm that solves a problem in the NP-complete class in polynomial time, it would be possible to solve all the problems in NP in polynomial time:

The holy grail of complexity theory:

P = NP ???
The Water Jug Problem

Given: one three-liter jug
one four-liter jug
tap that can fill the jugs with water

Goal: exactly two liters of water in the four-liter jug
Search Space – States

Define a state space that contains all possible configurations.
Specify the initial states (one or more).
Specify goal states.
Specify a set of rules that describes current state is modified ("operators").

Example:

We represent the amount of water in the jugs with $(X,Y)$

1. $(X,Y) \rightarrow (4,Y)$  Fill the 4 liter jug.
2. $(X,Y) \rightarrow (X,3)$  Fill the 3 liter jug.
3. $(X,Y) \rightarrow (0,Y)$  Empty the four liter jug
4. $(X,Y)$ if $X+Y \geq 4$ and $Y > 0 \rightarrow (4,Y-(4-X))$  Fill the 4 liter jug with water from the 3 liter jug.

...
Water Jug Problem

1. \((X,Y: \ X < 4) \rightarrow (4,Y)\) 
   Fill the 4-liter jug

2. \((X,Y: \ Y < 3) \rightarrow (X,3)\) 
   Fill the 3-liter jug

3. \((X,Y: \ X > 0) \rightarrow (0,Y)\) 
   Empty the 4-liter jug on the ground

4. \((X,Y: \ Y > 0) \rightarrow (X,0)\) 
   Empty the 3-liter jug on the ground

5. \((X,Y: \ X+Y \geq 4 \text{ and } Y > 0) \rightarrow (4,Y-(4-X))\) 
   Fill the 4-liter jug from the 3-liter jug

6. \((X,Y: \ X+Y \geq 3 \text{ and } X > 0) \rightarrow (X-(3-Y),3))\) 
   Fill the 3-liter jug from the 4-liter jug

7. \((X,Y: \ X+Y \leq 4 \text{ and } Y > 0) \rightarrow (X+Y,0)\) 
   Pour all water from the 3-liter jug into the 4-liter jug

8. \((X,Y: \ X+Y \leq 3 \text{ and } X > 0) \rightarrow (0,X+Y))\) 
   Pour all water from the 4-liter jug into the 3-liter jug

9. \((X,Y: \ X > 0) \rightarrow (X-D,Y)\)

10. \((X,Y: \ Y > 0) \rightarrow (X,Y-D)\)
Some of the possible states we can reach by using these operators

Water Jug Problem
Blind search

If we have no extra information to guide the search

• Depth first search
• Breadth first search
• Iterative deepening
Depth first

1. Set $N$ to be the set of initial nodes.
2. If $N$ is empty then signal failure and exit.
3. Set $n_1$ to be the element of $N$ and remove $n_1$ from $N$.
4. If $n_1$ is a goal node then signal success and exit.
5. Add all the children of $n_1$ to the front of $N$ and go to step 2 ("expand" $n_1$).
Breadth first search

1. Set $N$ to be the set of initial nodes.
2. If $N$ is empty then signal failure and exit.
3. Set $n_1$ to be the element of $N$ and remove $n_1$ from $N$.
4. If $n_1$ is a goal node then signal success and exit.
5. Add all the children of $n_1$ to the end of $N$ and go to step 2 ("expand" $n_1$).
Iterative Deepening

First define: \( depth\text{-}first\text{-}fixed\text{-}depth(\text{MAX}) \)

1. Set \( N \) to be the set of initial nodes.
2. If \( N \) is empty then signal failure and exit.
3. Set \( n1 \) to be the element of \( N \) and remove \( n1 \) from \( N \).
4. If \( n1 \) is a goal node then signal success and exit.
5. If the depth of \( n1 \) is equal to \( \text{MAX} \) then goto step 2.
6. Add all the children of \( n1 \) to the front of \( N \) and go to step 2.

Next: Call depth-first-fixed-depth with increasing max until you have found a solution.
Evaluation Criteria

**Complete:** Does the strategy always find a solution if one exists?

**Optimal:** Is the solution the shortest one?

**Space:** How much memory does the strategy need?

**Time:** How long time does it take to find a solution.
Evaluation

- Iterative deepening is asymptotically space and time optimal.

- Depth first search is preferred when goals are at leaves and there are no risk for loops.

- Breadth-first search is preferred in case of low branching factor, expensive operators or goals at reasonable depth.
Week 46

Lectures week 46:
- Optimization and Search
- Heuristic search
- Expert systems
  (Search and games)
  (Structured knowledge representation)

Reading instructions:
- Structured knowledge representation – chapter 2.1 – 2.3 in Cawsey
- Expert systems – chapter 2.4 & 3 in Cawsey

Labs:
- Lab 3 – Expert systems; Design a simple expert system.

Important: Since laptop is broken - take a look at lab3 before tomorrow and ask questions.
Optimisation and Search – Hill climbing

If the problem is to optimise some function associated to every world state.

Hill climbing:

1. Set N to be the initial node.
2. If the value of N is greater than the value of any of its children then return N and exit.
3. Set N to be the highest-value child and go to step 2.
Gradient Search

maximize $f(x)$

\[
\frac{d f(x)}{dx}
\]

\[
x \leftarrow x + b \frac{d f(x)}{dx}
\]
Problems with HC and gradient search

local minima
plateau
ridge
Heuristic Search

We use some extra domain information to guide the search.

- Best-first Search
- A* Search

**General idea:** We define a *heuristic* function gives us a general idea of how *good* or *bad* a given state is. The heuristic does not have to be perfect and can guide us in the wrong direction. Search allows us to find a solution anyway.
Best-first search

Heuristic: $H(n)$ – estimated distance to goal state.

1. Set $N$ to be the set of initial nodes.
2. If $N$ is empty then signal failure and exit.
3. Set $n_1$ to be the element of $N$ and remove $n_1$ from $N$.
4. If $n_1$ is a goal node then signal success and exit.
5. Add all the children of $n_1$ to $N$, sort the nodes of $N$ according to estimated distance to goal and go to step 2.
A* search

Best first search with distance function \( f(N) \)

\[
f(N) = g(N) + h(N)
\]

- \( g(N) \) - Distance ("cost") from root to \( N \)
- \( h(N) \) - Admissible heuristic

Properties:
- Optimal
- Complete for all graphs with a finite branching factor
- Exponential complexity in general.
- Large space requirements.
Expert systems

Systems that solve real problems that are usually solved by human experts.

- Reasoning about knowledge, inference rules.
- Design an expert system
- Reasoning under uncertainty
- Examples of expert systems
General idea

An expert system can be described as a system that:

- Can represent facts (information)
- Contains rules for adding new facts based on old information.
- Has an inference mechanism for drawing conclusions
- Asks the user *relevant* questions needed to draw the right conclusion
- Presents the user with the conclusions and their motivation.

Consider the following example dialogue from lab3.

* (prove 'computer-broken)
Is your computer on right now? unknown
Have you pressed the power button? yes
Is the computer plugged into the wall right now? no
COMPUTER-BROKEN proven with certainty 1.0
1. [1.0] Computer appears broken since it is not on
2. [-1.0] Computer is not on since it has no electricity
3. [-1.0] ELECTRICITY given by user
Representing information

Representing and manipulating *symbolic information* is the essence of expert systems. We therefore need a way to represent facts about the world and rules for drawing conclusions.

Simplest solution:
- Each fact is represented by a symbol.
- We have a list of all facts that are true in the world.

F1: HOT
F2: SMOKY

We add rules to the system describing conclusions that can be drawn from facts.

R1: IF HOT AND SMOKY THEN ADD FIRE
Forward chaining

1. For all rules: if their precondition holds then fire that rule - adding the post condition to the set of all known facts.
2. If no more rules can be fired – no more conclusions can be reached.
3. Difference in which order rules are fired – common approach to try ”depth-first”.
4. Conflicts if many rules can be fired simultaneously.
Example of forward chaining

Initial facts and rules:

R1: IF smoke AND hot THEN ADD fire
R2: IF alarm_beeps THEN add smoke
R3: IF alarm_beeps THEN add use_ear_plugs
R4: IF fire THEN ADD switch_on_sprinklers
R5: IF smoke THEN ADD poor_visibility
F1: alarm_beeps
F2: hot

Conclusions:

F3: smoke (R2 + F1)
F4: use_ear_plugs (R3 + F1)
F5: fire (R1 + F3,F2)
F6: switch_on_sprinklers (R4 + F5)
F7: poor_visibility (R5 + F3)
Proof trees

When we use rules to add new facts to the system we store the rule used so that a proof of the fact can be given. We can eg. construct a proof tree for the fact switch_on_sprinklers from the previous example:
Backwards chaining

Problems with forward chaining:
A lot of ”irrelevant” facts are deduced. Not ”goal-oriented”.

Idea of backward chaining:
We start with some initial hypothesis H that we want to prove.
If H is in the initial facts the finished.
Otherwise, choose some rule whose effect is H and prove all it's preconditions.

When proving a fact H we may have many choises of rules R1,....,Rn whose effect all is H. If we cannot prove the preconditions of R1 then try R2 etc.

Can be seen as a search problem.
Some issues for expert systems

- How do we represent facts that are: true, false, unknown.
- Representing uncertainty
- Reasoning about probability
- Completeness
  Can our system always probe everything that follows from the rules and the facts?
- Closed world assumption:
  If we cannot prove a fact F is that the same as F being false?
Designing an expert system

- Not all problems are suitable for expert systems. I.e. How do you define rules for determining subjective things.

- Knowledge engineering.
  Building an expert system is the task of a knowledge engineer. Has too cooperate with a domain expert to extract suitable rules. Feedback from user.

- Expert system architecture.
  How do we represent facts, reasoning about uncertainty or probability etc.

- Inference engine.
  Use existing expert system shells with suitable inference mechanism or write one youself?
Choosing a problem

Some criterias for a problem to be suitable for expert systems.

- Complex problems.
- Can be described by rules. No common sense needed.
- Cooperative experts and potential users
- Experts in short supply (we need a market!)
- Highly specialized expertize
Knowledge Engineering

- Knowledge engineer
- Example problems
- Interview expert:
  - analyze his method to solve problems.
  - Formulate rules and check validity.
- Collaboration with potential users.
- Initial prototype and iterative testing.
Problem solving method

- Forward and/or backward chaining
- Differential diagnosis
- Rules for conflict resolution. Program solving strategies.
- Uncertainty / Probabilities.
- Askable facts
- Explanation facilities
Reasoning under uncertainty

Probability theory
	vs.
Possibility theory (uncertainty)
Probability theory: needed for some advance expert systems and for machine learning.

\[ P(X) \]: probability of \( X \) being true.
\[ P(X|Y) \] - probability of \( X \) given \( Y \).

\[ P(H|E) = \frac{P(H \land E)}{P(E)} \]

Bayes theorem:

\[ P(H|E) = \frac{P(E|H) \times P(H)}{P(E)} \]

Independence: \[ P(X \land Y) = P(X) \times P(Y) \]
Using probabilities

Problem: We want to assign probabilities to facts so that our expert systems can respond with the probabilities for different facts to be true. Eg:

*How likely is it that the patient has malignant brain cancer given that he has a brain tumor?*

*How likely is it that the patient has malignant brain cancer if we know that it's 50% that he has a brain tumor?*

One simple method is to use ordinary probability theory as follows. Collect statistics in advance and compute:

\[ P(\text{Cancer} | \text{Tumor}) = 0.25, \ P(\text{Cancer} | \text{not tumor}) = 0.0 \]

Solve it:

\[ P(\text{Cancer}) = 0.50 \times 0.25 = 0.125 \]
How do we compute $P(\text{Cancer} | \text{Headache})$?

**Method 1:** Collect statistics of everyone having a headache and test how many of them has cancer.

**Problem:** Difficult to collect information, ie. We need to sample 10,000 persons with headache to find 10 person with brain cancer.

**Use bayes theorem:** $P(H | E) = (P(E | H) * P(H)) / P(E)$

**Example:** We know that there's a 50% chance of having a headache and a 0.05% chance of having brain cancer overall. We sample 100 patients with brain cancer and see that 75 of them have headache.

$P(\text{Cancer} | \text{Headache}) = 0.75 * 0.0005 / 0.50 = 0.00075$

So there's only a 0.075% chance of having cancer next time you have a hangover!
Problems with the probability theory approach

Normally there are lots of relevant evidence: symptom-1, ..., symptom-N. To compute the final probability we need lots of statistics. Eg:

\[
\begin{align*}
P(\text{Disease} \mid \text{Symptom-1 and Symptom-2 and ... Symptom-K}) \\
P(\text{Disease} \mid \text{not Symtom-1 and Symptom-2 and ... Symptom-K}) \\
P(\text{Disease} \mid \text{Symtom-1 and not Symptom-2 and ... Symptom-K}) \\
P(\text{Disease} \mid \text{not Symtom-1 and not Symptom-2 and ... Symptom-K})
\end{align*}
\]

Practically infeasible. Solve by using notion of independence. If we know that E1, ..., E-k is independent we can write:

\[
P(\text{H} \mid E1 ... E-k) = P(E1 \mid \text{H}) \times ... \times P(E-k \mid \text{H}) \times P(\text{H}) / P(E1 \text{ and } ... \text{ and } E-k)
\]
Likelihood ratios

Equivalent to reasoning about probabilities where we assume all symptoms to be independent!

The odds of an event H is defined by:

\[ O(H) = \frac{P(H)}{1 - P(H)} \]
\[ O(H \mid E) = \frac{P(H \mid E)}{1 - P(H \mid E)} \]

We define the likelihood ratio (level of sufficiency) for drawing the conclusion H given evidence E as follows:

\[ LS = \frac{P(E \mid H)}{P(E \mid \text{not } H)} \]

Using this we see that:

\[ O(H \mid E) = LS \times O(H) \]

and if we have two pieces of evidence LS1 and LS2 we just do:

\[ O(H \mid E1, E2) = LS1 \times LS2 \times O(H) \]
Expert systems based on Bayes approach

If we assume conditional independence between symptoms, just use the calculations above to get a probability. Problematic since the conditional independence usually does not hold in the real world. Gives a *false sense of accuracy*, performance deteriorates for larger systems.

Without conditional independence: collect statistics for every probability of disease given every combination of symptoms. Practically infeasible, 16 symptoms -> 65536 probability values
Mycin

One of the earliest and most influential expert systems constructed:
  • IF-THEN rules
  • Certainty factors instead of probabilities
  • Backward chaining

Performs as well as faculty members and better than tested students and physicians.

Pure Mycin not in use today because of cumbersome textual interface and limited domain.

Many other systems have been constructed with Mycin as base, some of them in clinical use today. Also for academic use, training and evaluating students.

Certainty factors are an *ad. hoc.* solution to the problem of reasoning about uncertainties.
Mycin - certainty factors

Using probabilities was not feasible. Instead, introduce a notion of certainty factors (CF) as a number between -1.0 (definitely false) and 1.0 (definitely true) where 0.0 mean unknown.

If the certainty factor of E1, E2 is denoted by CF(E1) and CF(E2) respectively we say:

\[
\begin{align*}
\text{CF(not E1)} &= -\text{CF(E1)} \\
\text{CF(E1 and E2)} &= \min(\text{CF(E1)}, \text{CF(E2)}) \\
\text{CF(E1 or E2)} &= \max(\text{CF(E1)}, \text{CF(E2)})
\end{align*}
\]

If we have a rule with certainty CF(R) whose preconditions are true with certainty CF(P) we can add the conclusion CF(C,R) with certainty:

\[
\text{CF(C,R)} = \text{CF(P)} \times \text{CF(R)}
\]

If we have two rules with the same consequent:

\[
\text{CF(C)} = \text{CF(C,R1)} + \text{CF(C,R2)} - \text{CF(C,R1)} \times \text{CF(C,R2)}
\]
Internist

- Differential diagnosis
- Disease profiles
- Disease tree
- Ad hoc methods, but great performance
- Variant QMR in use today, *aiding* physicians diagnosing diseases.
Path finder

- Combines advantage of intermist (ask only relevant questions) with sound reasoning methods.
- Comparison of different uncertainty methods. Practical study suggest that simple bayesian methods are best. (All assuming conditional independence)
- Controlling the dialogue. System displays top hypothesis and useful observations. User decides observations to give.
- Decision theory. Uses decision theory to rank the utility of different observations. Used to suggest which tests to perform
Bayesian networks

Network illustrating causal links between different conditions and their probabilities. Change in probability for one condition propagates to other conditions.

**Advantages**: We no longer need to assume independence between all symptoms. We give only a table of conditional probabilities for each node in the graph.
Path finder extended with Bayesian networks
A version of path finder using Bayesian networks exists, it's performs better than human expert pathologists!

"Anything created must necessarily be inferior to the essence of the creator."
-- Claude Shouse (shouse@macomw.ARPA)

"Einstein's mother must have been one heck of a physicist."
-- Joseph C. Wang (joe@athena.mit.edu)
Lab 3 – Constructing an expert system

In this lab you will be using a minimalistic expert system shell to create a simple expert system suitable for diagnosing problems.

- Uncertainty is handled similarly (but not identically!) to certainty factors in Mycin.
- Backwards chaining to ask relevant questions.
- Forwards chaining to propagate answers.
- All facts are symbols or negated symbols

Example:

* (prove 'car-broken)
Is the engine broken? no
Are you out of gas? debug
Currently known facts:
  (ENGINE-BROKEN -1.0 USER)
Are you out of gas? no
Sorry, cannot prove CAR-BROKEN
NIL
* *known-facts*
  ((NO-GAS -1.0 USER) (ENGINE-BROKEN -1.0 USER))
Lab 3 – Constructing a domain

To construct a simple expert system from this shell you need to:

1. Load the expert system shell *mrExpert*.
2. Add allowed questions to be asked.
3. Add rules for inferring new facts from previous facts.

**Example:**

```
(add-question 'engine-broken "Is the engine broken?"

(add-rule
 :name 'car-broken-if-engine-broken
 :consequent 'car-broken
 :certainty 1.0
 :antecedents '(engine-broken)
 :explanation "The car is broken since the engine is broken"
)

(add-rule
 :name 'engine-broken-if-overheated
 :consequent 'engine-broken
 :certainty 0.75
 :antecedents '(overheated)
 :explanation "The engine is overheated so it's probably broken"
)
```
Using negation

Negative facts are handled by either negative certainty values or by inverting the fact. Eg:

computer-broken with CF -1.0  <->  (not computer-broken) with CF 1.0
power-on with CF 1.0  <->  (not power-on) with CF -1.0

Always use positive certainty factors in the rules! Eg:

(add-rule
  :name 'not-on-if-no-power
  :consequent '(not is-on)
  :certainty 1.0
  :antecedents '((not turned-on))
  :explanation "Computer is not on since you have not turned it on"
)
The search problem for general games (chess, tic-tac-toe, othello, ...) is different in that we don't know what our opponents will do!

Some search algorithms for games:

- Minimax search
- Alpha-beta search
Example search space for some game
Minimax Search

1. Set N to be the list consisting of the element m.
2. Let n be the first node in N.
3. If n=m and n has been assigned a value, then return the value and exit.
4. If n has been assigned a value, then remove n from N.
5. If n has not been assigned a value and is a terminal node, then assign to n the value -1, 1 or 0, representing a win for the minimizer a win for the maximizer and a draw, respectively.
6. If n has not been assigned a value and all its children have been assigned a value, then assign to n the minimum or the maximum of the values of the children if n is a minimizing, respectively a maximizing node.
7. If n has not been assigned a value and its children have not all been assigned values, then add the children to the front of N.
8. Go to step 2.
Some observations about games

- Not whole game tree is known for all games, we have to stop searching after a while.
- Not all games suitable to be represented by these rules.
- Large branching factors (eg. GO).
- Evaluation functions.
- Stop conditions for searching.
- Pruning
Nontrivial games

Some most (nontrivial) games have a very large gametree we cannot "afford" to search for all possible wins and losses.

Simple stop condition – expand the tree only to depth $K$ for some $K$.

1. If we reach a node where we have won, assign the score $+\infty$.
2. If we reach a node where we have lost, assign the score $-\infty$.
3. Otherwise, use a heuristic for computing the score.

Example: For chess, compute a sum of points for taken/lost pieces and extra points for holding different "strategic" parts of the game board. Eg:

- For every rook taken $+1p$, for every rook lost $-1p$
- Taken opponent queen $+8p$, lost queen $-8p$.
- Controlling the center $+Xp$, Opponent controlling the center $-Xp$
- ...
Alpha-Beta pruning

A simple method of reducing the size of the search tree.

When doing the min-max computations we can get an initial estimate of what the result will be and avoid doing unnecessary computations. Eg:

Squares – maximizing nodes
Circles – minimizing nodes

Can we compute what the final score is without expanding node D – which is a really expansive node to search from!

Yes we can by looking at B and E.
Alpha-Beta pruning – continued

- Alpha: Best choice for a MAX node so far.
- Beta: Best choice for a MIN node so far.
- Unbound MAX nodes have alpha = -infinity
- Unbound MIN nodes have beta = +infinity
- Alpha pruning: Cut of under MIN nodes having beta value < alpha value of any of its MAX-node ancestors.
- Beta pruning: Cut of under MAX nodes having alpha value > beta value of any of its MIN node ancestors.
Alpha-Beta pruning – reformulated

General idea:
If N is a maximizing node, look at the minimum value of the siblings (V) and the maximum of the siblings of the parent of N (U). If V ≤ U prune N and all its siblings. Equivalent for minimizing case.
**Alpha-Beta pruning**

The algorithm, using an upper/lower bound of the value of all nodes.

1. Set N to be the list consists of the single starting element, m.
2. Let n be the first node in N
3. If n=m and n has been assigned a value then return the value and exit
4. Try to prune n as follows:
   - If n is a maximizing node:
     - Let v be the *minimum* of the *bounded values* of the siblings of n.
     - Let u be the *maximum* of the *bounded values* of the siblings of all ancestors of n that are minimizing nodes.
     - If v \( \leq \) u then remove n and all its siblings and any successors of n and their siblings from N. Also back up the value of the parent of n.
   - Otherwise: equivalent case with min replaced by max and \(<\) with \(>\).
5. If N cannot be pruned, then if n is a terminal node or we decide not to expand n, assign n the value determined by the *evaluation function* and back up the value of n and remove n from N.
6. Otherwise, remove n from N. Add the children of n to the front of N and assign the children initial values so that maximizing nodes are assigned -\(\infty\) and minimizing nodes +\(\infty\).
Alpha-Beta pruning – backing up

How do we compute the upper / lower bound of the values of all nodes?

1. Let $v$ be the current value of $n$.
2. Let $m$ be the parent of $n$ and $u$ the current value of $m$.
3. If $m$ is a maximizing node then set the value of $m$ to $\max(u,v)$
4. If $m$ is a minimizing node then set the value of $m$ to $\min(u,v)$
5. If $m$ is the root node or not all children of $m$ have backed up their values then quit. Else back up the value at $m$. 
Week 47

Lectures week 47:
(Expert Systems)
(Games)
Agents
Structured knowledge representation – semantic networks and frames

Lectures week 47 - 48:
Proposistional logics
Planning

Reading instructions:
Chapter 2 and Chapter 8 in Cawsey

Labs:
Lab 4 – Vacuum cleaning; Create a (simulated) agent for vacuum cleaning.
You have three weeks to complete this lab
Agents

- What an agent does
- How it is related to its environment
- How it is evaluated
- How we might go about building one
- Examples of agents
Definitions of agents

Russel and Norvig:
An agent is anything that can be viewed as **perceiving** its environment through **sensors** and **acting** upon that environment through **effectors**.

Franklins definition:
An autonomous agent is a system situated within and as part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.
Rational agents

Our interest is rational agents, that is agents that do the "right things", are successful.

What does it mean for a rational agent to be successful?

The performance measure is the criterion that determines how successful an agent is.

Problems:

- Different criteria of success
- An agent can act rationally, but be unsuccessful due to lack of information.
Rational agents

What is rational depends on:
  • The performance measure
  • The percept sequence
  • What an agent knows about the environment
  • The actions that the agent can perform

An ideal rational agent should do whatever action is expected to maximize it's performance measure, on the basis of the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

Not forgetting to do actions in order to obtain useful information!
An Agents structure

Agent = Architecture + Program

What do we need to know to design an agent program?

- Possible percepts and actions
- Goals or performance measure the agent is supposed to achieve
- Environment it will operate in
Taking the right action

An agent can be described by a mapping from percept sequences to actions.

Specifying which action an agent ought to take in response to any given percept sequence provides a design for an ideal agent.

Why not keep a table in memory and look up the answers?

- Size of the table very large, infinite in general, unless we put a bound on the length of the percept sequences.
- It would take quite a long time for the designer to build the table.
- Lack of generallity: the agent has no autonomy and if the environment changes it would be lost.
- Even if we give the agent a learning mechanism as well, it would take forever to learn the right value for all the table entries!
Example: A Vacuum-cleaner agent

- Percepts: A three element percept vector
  [bumping, dirt presence, home location]
- Actions:
  go-forward, turn left/right 90 degrees, suck up dirt, turn off.
- Goals: clean up and go home
Simple reflex agent

The agent has a number of **condition-action** rules.

Example of rule: ”If bumped then turn left”

Every time the agent receives an input:

- Process the input
- Find a rule to apply
- Apply the rule
Agents that keep track of the world

In some cases the correct decision cannot be made just on the basis of the current percept.

For example, the decision about where to move next in the vacuum-cleaning agent can be based on which parts of the room the agent has already cleaned.

An **internal state** is required where the agent records some of the past percepts / actions.

Eg: store my current position by counting number of steps taken, store map of area which have been explored etc.
An imperfect world

If the sensors give incomplete information about the world, updating the internal state also requires:

- Information about how the world evolves independently of the agent.
- Information about how the agent's actions affect the world.
Goal-based agents

A **goal** describes the situations that are desirable.

The agent can combine the information about the goal with the information about the results of possible actions in order to choose actions that achieve the goal.

A goal-based agent can be less efficient, but it is more flexible.

The techniques used are **search** and **planning**.

**Problem:** Goals just provide a crude distinction between "happy" and "unhappy" states.
Utility-based agents

Utility is a function that maps a state of the world onto a real number, which describes the associated "degree of happiness" of the agent. A utility function is specially important when:

- The agent has conflicting goals -> tradeoff between goals.
- The agent has several goals, none of which can be achieved with certainty -> likelihood of success weighted up against importance of the goals.
Environments

- **Accessible vs. Inaccessible:** accessible if the sensors give the complete state of the environment.

- **Deterministic vs nondeterministic:** Deterministic if the next state of the environment is completely determined by the current state and the actions selected by the agent.

- **Static vs. Dynamic:** Dynamic if the environment can change while the agent is deliberating.

- **Discrete vs. Continous:** Discrete if there are a limited number of distinct, clearly defined percepts and actions.
The simulator takes one or more agents as input and gives each agent the right percepts and receives back actions.

The simulator then updates the environment based on the actions, and possibly other dynamic processes in the environment.

An **evaluation function** can also be present in the simulator that applies a performance measure to the agents and returns the resulting scores.
Example: The Electric Cow

A simulated world consisting of agent cows living on a (rectangular) field.

- Each square in the field can contain one of "electric-fence", "grass", "mud" or another "cow".
- Every time step the cows can sense in the directions left, front and right and receive the content of the square in that direction. Eg, a vector '(grass mud mud) indicates grass on the left and mud front and right.
- They also sense a binary variable "pain" which is true if they walk into a fence or another cow and a variable "hungry" which is true if they have digested their last food.
- The cows can take the actions "move-forward", "move-left", "move-right", "eat" and "digest" (produces fertilisers!).
- The cows get +1 happiness if they are hungry and eat. Since they are social creatures they get +1 happiness if they digest when they are standing next to another cow. They lose -1 happiness whenever they feel pain.

Produce a reactive agent program that maximises the cow's happiness.
About lab 4

Your assignment in this laboration is to make a vacuum agent. The agent should be able to remove all the dirt in a rectangular world of unknown dimensions with a random distribution of dirt and obstacles. When the world is cleaned the agent should return to the start position and shut down.

The environment simulator

- The agents perceive the world.
- The agents decide on their actions
- The actions are executed
About lab 4

(load "basic-env.lisp")
(load "grid-env.lisp")
(load "moves.lisp")
(load "vacuum.lisp")
(load "tools.lisp")

(setq *map*
   (make-vacuum-world
        :agents (list (vacuum-agent 'Reactive 'reactive-vacuum-moves)))
)

(print-map *map*)
About lab 4

(run-map *map* 10 t)
Time step 1:
Agent REACTIVE=-100 perceives (NIL NIL HOME)
   and does FORWARD
   ---------------------------------------------
  7 ! # ! # ! # ! # ! # ! # ! # ! # ! # !
  |---|---|---|---|---|---|---|---|---|
  6 ! # ! ! >! * ! ! * ! * ! # !
  |---|---|---|---|---|---|---|---|---|
  5 ! # ! ! * ! ! ! ! ! * ! # !
  |---|---|---|---|---|---|---|---|---|
  4 ! # ! ! ! ! ! ! ! ! ! ! # !
  |---|---|---|---|---|---|---|---|---|
  3 ! # ! ! ! ! ! ! * ! ! ! ! # !
  |---|---|---|---|---|---|---|---|---|
  2 ! # ! ! ! ! ! ! * ! ! ! ! # !
  |---|---|---|---|---|---|---|---|---|
  1 ! # ! ! ! ! ! ! * ! ! ! ! # !
  |---|---|---|---|---|---|---|---|---|
  0 ! # ! ! ! ! ! ! ! ! ! ! ! ! !
  |---|---|---|---|---|---|---|---|---|

Time step 2:
Agent REACTIVE=-902 perceives (NIL DIRT NIL)
   and does SUCK
   ---------------------------------------------
  7 ! # ! # ! # ! # ! # ! # ! # ! # !
  |---|---|---|---|---|---|---|---|---|
  6 ! # ! ! >! ! ! ! ! * ! ! ! # !
  |---|---|---|---|---|---|---|---|---|
  5 ! # ! ! ! ! ! ! ! ! ! ! ! ! ! # !
  |---|---|---|---|---|---|---|---|---|
  4 ! # ! ! ! ! ! ! ! ! ! ! ! ! ! # !
  |---|---|---|---|---|---|---|---|---|
  3 ! # ! ! ! ! ! ! * ! ! ! ! ! ! ! # !
  |---|---|---|---|---|---|---|---|---|
  2 ! # ! ! ! ! ! ! * ! ! ! ! ! ! ! # !
  |---|---|---|---|---|---|---|---|---|
  1 ! # ! ! ! ! ! ! * ! ! ! ! ! ! ! # !
  |---|---|---|---|---|---|---|---|---|
  0 ! # ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !
  |---|---|---|---|---|---|---|---|---|

Artificial Intelligence
HT2005
Structured knowledge representation

Knowledge representation and reasoning is a core element of artificial intelligence. Information can be represented in different ways:

- Semantic networks
- Frames
- Propositional logics
- Predicate logics
- ...
Semantic networks

A semantic network is a directed graph in which the nodes correspond to classes, instances and the possible values for attributes. The labeled arcs in a semantic network encode instance and subclass relationships as well as possible attributes describing classes.

Object oriented programming is based on semantic networks!
Semantic network

subclass(Football-Team, Team)
subclass(Large-Team, Team)
has-member(Team, Human)
has-member(IDA FF, Anders)
has-member(IDA FF, Sture)

instance(IDA FF, Large-Team)
instance(IDA FF, Football-Team)
instance(Anders, Human)
instance(Sture, Human)
If we need to express non-binary relationships we can use auxiliary nodes as follows:

Semantic network

\[
\text{event(IDA FF, Loosers, football-game, 5-1)}
\]
Relationships in Semantic networks

Subclasses: subset(A,B)
   Cats are animals (A = Cats, B = Animals)

Instances: is-a (A,B)
   Bill is a cat (A = Bill, B = Cats)

Relationships between objects and objects/values - R(a,b):
   Bill has age 12 (a = Bill, b = 12, R = has age)

Relationships between class and object value – forall x in A : R(x,b)
   All birds have two legs (A = Bird, B = 2, R = Has-Legs)

Between classes: forall x in A, exists some y in B: R(x,y)
   All birds have a bird as a parent (A = Bird, B=Bird, R=Has parent)
Frames

A frame is a collection of attributes (slots) and associated values (and possibly constraints on values) that describe some entity in the world.

Example

Team;
  * has-member: Human

Football-team;
  is-a: Team

Large-team;
  is-a: Team

IDA-FF;
  instance-of: {Large-team, Football-team}
  has-member: {Anders, Sture, Ola}
Inheritance

• Proper inheritance
  Slots and values are inherited by subclasses and instances

• Inheritance with exceptions
  Overriding values given from super classes

• Multiple inheritance
  Inheriting values recursively from multiple parents
  Problem with order of inheritance
Example 2

Animal:
   Flies: False
   Alive: True
Mammal:
   is-a: Animal
   Legs: 4
Cats:
   is-a: Mammal
Bird:
   is-a: Animal
   Flies: True
   Wings: True
Penguin:
   is-a: Birds
   Flies: False
Bill:
   instance-of: {Cat}
   Friends: {Opus}
Opus:
   instance-of: {Penguin}
   Friends: {Bill}
Meta slots

Slots can themselves be slots, eg:

Spotty:
  instance-of: Dog
  Owner: Mathias
Owner:
  value: Person
  inverse: Owns

Conclusions:

Person(Mathias)
Owns(Mathias, Spotty)

Many frame systems allow slots to include procedures (*procedural attachments*) that is executed when a value is added/changed.
Planning

Implementing goal and utility oriented agents.

- Representation of actions
- Representation of states
- Representation of goals
- Representation of plans

Idea:
- Open up representation of states, goals and actions.
- Add actions to a plan whenever needed.
- Most parts of the world are independent of other parts.
- Solve through eg. Search
Desirable search procedures

- Systematic search procedure
- Sound search procedure
- Complete search procedure
State space search

State variables:

door-open, door-closed, light-on, light-off, ...

state s:

\[ s(\text{door-open}) = \text{True}, \quad s(\text{light-on}) = \text{False} \]
\[ s(\text{door-closed}) = \text{False}, \quad s(\text{light-off}) = \text{False} \]

goal g:

\[ g(\text{door-open}) = \text{True} \]

Action close-door:

\text{if door-open = True then closing the door will set door-open to false and door-closed to true and leave all other truth assignments as they are.}
Operators

For all actions/operators we need to specify:

- **P** - Preconditions. Facts that have to be true to perform action
- **A** - Additions. Facts that will be true after performing action.
- **D** - Deletions. Facts that will be false after performing action.

State progression:

We can apply an operator \((P,A,D)\) to a state \(S\) if all facts in \(P\) is true in \(S\).

The resulting state is \(A + (S – D)\)

Goal regression:

We can apply an operator \((P,A,D)\) in the backwards direction from a goal \(G\) if no facts in \(D\) are true in \(G\). The resulting goal is \(P + (G – A)\)
Choosing operators

Means/End analysis:

Prefer to apply operators in a state so as to reduce the number of differences between the current state and the goal.

Example:

$S(\text{at-me}) = \text{my-room}$

$S(\text{door-open}) = \text{true}$

$S(\text{light-on}) = \text{true}$

$G(\text{light-on}) = \text{false}$

$G(\text{at-me}) = \text{my-bed}$

Action 1: close-door

Action 2: turn-off light
Blocks world domain

- on(A,B)
- on(B,table)
- on(C,table)
- clear(A)
- clear(C)
Operators schema

Move block \(?x\) from \(?y\) to the table:

P: on(?x,?y), clear(?x)
A: on(?x, table), clear(?y)
D: on(?x,?y)

Move block \(?x\) from block \(?y\) to block \(?z\)

P: on(?x,?y), clear(?x), clear(?z)
A: on(?x,?z), clear(?y)
D: on(?x,?y), clear(?z)

...

We can solve the problem by doing search over the search space.
Plan-space search

**Basic idea:** instead of searching over possible states of the world we search over all possible plans until we have found a suitable plan.

- Plan = set of sequences of operators
- Partially ordered plans
- Search in space of all possible plans
- Refinement and modification
- Least commitment
Example of partially ordered plan
Plan space search

A plan consists of

- steps (corresponding to the operators)
- constraints (on the order the steps are performed)
- dependency information (requirements, links and conflicts)

Requirements:
Step and condition that must be true immediately prior to the step

Links:
Condition + producer step (having condition in A) + consumer step (having condition in P)

Conflicts:
Condition + clobberer step (having condition in D)
Plan refinement

Add constraints to eliminate conflict:

Constrain the clobberer to come before the producer or after the consumer of the threatened link.

Linking steps to eliminate requirements:

Eliminate requirement with condition $r$ for step $q$ by adding link $L$ with producer $P$, consumer $q$ and condition $r$ where $p$ is a new or existing step that adds $r$.

Note: A link implies an ordering.
Desired properties of the new plans

A new plan must have all the steps, constraints, and links of the plan it was generated from.

If, in a given plan, there is a link $L$ with producer $P$, consumer $q$, and condition $r$, then there must be corresponding steps $P$ and $q$ such that $P$ adds $r$ and $q$ has $r$ as a precondition.

If, in a given plan, there is a link $L$ with producer $P$, consumer $q$, and condition $r$ and a step $c$ that (given the current set of constraints) could come between $P$ and $q$ and deletes $r$: then there is a conflict with link $L$ and clobberer $c$. 
start on(C,A), on(A,table), on(B,table), clear(B), clear(C)

goal on (A,B), on(B,C)

(1) P: on(C,A), clear(C)
    A: on(C,table), clear(A)
    D: on(C,A)

(2) P: on(B,table), clear(B), clear(C)
    A: on(B,C)
    D: on(B,table), clear(C)

(3) P: on(A,table), clear(A), clear(B)
    A: on(A,B)
    D: on(A,table), clear(B)
Artificial Intelligence
HT2005
Task reduction planning

Plan on first level using tasks.

Reduction: Replace a task in the plan with (a sequence of) more specific tasks.

The reduction goes on until all tasks are specified in sufficient detail.
Adapting previously generated plans

Idea:
- Method to reuse old plans
- Have a library of old plans
STRIPS: One of the first planners

(strips-op move-block-to-table
  :preconditions ((clear ?x)(on ?x ?y))
  :add-list ((on ?x table)(clear ?y))
  :delete-list ((on ?x ?y))
  :command-string "move ?x from ?y to the table"
  :variables (?x ?y)
)

;setq initial-state
'((block a) (block b) (block c) (block d)
  (clear a)
  (on a b) (on b table) (clear c) (on c d)
  (on d table))
(setq goal '((on a b) (on b d)))

(strips initial-state goal)
=>
((MOVE-BLOCK-TO-TABLE A B) (MOVE-BLOCK-TO-TABLE C D)
 (MOVE-BLOCK-FROM-TABLE B D) (MOVE-BLOCK-FROM-TABLE A B))
STRIPS: One of the first planners

Strips uses a version of depth first search:

- Inefficient, needs loop elimination. Gives suboptimal solutions, eg:

```
(strips initial-state '((on a c)(on b d)))
=>
((MOVE-BLOCK-TO-BLOCK A B C) (MOVE-BLOCK-TO-BLOCK A C B)  
(MOVE-BLOCK-TO-TABLE C D) (MOVE-BLOCK-TO-BLOCK A B C)  
(MOVE-BLOCK-FROM-TABLE B D))
```
Week 48

This week:

- Propositional logic: syntax, semantics, deduction and resolution.
- Predicate logic: syntax, semantics and deduction.

This week / next week:

- Natural Language Processing
- Machine Learning

- Chapter 5 in Cawsey – Natural language processing
- Chapter 7 in Cawsey – Machine learning and Neural networks
In ordinary language, logic is the reasoning used to reach a conclusion from a set of assumptions. More formally, logic is the study of inference—the process whereby new assertions are produced from already established ones.

[ wikipedia.org ]

Many different logics exist.

- Syntax
- Semantics

Different inference mechanisms for different logics

- Soundness (truth-preserving)
- Complete
Logic

Interpretation: Truth assignment to propositions, a "possible world".
Valid sentence: True in every interpretation
Satisfiable sentence: True in some interpretation
Unsatisfiable sentence: True in no interpretation
Propositional logic - Syntax

(sats-logik)

- Propositional symbols (p,q,r,s, ....)

- Connectives: \( \land, \lor, \neg, \rightarrow, \leftrightarrow \)

- Parentheses: (, )

- No other symbols.
Well-formed formula

(wff = well-formed formula)

- Propositional symbols are wffs

- If $\alpha$ and $\beta$ are wff then

  $\neg \alpha$ is wff

  $\alpha \land \beta$ is wff

  $\alpha \lor \beta$ is wff

  $\alpha \rightarrow \beta$ is wff

  $\alpha \leftrightarrow \beta$ is wff

- No other wffs.
Propositional logic – Semantics

Interpretation:

An interpretation for propositional logic is an assignment to each propositional variable either the value true or false and to each propositional connective it's "usual" truth function meaning.

\[-\alpha \text{ is true iff } \alpha \text{ is not true}\]

\[\alpha \land \beta \text{ is true iff } \alpha \text{ and } \beta \text{ both are true}\]

\[\alpha \lor \beta \text{ is true iff at least one } \alpha \text{ and } \beta \text{ is true}\]

\[\alpha \rightarrow \beta \text{ is true iff } \alpha \text{ is not true or } \beta \text{ is true}\]

\[\alpha \leftrightarrow \beta \text{ is true iff}\]

\[\alpha \rightarrow \beta \text{ is true and } \beta \rightarrow \alpha \text{ is true}\]
Propositional logic – truth tables

Truth table for the logical connectives of propositional logic.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Propositional logic- example 1

Use a truth table to determine in which interpretations the statement below is true:

\[(p \rightarrow q) \land p \rightarrow q\]

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p → q</th>
<th>(p → q) ∧ p</th>
<th>((p → q) ∧ p) → q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Thus we see that the statement above is a *valid sentence* (since it is true in every interpretation).
Inference rules for propositional logic

1. **And introduction**: \( A, B \vdash A \land B \)

2. **And elimination**: \( A \land B \vdash A, B \)

3. **Or introduction**: \( A \vdash A \lor B \)

4. **Or elimination**: \( A \rightarrow C, B \rightarrow C, A \lor B \vdash C \)

5. **If introduction**: 
   \( (A, B, \ldots \vdash X) \vdash (A \land B \land \ldots) \rightarrow X \)

6. **If elimination (Modus Ponens)**: 
   \( A \rightarrow B, A \vdash B \)

7. **False introduction**: \( A, \neg A \vdash \bot \)

8. **False elimination**: \( \bot \vdash A \)

9. **Reductio ad Absurdum**: \( (\neg A \vdash \bot) \vdash A \)

10. **Negation introduction** \( A \vdash \neg \neg A \)

11. **Negation elimination** \( \neg \neg A \vdash A \)

12. **Case elimination** \( (A \lor B), \neg A \vdash B \)

13. **Neg. Or intro.** \( \neg A, \neg B \vdash \neg (A \lor B) \)

14. **Neg. Or elim.** \( \neg (A \lor B) \vdash \neg A \)
Propositional logic – example 2

AI is not easy or computers are sentient.
If AI is hard then AI is fun.
Computers are not smart.

\( \neg E \lor S, \neg E \rightarrow F, \neg S \)

- Case elimination: \( \neg E \lor S, \neg S \vdash \neg E \)
- If elimination: \( \neg E \rightarrow F, \neg E \vdash F \)

Result: AI is fun!
Conjunctive Normal Form

Aim: Conjunction of disjunctions of literals

1. eliminate $\rightarrow$

$$(p \rightarrow q) \equiv \neg p \lor q$$

2. push negation inwards

$$\neg \neg p \equiv p$$

$$\neg (p \land q) \equiv \neg p \lor \neg q$$

$$\neg (p \lor q) \equiv \neg p \land \neg q$$

3. distribute

$$p \lor (q \land r) \equiv (p \lor q) \land (p \lor r)$$

$$p \land (q \lor r) \equiv (p \land q) \lor (p \land r)$$

4. flatten

$$p \lor (q \lor r) \equiv (p \lor q) \lor r \equiv p \lor q \lor r$$

$$p \land (q \land r) \equiv (p \land q) \land r \equiv p \land q \land r$$
Resolution

If we have all propositions in conjunctive normal form we can use the resolution rule for all deduction needed to prove facts.

\[
A_1 \lor \ldots \lor A_i \lor C \lor A_{i+1} \lor \ldots \lor A_n \\
B_1 \lor \ldots \lor B_j \lor \neg C \lor B_{j+1} \lor \ldots \lor B_m
\]

\[\Rightarrow\]

\[
A_1 \lor \ldots \lor A_n \lor B_1 \lor \ldots \lor B_m
\]

Example:

\[X \lor \neg Y,\]
\[Y \lor Z\]

\[\Rightarrow\]

\[X \lor Z\]
Resolution in predicate logic

To prove that P holds with respect to a set of axioms F, do:

1. Convert all propositions in F into CNF
2. Negate P and convert the result into CNF, add this to the result of 1.
3. Repeat until a contradiction is found:
   a) select two clauses
   b) resolve the two clauses
   c) if the resolvent is empty, then a contradiction is found. Otherwise, add the resolvent to the set of clauses available to the procedure.

If we find contradiction then P must be true.

This algorithm is sound.
Predicate logic

- Predicate symbols
- Function symbols
- Constant symbols
- Variable symbols
- Connectives: $\land$, $\lor$, $\neg$, $\rightarrow$, $\leftrightarrow$
- Parentheses: $(, )$
- Quantifiers: $\forall$, $\exists$
Predicate logic – syntax

Terms:
- Constants and variables are terms.
- $f(t_1, \ldots, t_N)$ are a term if $f$ is a function or arity $N$ and $t_1, \ldots, t_N$ are terms.

Atomic sentence:
$p(t_1, \ldots, t_N)$ is an atomic sentence where $p$ is a predicate arity $N$ and each $t_i$ is a term.
Predicate logic – well formed formulas

- Atomic sentences are wffs

- If $A$, $B$, $A_1$, ..., $A_n$ are wffs then

  $\neg A$ is wff

  $A_1 \land ... \land A_1$ is wff

  $A_1 \lor ... \lor A_1$ is wff

  $A \rightarrow B$ and $A \leftrightarrow B$ are wffs

  $\forall x_1, ..., x_n: A$ and $\exists x_1, ..., x_n: A$

  are wffs

- No other wffs.
Quantifiers

- scope of quantifier

- closed formula

- ground term

- quantification rules:
  \[ \forall x: A \text{ is equivalent to } \neg (\exists x: \neg A) \]
  \[ \exists x: A \text{ is equivalent to } \neg (\forall x: \neg A) \]
Translating english sentences into predicate logics

Some examples:

1. All purple mushrooms are poisonous.
2. No purple mushrooms are poisonous.
3. All mushrooms are purple or poisonous.
4. All mushrooms are either purple or poisonous, but not both.
5. There are exactly two purple mushrooms.
6. All purple mushrooms except one are poisonous.
Interpretation

An interpretation $I$ for predicate logic is a triple $(D, M_1, M_2)$ where $D$ is a domain, $M_1$ a mapping that maps terms to $D$ and $M_2$ a mapping that maps each predicate symbol of arity $n$ to a set of $n$-tuples in $D$. 
An interpretation $I$ is a model of a wff $\phi$ ($I \models \phi$) under the following conditions.

$I \models p(t_1, \ldots, t_n)$ iff

$< M_1(t_1), \ldots, M_1(t_n) > \in M_2(p)$

$I \models (\phi_1 \land \phi_2)$ iff $I \models \phi_1$ and $I \models \phi_2$

$I \models (\phi_1 \lor \phi_2)$ iff $I \models \phi_1$ or $I \models \phi_2$

$I \models \neg \phi$ iff $I \not\models \phi$

$I \models \forall x: \phi$ iff $I$ is a model for every formula obtained by substituting any domain element for the quantified variable in $\phi$.

$I \models \exists x: \phi$ iff there is a domain element such that $I$ is a model for the formula obtained by substituting that domain element for the quantified variable in $\phi$. 

Predicate logic – resolution

Basic idea same as for propositional logic.

To prove that P holds with respect to a given set of facts F, do:

1. Convert all statements in F into CNF
2. Negate P and convert the result into CNF, add to 1.
3. Repeat until a contradiction is found:
   a) Select two clauses
   b) Resolve the two clauses
   c) If the resolvent is empty then add a contradiction. Otherwise add the resolvent to the set of clauses under consideration.

The only new part is to decide how to convert predicate logic expressions into CNF and how to resolve them.
Predicate logics – Conjunctive normal form

1. Eliminate →

\[ p \to q \text{ becomes } \neg p \lor q \]

2. Move \( \neg \) inwards

\[ \neg \neg p \text{ becomes } p \]
\[ \neg (p \lor q) \text{ becomes } \neg p \land \neg q \]
\[ \neg (p \land q) \text{ becomes } \neg p \lor \neg q \]
\[ \neg (\forall x: p) \text{ becomes } \exists x: \neg p \]
\[ \neg (\exists x: p) \text{ becomes } \forall x: \neg p \]

3. Standardize variables.

4. Move quantifiers left.

5. Eliminate existential quantifiers –
Skolemize.

\[ \exists y: \text{President}(y) \text{ becomes President}(S1) \]
\[ \forall x: \exists y: \text{has-parent}(x, y) \text{ becomes } \forall x: \text{has-parent}(x, F1(x)) \]

6. Drop universal quantifiers.

7. Distribute \( \land \) over \( \lor \).

\[ (p \land q) \lor r \text{ becomes } (p \lor r) \land (q \lor r) \]

8. Flatten - use associativity for \( \lor \) and \( \land \).

(9. Create a separate clause corresponding to each conjunct.)
Unification

Two formulas match if we can find substitutions for the variables appearing in the formulas such that the two become syntactically equivalent. (Use the *most general* substitution).

The substitution algorithm for $L_1$, $L_2$

1. Set SUBST to nil
2. If $L_1$ or $L_2$ is a variable or a constant, then:
   1. If $L_1$ and $L_2$ are identical, then return success.
   2. If $L_1$ is a variable then: if $L_1$ occurs in $L_2$ return fail, otherwise set SUBST to $\{L_2/L_1\}$ and return success. Equivalent case if $L_2$ is a variable.
   3. FAIL.
3. If the initial predicate symbol of $L_1$, $L_2$ are not identical then FAIL.
4. If $L_1$, $L_2$ have different number of arguments then FAIL.
5. For $i=1$ to the arity of $L_1$, do:
   1. Unify $i$:th argument of $L_1$ with $i$:th argument of $L_2$
   2. If unification fails then FAIL.
   3. Apply resulting substitution to remainder of $L_1$, $L_2$ and concatenate with SUBST.
6. Success
Predicate logics – example

Jack owns a dog
Every dog owner is an animal lover
No animal lovers kills an animal
Either Jack or Curiosity killed the cat, who is named Tuna.

Did Curiosity kill the cat?
Decidability

Predicate calculus is undecidable!
Ie. there is no effective method for deciding whether a given formula is a theorem).

Some subsets of Predicate calculus are decidable. eg. Horn-clauses (clauses with at most one positive literal).

Resolution is *refutation complete*.

If a set of sentences are unsatisfiable, then a contradiction is found. Resolution cannot be used to generate all logical consequences of a set of sentences, but it can be used to *establish* that a given sentence is a logical consequence.

Example:

\[ P(x) \text{ or not } P(f(x)), \text{ not } P(a) \]
Prolog

Prolog is a programming language based on predicate logics. In prolog all statements are written in an implication normal form. Eg:

\[
\begin{align*}
\text{ancestor}(X, Y) & : = \text{father}(X, Y). \\
\text{ancestor}(X, Y) & : = \text{father}(X, Z), \ \text{ancestor}(Z, Y). \\
\text{father}(\text{kalle}, \text{erik}). \\
\text{father}(\text{erik}, \text{lars}).
\end{align*}
\]

Prolog uses a version of resolution to solving questions. For instance:

\[
\begin{align*}
\text{ancestor}(X, \text{lars}) & \quad \text{gives } X=\text{erik} \text{ or } X=\text{kalle} \\
\text{ancestor}(\text{kalle}, Y) & \quad \text{gives } Y=\text{erik} \text{ or } Y=\text{lars}
\end{align*}
\]

By knowing in which order prolog will try to prove clauses we can influence in what order different parts of a "program" will be executed.
Natural language processing

Why:
- More natural and free communication between man and machine
- Interpret and process information that is stored as natural language.

Advantages:
- People know their natural language
- No need to learn a formal language
- No need to translate a query into an artificial language

Disadvantages:
- Computationally expensive, slow
- Difficult, not completely sure.
Applications

1. Spelling and grammar checkers
2. Spoken language control systems
3. Automatic message understanding
4. Machine translation tools
Aspects of natural language processing

**Speech recognition**: Translating spoken language into a sequence of words.

**Syntactic analysis**: Extracting the grammatical structure of sentences.

**Semantic analysis**: What does the sentence (by itself) mean?

**Pragmatic analysis**: What does the sentence mean *in this context*.

Eg:

-- Do you know what time it is?
-- Yes!
Language components

**Spoken language:** phonems

**Content words:**
- Nouns
- Adjectives
- Verbs
- Adverbs

**Function words:**
- Determiners
- Quantifiers
- Prepositions
- Connectives

**Phrases:**
- Noun phrases, Adjective phrases, Verb phrases, ..
Component steps of communication

Speaker: Intention
Speaker: Generation
Speaker: Synthesis
Hearer: Perception
Hearer: Analysis
Hearer: Disambiguation
Hearer: Incorporation
Speech recognition

- Frequency spectogram
- Phonemes
- Template matching
- Statistical information

Some problems:
- Different pronunciations
- No gaps between words
- Background noise

Restrictions:
- Single speaker vs. speaker independent
- Individual word vs. continuous speech recognition.
- Degree of restriction in vocabulary.
Syntax: rewrite systems

EXPR → NUMBER
EXPR → VARIABLE
EXPR → (EXPR + EXPR)
EXPR → (EXPR * EXPR)
Context-Free grammars

A grammar consisting solely of rules that have a single symbol on the left side is a context-free grammar.

Context-sensitive grammar.

Recursively-enumerable grammar

What is natural language?
Parsing Context-free grammars

Context free grammars can be parsed with a number of techniques eg. top-down, bottom-up, LR-n parsing.

Example of context free grammar for english:

\[
S \rightarrow NP \ VP \\
VP \rightarrow V \ NP \\
NP \rightarrow NAME \\
NP \rightarrow DET \ N \\
DET \rightarrow a | the \\
V \rightarrow ate | saw \\
N \rightarrow cat | mouse \\
NAME \rightarrow Sue | Zak
\]
Parsing Context-Free grammars

Parsing a context-free grammar can be done through search. Each state is a tuple consisting of a list of symbols remaining to be expanded and a list of remaining words.

$$((\text{symbol}_1 \ldots \text{symbol}-N) \ (\text{word}_1 \ldots \text{word}-M))$$

The successor states are generated as follows:
If symbol1 can be expanded to word1:

$$((\text{symbol}_2 \ldots \text{symbol}-N) \ (\text{word}_2 \ldots \text{word}-M))$$

Otherwise, for each rule
symbol1 -> symbol1' \ldots symbolS'

generate a successor state:

$$((\text{symbol}_1' \ldots \text{symbol}_S' \text{symbol}_2 \ldots \text{symbol}-N) \ (\text{word}_1 \ldots \text{word}-M))$$

The goal state is a state with no remaining symbols or words left.
Problems

Some sentences are ambiguous:

The man saw the girl with the hat.
The man saw the girl with the binoculars.

The following sentences are all grammatically correct!

The boy sees the idea
A boy saw the pizza
I is a man
Sue sighed the pizza

Being grammatically correct does guarantee that the sentence makes sense!
Semantics

Compositional semantics

Predicate Logic language

The goal of semantic interpretation is to compute the logical forms for a sentence.

Example:

Fido barks.
=> (BARKS FIDO)

The boy saw a girl
=> (EXISTS G (AND (GIRL G) (SAW BOY-1 G)))
Pragmatic

- Speech act: informing, requesting, promising, suggesting, etc.
- Speech planning.
- Plan recognition.
- Previous sentences provide context to later sentences.

Example:

The man saw the boy.
He was looking through a pair of binoculars.

Did the boy or the man have the binoculars? Who are we referring to in the second sentence?
Ambiguity

Homophones:
  bear, bare

Lexical ambiguity:
  same word, different syntax category.

Syntactic ambiguity:
  The man saw the girl with the hat.
  The man saw the girl with the binoculars.

Semantic ambiguity:
  Same word, different meanings (e.g. bank).

Referential ambiguity:
  He, She, It, ...

Pragmatic ambiguity:
  Do you know what time it is?
Disambiguation

World model
Mental model
Language model
Acoustic model

The ruler likes the house.

Diagram:

```
OBJECT
  ANIMATE
    PERSON
      RULER-PERSON
  INANIMATE
    TOOL
      RULER-TOOL
```
Generation

What to say?
Canned responses or text planning.
Sentence production.
Speech synthesis.
Machine learning

Pattern matching:
   Classify data into categories.
   Eg: recognise faces from images, words from sound, engine problems from sensors, email spam, ...

Supervised learning:
   Train the system using examples.

Unsupervised learning:
   Learn to recognise categories as you go.

Online:
   Interactive training / usage.

Batch learning:
   Give examples and let system train itself noninteractively.
Machine Learning

Decision trees
Version spaces
Genetic algorithms
Neural nets
...

Artificial Intelligence
HT2005
Concept learning

Given:
- A language for examples: Le
- A language for concepts: Lc
- The covers relationship between Le and Lc
- A set of positive examples: P
- A set of negative examples: N
- A quality criterion Q taking into account P and N.

Find:
- A concept description C from Lc satisfying the quality criterion.
Version spaces

Idea:
Keep a representation of all hypotheses that cover all positive examples and do not cover any negative examples up until now.

Update the representation for each new example.

Operations:
A concept C1 is more general than a concept C2 if C1 covers C2.
A concept C1 is more specific than a concept C2 if C2 covers C1.

Generalization by dropping conditions.
Specialization by adding conditions.
Example: Toy robots

Attributes

Is-smiling: yes-s / no-s
Has-tie: yes-t / no-t
Holding: sword, balloon, flag

Classes:

Friendly, unfriendly.

Very simple language of categories: A list of attributes that must hold for all objects in that category. Example:

friendly = (YES-S, BALLOON)

Existance of and quality of solution depenends on the language of categories!
<table>
<thead>
<tr>
<th></th>
<th>Is-similing</th>
<th>Holding</th>
<th>Has-Tie</th>
</tr>
</thead>
<tbody>
<tr>
<td>friendly</td>
<td>yes</td>
<td>balloon</td>
<td>yes</td>
</tr>
<tr>
<td>unfriendly</td>
<td>yes</td>
<td>sword</td>
<td>no</td>
</tr>
<tr>
<td>friendly</td>
<td>yes</td>
<td>flag</td>
<td>yes</td>
</tr>
<tr>
<td>unfriendly</td>
<td>no</td>
<td>sword</td>
<td>yes</td>
</tr>
</tbody>
</table>

How do we categorise all friendly robots?
Version spaces

\( G = \) boundary set of concepts that are more general than the concept to learn.
\( S = \) boundary set of concepts that are more specific than the concept to learn.

Everything between \( G \) and \( S \) must be consistent with the examples.

Algorithm: versionSpaces

1. \( G = \text{TOP} \)
2. \( S = \{\} \)
3. for each example \( e \) do:
4. \hspace{1em} if \( e \) is positive \( S = \text{UpdatePositive}(e) \)
5. \hspace{1em} if \( e \) is negative \( G = \text{UpdateNegative}(e) \)
6. od
UpdatePositive

Input: positive example e and sets of hypothesis G,S.
- eliminate all concepts in G that are not consistent with the example.
- minimally generalize all concepts in S until they are consistent with the example.
- eliminate all those concepts in S not are not a specialization of some concept in G and all those concepts that are a generalization of some other concept in S.

Version Spaces - Update negative

Input: negative example e and sets of hypothesis G,S.
- eliminate all concept in S that are consistent with the example.
- minimally specialize all concepts in G until they are not consistent with the example.
- eliminate all those concepts in G that are not a generalization of some concept in S and all those concepts that are specialization of some other concept in G.
Version spaces

Possible results from the algorithm:

G = S \quad \text{Gives us a unique solution.}

Collapse (G = {} or S = TOP) mean that there are no consistent hypothesis for the example.

Otherwise: No more examples but still hypothesis in the version space.
Limited example

UpdatePositive:
- eliminate in G if not consistent
- generalize in S until consistent
- eliminate in S unless specialization of G or if generalization of S.

UpdateNegative:
- eliminate in S if consistent
- specialize in G until unconsistent.
- eliminate in G unless generalization of S or is specialization of G.

Training:
+ Yes-S No-T
- Yes-S Yes-T
+ No-S No-T
Decision Trees

Toy robot example, attributes:

- Is-smiling: Yes, No
- Has-Tie: Yes, No
- Holding: Sword, Balloon, Flag
- Head-shape: Round, Square, Octagon
- Body shape: Round, Square, Octagon

Classification: friendly or unfriendly

Classification language:
Express classification using a tree.
Decision Trees

Algorithm: BuildTree

Input: Set of examples E

1. If all examples in E are positive or all examples are negative then create terminal node labeled positive resp. negative and return.
2. If there are some positive and negative examples, choose the "best" attribute A and split examples into:
   \[ P = \text{All examples } e \text{ in E such that } A \text{ is true in } e. \]
   \[ N = \text{All examples } e \text{ in E such that } A \text{ is false in } e. \]
3. Create node branching on A with children given by BuildTree(P) resp. BuildTree(N).

Easy to implement recursively.

Problem: How do we limit the depth of the tree to get better and faster classifications?
Minimizing tree depth

Idea: We compute the entropy of the training data and always branch on the attribute which gives us the least entropy in the remaining partitions.

Entropy $I$ of set $S = P \cup N$:

$$I(\{P,N\}) = \log_2(|S|) - (|P|/|S|) \log_2(|P|) - (|N|/|S|) \log_2(|N|)$$

Entropy $V$ when partitioning $S$ into $S_1, \ldots, S_n$:

$$V(\{S_i \mid 1 \leq i \leq n\}) = \sum_{i=1}^{n} (|S_i|/|S|) I(\{P(S_i),N(S_i)\})$$
<table>
<thead>
<tr>
<th>Example</th>
<th>Pos/Neg</th>
<th>Smiling</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>-</td>
<td>yes</td>
</tr>
<tr>
<td>R2</td>
<td>-</td>
<td>yes</td>
</tr>
<tr>
<td>R3</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td>R4</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td>R5</td>
<td>+</td>
<td>yes</td>
</tr>
<tr>
<td>R6</td>
<td>+</td>
<td>yes</td>
</tr>
</tbody>
</table>

Attribute: Is-smiling

attribute partition of example set:
\{ \{R3,R4\}, \{R1,R2,R5,R6\} \}

class partitions:
1: \{ \emptyset, \{R3,R4\} \}
2: \{ \{R5,R6\}, \{R1,R2\} \}

\[ I(\{\emptyset, \{R3,R4\}\}) = \log_2 2 - 0/2 \log_2 0 - 2/2 \log_2 2 = 0 \]

\[ I(\{\{R5,R6\}, \{R1,R2\}\}) = \log_2 4 - 2/4 \log_2 2 - 2/4 \log_2 2 = 1 \]

\[ V(\{S_i \mid 1 \leq i \leq n\}) = \frac{2}{6} \cdot 0 + \frac{4}{6} \cdot 1 = \frac{2}{3} \]
Genetic Algorithms

Basic idea:

Use a population of genomes that represent possible solutions.
Evaluation of individual performance.
The fittest individuals survive to "mate" and create offspring (replacing the weakest candidates).
Produce offspring by mutation and combination.

Based on evolution: "survival of the fittest"

Genetic programming: special case of GA

Typical problems:

Optimise a set of parameters (e.g. optimise engine parameters)
Design (e.g. create electric circuit, create a function f(x) "program")
Characteristics of Genetic Algorithms

- Blind search.
- Use of codings, not decision variables.
- Search from a set of possible solutions, not from a point.
- Use of randomized operators, not deterministic rules.

- Robust method dealing with a wide area of problems.
- Good for hard problems (Easy problems can usually be solved better by other means).
- Good for problems where no good specialized algorithms exist.
- Provides "acceptably good" solutions "acceptably fast".

Genetic algorithms have no guarantees to work
Depends very much on problem type, problem description (genomes), implementations (mutations, crossovers), population size etc.
Applications

Numerical function optimization:
   Eg. sensor calibration, training of neural networks, ...

Image processing
   Eg. Recognising persons, objects, ...

Combinatorial optimization tasks
   Eg. Travelling salesman problem, job scheduling, bin packing.

Design tasks
   Eg. graph layout, floor scheduling, ...

Machine learning in control:
   ...
Basic idea

Individuals
- Represent solutions to the problem to solve, has a *genome*.
- Are assigned *fitness score* (how good is the solution?). Compare with heuristics or the *utility* of an agent.
- Fittest individuals are given opportunity to "reproduce".

Population
Population = set of individuals
From one population (generation) a new population (generation) is derived by:
- Selecting a subset of the individuals from the current generation
- Letting the selected individuals reproduce.
Genetic Algorithms - The algorithm

Algorithm: GeneticAlgorithm

begin
    t = 0
    P(t) = initial population
    evaluate P(t)
    while (not terminal condition) do
        t = t+1
        select and update P(t) from P(t-1)
        evaluate P(t)
    od
    return best individual of P(t)
end

Initial population: Usually random solutions (individuals).
Selection: Select the N fittest individuals from the whole population.
Update: Create offspring by mutation and crossovers.
Terminal condition: After specified number of iterations, good enough solution, convergence speed, etc.
Coding

Assumption: a potential solution to the problem can be represented as a set of parameters. The parameters are joined together to form a string of values.

Example:

Problem: Maximize the function \( f(x,y,z) \).

A solution is a triple with values for \( x \), \( y \) and \( z \). If we represent these values in binary notation with 16 digits we may get as a possible solution:

\[
\begin{align*}
x &= 011010110110110111 \\
y &= 100000101101110110 \\
z &= 1100111001101110
\end{align*}
\]

Possible encoding:

011010110110110110000010111011101100111100110110
Fitness function

Fitness function: Return a numerical value for each individual representing its utility.

Example:

Problem: Maximize the function \( f(x, y, z) \).
fitness function = \( f \)

The fitness function can measure a combination of things:

Problem: Design of product

fitness function is combination of size of product, functionality of product, construction cost, construction time, transportation cost etc.
Reproduction

Selection:
   For favouring good solutions.

Crossover:
   For rapid exploration of the search space.

Mutation:
   For randomness, no point has zero probability.
Selection

Individuals are selected using a random scheme favouring more fit individuals.

- Roulette method, probability of selection proportional to fitness function.

- Ranking, probability of selection proportional to ranking.

- Tournament, compare two individuals and select the winner.
Crossover

Create offspring from two parents by combining their genomes around some randomly selected crossover point.

Example:

Parent A: 0111 1011
Parent B: 0101 1010

Child: 0111 1010
Child: 0101 1011

...
Mutation

Randomly change one gene of an individual.

Example:

Before: 0111 1011
After: 1111 1011
Convergence

The fitness of the best and the average individual in a generation increases towards the optimum.

Possible termination criteria:
- Run for fixed number of generations
- Best individual is sufficiently good
- A certain percentage of the population has the same fitness value.
- The convergence rate from generation to generation below certain threshold.
Neural Networks

Based on how the human brain works.

Simulate a number of "neurons"

Give problems as input, result is the output from (a subset of) the neurons.

Different neural networks – different layout and technical details of neurons.

"Train" the neurons to recognise test data.

Supervised vs. Unsupervised training.