Introduction to Computer Science

S. Arun-Kumar
sak@cse.iitd.ernet.in
Department of Computer Science and Engineering
I. I. T. Delhi, Hauz Khas, New Delhi 110 016.

April 29, 2004
Contents

Lecture 1  Lecture 11  Lecture 21
Lecture 2  Lecture 12  Lecture 22
Lecture 3  Lecture 13
Lecture 4  Lecture 14
Lecture 5  Lecture 15  Lecture 25
Lecture 6  Lecture 16  Lecture 26
Lecture 7  Lecture 17
Lecture 8  Lecture 18
Lecture 9  Lecture 19
Lecture 10  Lecture 20
Introduction

- This course is about computing
- Computing as a process is nearly as fundamental as arithmetic
- Computing as a mental process
- Computing may be done with a variety of tools which may or may not assist the mind
Computing tools

- Sticks and stones (counting)
- Paper and pencil (an aid to mental computing)
- Abacus (still used in Japan!)
- Slide rules (ask a retired engineer!)
- Ruler and compass
Ruler and Compass

Actually it is a **computing** tool!

- Construct a length that is **half** of a given length
- **Bisect** an angle
- Construct a square that is **twice** the area of a given square
- Construct $\sqrt{10}$
Computing and Computers

- **Computing** is much more fundamental
- **Computing** may be done without a **computer** too!
- But a **Computer** cannot do much besides **computing**.
Primitives

- Each tool has a set of capabilities called **primitive operations** or **primitives**

**Ruler:** Can specify **lengths**, **lines**

**Compass:** Can define **arcs** and **circles**

- The primitives may be combined in various ways to perform a **computation**.

- **Example** Constructing a right bisector of a given line segment.
Algorithm

Given a problem to be solved with a given tool, the attempt is to evolve a combination of primitives of the tool in a certain order to solve the problem.

An explicit statement of this combination along with the order is an algorithm.
The Digital Computer: Our Computing Tool

**Algorithm:** A *finite* specification of the solution to a given problem using the primitives of the computing tool.

- It specifies a definite input and output
- It is unambiguous
- It specifies a solution as a *finite* process i.e. the number of steps in the computation is *finite*
Algorithms

An algorithm will be written in a mixture of English and standard mathematical notation. Usually,

- Algorithms written in a natural language are often ambiguous
- Mathematical notation is not ambiguous, but still cannot be understood by machine
- Algorithms written by us use various mathematical properties. We know them, but the machine doesn’t.
Programming Language

- Require a way to communicate with a machine which has essentially no intelligence or understanding.
- Translate the algorithm into a form that may be “understood” by a machine.
- This “form” is usually a program.

Program: An algorithm written in a programming language.
Programs and Languages

- Every programming language has a well-defined **vocabulary** and a well-defined **grammar**
- Each **program** has to be written following rigid **grammatical** rules
- A programming language and the computer together form our single **computing tool**
- Each program uses **only** the primitives of the computing tool
Programs

Program: An algorithm written in the grammar of a programming language.

A grammar is a set of rules for forming sentences in a language.

Each programming language also has its own vocabulary and grammar just as in the case of natural languages.

We will learn the grammar of the language as we go along.
Programming

The act of writing programs and testing them is programming

Even though most programming languages use essentially the same computing primitives, each programming language needs to be learned.

Programming languages differ from each other in terms of the convenience and facilities they offer even though they are all equally powerful.
Computing Models

We consider mainly two models.

- **Functional**: A program is specified simply as a mathematical expression.
- **Imperative**: A program is specified by a sequence of commands to be executed.

Programming languages also come mainly in these two flavours. We will often identify the computing model with the programming language.
Primitives

Every programming language offers the following capabilities to define and use:

- Primitive expressions and data
- Methods of combination of expressions and data
- Methods of abstraction of both expressions and data

The functional model
Primitive expressions

The simplest objects and operations in the computing model. These include

- **basic data elements**: numbers, characters, truth values etc.
- **basic operations on the data elements**: addition, substraction, multiplication, division, boolean operations, string operations etc.
- **a naming mechanism** for various quantities and expressions to be used without repeating definitions
Methods of combination

Means of combining simple expressions and objects to obtain more complex expressions and objects.

Examples: composition of functions, inductive definitions
Methods of abstraction

Means of naming and using groups of objects and expressions as a single unit

Examples: functions, data structures, modules, classes etc.
The Functional Model

The functional model is very convenient and easy to use:

- Programs are written (more or less) in mathematical notation
- It is like using a hand-held calculator
- Very interactive and so answers are immediately available
- Very convenient for developing, testing and proving algorithms

Standard ML
Mathematical Notation 1: Factorial

\[
n! = \begin{cases} 
1 & \text{if } n < 1 \\
n \times (n - 1)! & \text{otherwise}
\end{cases}
\]
Mathematical Notation 2: Factorial

Or more informally,

\[ n! = \begin{cases} 
1 & \text{if } n < 1 \\
1 \times 2 \times \ldots \times n & \text{otherwise}
\end{cases} \]
Mathematical Notation 3: Factorial

How about this?

\[ n! = \begin{cases} 
1 & \text{if } n < 1 \\
(n + 1)!/(n + 1) & \text{otherwise}
\end{cases} \]

Mathematically correct but computationally incorrect!
A Functional Program: Factorial

fun fact n = if n < 1 then 1 else n * fact (n-1)
A Computation: Factorial

sml
Standard ML of New Jersey,
A Computation: Factorial

sml
Standard ML of New Jersey,
- fun fact n =
= 
A Computation: Factorial

sml
Standard ML of New Jersey,
- fun fact n =
= if n < 1 then 1
=

A Computation: Factorial

sml
Standard ML of New Jersey,
- fun fact n =
  = if n < 1 then 1
  = else n * fact (n-1);
val fact = fn : int -> int
-
A Computation: Factorial

sml
Standard ML of New Jersey,
- fun fact n =
  = if n < 1 then 1
  = else n * fact (n-1);
val fact = fn : int -> int
- fact 8;
val it = 40320 : int
-
A Computation: Factorial

sml
Standard ML of New Jersey,
- fun fact n =
= if n < 1 then 1
= else n * fact (n-1);
val fact = fn : int -> int
- fact 8;
val it = 40320 : int
- fact 9;
val it = 362880 : int
A Computation: Factorial

sml

Standard ML of New Jersey,
- fun fact n =
  = if n < 1 then 1
  = else n * fact (n-1);
val fact = fn : int -> int
- fact 8;
val it = 40320 : int
- fact 9;
val it = 362880 : int
-
Standard ML

- Originated as part of a theorem-proving development project
- Runs on both Windows and UNIX environments
- Is free!
- http://www.smlnj.org
SML: Important Features

- Has a small vocabulary of just a few short words
- Far more “intelligent” than currently available languages:
  - automatically finds out what various names mean and
  - their correct usage
- Haskell, Miranda and Caml are a few other such languages.
Algorithms & Programs

- Algorithm
- Need for a formal notation
- Programs
- Programming languages
- Programming
- Functional Programming
- Standard ML

Factorial
SML: Primitive Integer Operations 1

sml
Standard ML of New Jersey,
SML: Primitive Integer Operations 1

sml
Standard ML of New Jersey,
- val x = 5;
val x = 5 : int
-
SML: Primitive Integer Operations 1

sml
Standard ML of New Jersey,
- val x = 5;
val x = 5 : int
- val y = 6;
val y = 6 : int
-
sml

Standard ML of New Jersey,
- val x = 5;
val x = 5 : int
- val y = 6;
val y = 6 : int
- x+y;
val it = 11 : int
-
SML: Primitive Integer Operations 1

sml
Standard ML of New Jersey,
- val x = 5;
val x = 5 : int
- val y = 6;
val y = 6 : int
- x+y;
val it = 11 : int
- x-y;
val it = ~1 : int
-
SML: Primitive Integer Operations 1

Standard ML of New Jersey,
- val x = 5;
val x = 5 : int
- val y = 6;
val y = 6 : int
- x+y;
val it = 11 : int
- x-y;
val it = ~1 : int
- it + 5;
val it = 4 : int
SML: Primitive Integer Operations 2

val x = 5 : int
- val y = 6;
val y = 6 : int
- x+y;
val it = 11 : int
- x-y;
val it = ~1 : int
- it + 5;
val it = 4 : int
- x * y;
val it = 30 : int
val y = 6 : int
- x+y;
val it = 11 : int
- x-y;
val it = ~1 : int
- it + 5;
val it = 4 : int
- x * y;
val it = 30 : int
- val a = 25;
val a = 25 : int
val it = 11 : int
- x - y;
val it = ~1 : int
- it + 5;
val it = 4 : int
- x * y;
val it = 30 : int
- val a = 25;
val a = 25 : int
- val b = 7;
val b = 7 : int
val it = ~1 : int
- it + 5;
val it = 4 : int
- x * y;
val it = 30 : int
- val a = 25;
val a = 25 : int
- val b = 7;
val b = 7 : int
- val q = a div b;
val q = 3 : int
-
SML: Primitive Integer Operations 2

- \( x \times y; \)
val it = 30 : int
- val a = 25;
val a = 25 : int
- val b = 7;
val b = 7 : int
- val q = a div b;
val q = 3 : int
- val r = a mod b;
GC #0.0.0.0.2.45: (0 ms)
val r = 4 : int
SML: Primitive Integer Operations 3

- val a = 25;
val a = 25 : int
- val b = 7;
val b = 7 : int
- val q = a div b;
val q = 3 : int
- val r = a mod b;
GC #0.0.0.0.2.45:  (0 ms)
val r = 4 : int
- a = b*q + r;
val it = true : bool
-
SML: Primitive Integer Operations 3

- val b = 7;
val b = 7 : int
- val q = a div b;
val q = 3 : int
- val r = a mod b;

GC #0.0.0.0.2.45: (0 ms)
val r = 4 : int

- a = b*q + r;
val it = true : bool
- val c = ~7;
val c = ~7 : int

-
SML: Primitive Integer Operations 3

val q = a div b;
val q = 3 : int
val r = a mod b;
GC #0.0.0.0.2.45: (0 ms)
val r = 4 : int
val a = b*q + r;
val it = true : bool
val c = ~7;
val c = ~7 : int
val q1 = a div c;
val q1 = ~4 : int
SML: Primitive Integer Operations 3

- val r = a mod b;
Earn #0.0.0.0.2.45: (0 ms)
val r = 4 : int
- a = b\*q + r;
val it = true : bool
- val c = ~7;
val c = ~7 : int
- val q1 = a div c;
val q1 = ~4 : int
- val r1 = a mod c;
val r1 = ~3 : int
-
val r = 4 : int
- a = b*q + r;
val it = true : bool
- val c = ~7;
val c = ~7 : int
- val q1 = a div c;
val q1 = ~4 : int
- val r1 = a mod c;
val r1 = ~3 : int
- a = c*q1 + r1;
val it = true : bool
-
Quotient & Remainder

For any two integers $a$ and $b$, the quotient $q$ and remainder $r$ are uniquely determined to satisfy

- $a = b \times q + r$
- $0 \leq |r| < |b|$

So

- $0 \leq r < b$ when $b > 0$
- $b < r \leq 0$ when $b < 0$
sml
Standard ML of New Jersey,
- val real_a = real a;
val real_a = 25.0 : real
-
SML: Primitive Real Operations 1

sml

Standard ML of New Jersey,
- val real_a = real a;
val real_a = 25.0 : real
- real_a + b;

stdin:40.1-40.11 Error: operator and operand don’t agree [tycon mismatch]

operator domain: real * real
operand: real * int
in expression:
  real_a + b

SML: Primitive Real Operations 1

stdin:40.1-40.11 Error: operator and operand don't agree [tycon mismatch]
operator domain: real * real
operand: real * int
in expression:
  real_a + b
- b + real_a;

stdin:1.1-2.6 Error: operator and operand don't agree [tycon mismatch]
operator domain: int * int
operand: int * real
in expression:
  b + real_a
-
- val a = 25.0;
val a = 25.0 : real
- val b = 7.0;
val b = 7.0 : real
- a/b;
val it = 3.57142857143 : real
- a div b;

stdin:49.3-49.6 Error: overloaded variable not defined at type symbol: div
type: real
GC #0.0.0.0.3.98: (0 ms)
val c = a/b;
val c = 3.57142857143 : real
- trunc(c);
val it = 3 : int
- trunc (c + 0.5);
val it = 4 : int
SML: Primitive Real Operations 4

- val d = 3.0E10;
val d = 30000000000.0 : real
- val pi = 0.314159265E1;
val pi = 3.14159265 : real
- d+pi;
val it = 30000000003.1 : real
- d-pi;
val it = 29999999996.9 : real
- pi + d;
val it = 30000000003.1 : real
SML: Precision

- $\pi + d \times 10.0$;
val it = 30000000003.0 : real

- $\pi + d \times 100.0$;
val it = 3E12 : real

- $d \times 100.0 + \pi$;
val it = 3E12 : real

- $d \times 100.0 - \pi$;
val it = 3E12 : real

- $d \times 10.0 - \pi$;
val it = 299999999997.0 : real

-
Fibonacci Numbers

\[
\begin{align*}
F(0) &= 1 \\
F(1) &= 1 \\
F(n) &= F(n - 1) + F(n - 2) \quad \text{if } n > 1
\end{align*}
\]

fun fib (n) = 
  if (n = 0) orelse (n = 1) then 1 
  else fib (n-1) + fib (n-2);
Euclidean Algorithm

Relies on a theorem proved by Euclid

fun gcd (a, b) =
    if b=0 then a
    else gcd (b, a mod b);

Is it really inductive?
Recapitulation: Integers & Real

- Primitive Integer Operations
- Primitive Real Operations
- Some algorithms
Recap: Integer Operations

- **Primitive Integer Operations**
  - Naming, $\oplus$, $\ominus$, $\sim$
  - Multiplication, division
  - Quotient & remainder
- **Primitive Real Operations**
- **Some algorithms**
Recapitulation: Real Operations

- Primitive Integer Operations
- Primitive Real Operations
  - Integer to Real
  - Real to Integer
  - Real addition & subtraction
  - Real division
  - Real Precision
- Some algorithms

Back
Recapitulation: Simple Algorithms

- Primitive Integer Operations
- Primitive Real Operations
- Some algorithms
  - Factorial
  - Fibonacci
  - Euclidean GCD
More Algorithms

- Powering
- Integer square root
- Combinations $\binom{n}{k}$
Powering: Math

For any integer or real number \( x \neq 0 \)
and non-negative integer \( n \)

\[
x^n = x \times x \times \cdots \times x
\]
\( n \) times

Noting that \( x^0 = 1 \) we give an inductive definition:

\[
x^n = \begin{cases} 
1 & \text{if } n = 0 \\
x^{n-1} \times x & \text{otherwise}
\end{cases}
\]
Powering: SML

fun power (x:real, n) =
  if n = 0
  then 1.0
  else power (x, n-1) * x

Is it technically complete?
Technical completeness

Can it be always guaranteed that

- $x$ will be real?
- $n$ will be integer?
- $n$ will be non-negative?
- $x \neq 0$?

If $x = 0$ what is $0.0^0$?
What SML says

sml
Standard ML of New Jersey
- use "/tmp/power.sml"
[opening /tmp/power.sml]
val power = fn : real * int -> real
val it = () : unit

Can it be always guaranteed that

• $x$ will be real? YES
• $n$ will be integer? YES
Technical completeness

Can it be always guaranteed that

• $n$ will be non-negative? NO
• $x \neq 0$? NO

If $x = 0$ what is $0.0^0$?

- `power(0.0, 0);`
- `val it = 1.0 : real`
What SML says ... contd

sml
Standard ML of New Jersey
val power = fn : real * int -> real
val it = () : unit
- power (~2.5, 0);
val it = 1.0 : real
- power (0.0, 3);
val it = 0.0 : real
- power (2.5, ~3)

Goes on forever!
Powering: Math 1

For any real number $x$ and integer $n$

$$x^n = \begin{cases} 
1.0/x^{-n} & \text{if } n < 0 \\
1 & \text{if } n = 0 \\
x^{n-1} \times x & \text{otherwise}
\end{cases}$$
Powering: SML 1

fun power (x, n) = 
  if n < 0 
  then 1.0/power(x, ~n) 
  else if n = 0 
  then 1.0 
  else power (x, n-1) * x 

Is this definition technically complete?
Technical Completeness

- $0.0^0 = 1.0$ whereas $0.0^n = 0$ for positive $n$
- What if $x = 0.0$ and $n = -m < 0$?

Then

\[
0.0^n = 1.0/(0.0^m)
= 1.0/0.0
\]

Division by zero!
What SML says

- `power (2.5, ~2);`
  `val it = 0.16 : real`
- `power (~2.5, ~2);`
  `val it = 0.16 : real`
- `power (0.0, 2);`
  `val it = 0.0 : real`
- `power (0.0, ~2);`
  `val it = inf : real`

SML is somewhat more understandable than most languages
Powering: Integer Version

\[ x^n = \begin{cases} 
\text{undefined} & \text{if } n < 0 \\
1 & \text{if } n = 0 \\
x^{n-1} \times x & \text{otherwise}
\end{cases} \]

Technical completeness requires us to consider the case \( n < 0 \). Otherwise, the computation can go on forever.

Notation: \( \bot \) denotes the undefined
Exceptions: A new primitive

```haskell
exception negExponent;
fun intpower (x, n) = 
  if n < 0
  then raise negExponent
  else if n = 0 then 1
  else intpower (x, n-1) * x
```
Integer Power: SML

- intpower(3, 4);
  val it = 81 : int
- intpower(~3, 5);
  val it = ~243 : int
- intpower(3, ~4);

uncaught exception negExponent
  raised at: intpower.sml:4.16-4.32

-
Integer Square Root

\[ isqrt(n) = \lfloor \sqrt{n} \rfloor \]

- fun isqrt n =
  
  trunc (Real.Math.sqrt (real (n)));

val isqrt = fn : int -> int
- isqrt (38);
val it = 6 : int
- isqrt (~38);

uncaught exception domain error
  raised at: boot/real64.sml:89.32-89.46

-
Integer Square Root

Suppose $\text{Real.Math.sqrt}$ were not available to us!

The integer $k \geq 0$ such that $k^2 \leq n < (k + 1)^2$

That is,

$$isqrt(n) = \begin{cases} \perp & \text{if } n < 0 \\ k & \text{otherwise} \end{cases}$$

where $0 \leq k^2 \leq n < (k + 1)^2$.

This value of $k$ is unique!
An analysis

\[ 0 \leq k^2 \leq n < (k + 1)^2 \]
\[ \Rightarrow 0 \leq k \leq \sqrt{n} < k + 1 \]
\[ \Rightarrow 0 \leq k \leq n \]

**Strategy.** Use this fact to close in on the value of \( k \). Start with the interval \([l, u] = [0, n]\) and try to shrink it till it collapses to the interval \([k, k]\) which contains a single value.
Algorithmic idea

If $n = 0$ then $\text{isqrt}(n) = 0$.
Otherwise with $[l, u] = [0, n]$ use one or both of the following to shrink the interval $[l, u]$.

- if $(l + 1)^2 \leq n$ then try $[l + 1, u]$
- if $u^2 > n$ then try $[l, u - 1]$. 
Algorithm: isqrt

\[ isqrt(n) = \begin{cases} \perp & \text{if } n < 0 \\ 0 & \text{if } n = 0 \\ shrink(n, 0, n) & \text{if } n > 0 \end{cases} \]

where
Algorithm: shrink

\[
shrink(n, l, u) = \begin{cases} 
  l & \text{if } l = u \\
  shrink(n, l + 1, u) & \text{if } l < u \text{ and } (l + 1)^2 \leq n \\
  shrink(n, l, u - 1) & \text{if } l < u \text{ and } u^2 > n \\
  \bot & \text{if } l > u 
\end{cases}
\]
SML: shrink

exception intervalError;

fun shrink (n, l, u) =
  if l > u orelse
    l*l > n orelse
    u*u < n
  then raise intervalError
  else if (l+1)*(l+1) <= n
  then shrink (n, l+1, u)
  else l;

intsqrt
exception negError;
fun intsqrt n =
  if n<0
  then raise negError
  else if n=0
  then 0
  else shrink (n, 0, n)
exception intervalError
val shrink = fn : int * int * int -> int
exception negError
val intsqrt = fn : int -> int
val it = () : unit
  - intsqrt 8;
val it = 2 : int
  - intsqrt 16;
val it = 4 : int
  - intsqrt 99;
val it = 9 : int
  -
SML: Reorganizing Code

- `shrink` was used to develop `intsqrt`
- Is `shrink` general-purpose enough to be kept separate?
- Shouldn’t `shrink` be placed within `intsqrt`?
Intsqrt: Reorganized

exception negError;
fun intsqrt n =
  let fun shrink (n, l, u) =
    in if n<0
      then raise negError
    else if n=0
      then 0
    else shrink (n, 0, n)
    end
  end
shrink: Another algorithm

\[ shrink2(n, l, u) = \begin{cases} 
  l & \text{if } l = u \text{ or } u = l + 1 \\
  shrink2(n, m, u) & \text{if } l < u \text{ and } m^2 \leq n \\
  shrink2(n, l, m) & \text{if } l < u \text{ and } m^2 > n \\
  \bot & \text{if } l > u 
\end{cases} \]

where \( m = (l + u) \div 2 \)
Shrink2: SML

fun shrink2 (n, l, u) =
  if l>u orelse
    l*l > n orelse
    u*u < n
  then raise intervalError
  else if l = u
  then l
Shrink2: SML ... contd

else

let val m = (l+u) div 2;
val msqr = m*m

in if msqr <= n
then shrink (n, m, u)
else shrink (n, l, m)

end;
Recap: More Algorithms

- $x^n$ for real and integer $x$
- Integer square root
Recap: Power

- $x^n$ for real and integer $x$
  - Technical Completeness
    - Undefinedness
    - Termination
  - More complete definition for real $x$
  - Power of an integer
  - $\bot$ and exceptions

- Integer square root
Recap: Technical completeness

Can it be always guaranteed that

- $x$ will be real? YES
- $n$ will be integer? YES
- $n$ will be non-negative? NO
- $x \neq 0$? NO

If $x = 0$ what is $0.0^0$?

INFINITE COMPUTATION
Recap: More Algorithms

- $x^n$ for real and integer $x$
- Integer square root
  - Analysis
  - Algorithmic idea
  - Algorithm
  - where
    - and let ...in ...end
except exception negError;
except exception intervalError;
fun intsqrt n =
  let fun shrink (n, l, u) =
    if l>u orelse
      l*l > n orelse
      u*u < n
    then raise intervalError
    else if (l+1)*(l+1) <= n
    then shrink (n, l+1, u)
    else l;

Intsqrt: Reorganized

\[
in \text{ if } n < 0 \\
\text{ then raise negError} \\
\text{ else if } n = 0 \\
\text{ then 0} \\
\text{ else shrink } (n, \ 0, \ n) \\
\text{ end}
\]
Some More Algorithms

- Combinations
- Perfect Numbers
Combinations: Math

\[ nC_k = \frac{n!}{(n-k)!k!} \]

\[ = \frac{n(n-1)\cdots(n-k+1)}{k!} \]

\[ = \frac{n(n-1)\cdots(k+1)}{(n-k)!} \]

\[ = n-1C_{k-1} + n-1C_k \]

Since we already have the function \texttt{fact}, we may program \( nC_k \) using any of the above identities. Let's program it using the last one.
Combinations: Details

Given a set of $n \geq 0$ elements, find the number of subsets of $k$ elements, where $0 \leq k \leq n$

$$nC_k = \begin{cases} 
\bot & \text{if } n < 0 \text{ or } k < 0 \text{ or } k > n \\
1 & \text{if } n = 0 \text{ or } k = 0 \text{ or } k = n \\
n-1C_{k-1} + n-1C_k & \text{otherwise}
\end{cases}$$
Combinations: SML

exception invalid_arg;

fun comb (n, k) =
    if n < 0 orelse
        k < 0 orelse
        k > n
    then raise invalid_arg
    else if n = 0 orelse
        k = 0 orelse
        n = k
    then 1
    else comb (n-1, k-1) +
    comb (n-1, k);

Back to Some More Algorithms
Perfect Numbers

An integer $n > 0$ is perfect if it equals the sum of all its proper divisors. A divisor $k \mid n$ is proper if $0 < k < n$

$$k \mid n \iff n \mod k = 0$$

perfect$(n)$

$$\iff n = \sum \{k : 0 < k < n, k \mid n\}$$

$$\iff n = \sum_{k=1}^{n-1} \text{ifdivisor}(k)$$

where
Refinement

1. $if\text{divisor}(k)$ needs to be defined

2. $\sum_{k=1}^{n-1} if\text{divisor}(k)$ needs to be defined algorithmically.
Perfect Numbers: SML

exception nonpositive;

fun perfect (n) =
  if n <= 0
  then raise nonpositive
  else
    n = sum_divisors (1, n-1)

where sum_divisors needs to be defined
\[
\sum_{l}^{u} \text{ifdivisor}(k)
\]

\[
\sum_{k=l}^{u} \text{ifdivisor}(k) =
\begin{cases}
0 & \text{if } l > u \\
\text{ifdivisor}(l) + \\
\sum_{k=l+1}^{n-1} \text{ifdivisor}(k) & \text{otherwise}
\end{cases}
\]

where \( \text{ifdivisor}(k) \) needs to be defined
SML: \texttt{sum\_divisors}

From the algorithmic definition of
\[ \sum_{k=1}^{u} \text{ifdivisor}(k) \]

\begin{verbatim}
fun sum_divisors (l, u) = 
  if l > u
  then 0
  else ifdivisor (l) +
    sum_divisors (l+1, u)
where ifdivisor(k) still needs to be defined
\end{verbatim}
if divisor \textbf{and} if divisor

\[
\text{if divisor}(k) = \begin{cases} 
  k & \text{if } k | n \\
  0 & \text{otherwise}
\end{cases}
\]

fun if divisor (k) =
  if n mod k = 0 then k
  else 0

Not \textbf{technically complete}! However \ldots
fun sum_divisors (l, u) = 
    if l > u then 0 
    else 
        let fun ifdivisor (k) = 
            if n mod k = 0 
            then k 
            else 0 
        in ifdivisor (l) + 
        sum_divisors (l+1, u) 
    end 

Clearly \( k \in [l, u] \)
SML: Assembly 2

exception nonpositive;
fun perfect (n) =
  if n <= 0
  then raise nonpositive
  else
    let fun sum_divisors (l, u) =
      ...
    in n = sum_divisors (1, n-1)
    end

Clearly $k \in [l, u] = [1, n - 1]$ whenever $n > 0$.

Technically complete!
Perfect Numbers ... contd.

Clearly for all \( k, \ n/2 < k < n \),
if \( \text{divisor}(k) = 0 \).

\[ \lfloor n/2 \rfloor = n \div 2 < n/2 \]

Hence

\[ \sum_{k=1}^{n-1} \text{ifdivisor}(k) = \sum_{k=1}^{n \div 2} \text{ifdivisor}(k) \]
Perfect Numbers

... contd.

Hence

\[ \text{perfect}(n) \]

\[ \iff n = \sum_{k=1}^{n-1} \text{ifdivisor}(k) \]

\[ \iff n = \sum_{k=1}^{n \text{ div } 2} \text{ifdivisor}(k) \]

where

\[ \text{ifdivisor}(k) = \begin{cases} k & \text{if } k | n \\ 0 & \text{otherwise} \end{cases} \]
exception nonpositive;
fun perfect (n) =
  if n <= 0
  then raise nonpositive
  else
    let fun sum_divisors (l, u) =
        ...
    in n = sum_divisors (1, n div 2)
    end

Clearly $k \in [l, u] = [1, n \operatorname{div} 2]$ whenever $n > 0$. 
Technically complete!
Perfect Numbers: Run

exception nonpositive
val perfect = fn : int -> bool
val it = () : unit
- perfect ~8;
uncaught exception nonpositive
raised at: perfect.sml:4.16-4.27
- perfect 5;
val it = false : bool
Perfect Numbers: Run

- perfect 6;
val it = true : bool
- perfect 23;
val it = false : bool
- perfect 28;
GC #0.0.0.1.3.88:   (1 ms)
val it = true : bool
- perfect 30;
val it = false : bool
SML: Code variations

exception nonpositive;

fun perfect (n) =
  if n <= 0
  then raise nonpositive
  else
    let
      fun ifdivisor (k) = ...;
      fun sum_divisors (l, u) = ...
    in
      n=sum_divisors (1, n div 2)
    end

Technically complete though ifdivisor, by itself is not!
SML: Code variations

What about this?

```sml
exception nonpositive;
fun perfect (n) =
  let
    fun ifdivisor (k) = ...;
    fun sum_divisors (l, u) =
      in if n <= 0
      then raise nonpositive
      else
        n = sum_divisors (1, n div 2)
  end
```
SML: Code variations

What about this?

```sml
exception nonpositive;
fun ifdivisor (k) = ...;
fun sum_divisors (l, u) = ...;
fun perfect (n) =
  if n <= 0
  then raise nonpositive
  else
    n = sum_divisors (1, n div 2)
```

Technically incomplete!
Summation: Generalizations

Need a method to compute summations in general.
For any function $f : \mathbb{Z} \rightarrow \mathbb{Z}$ and integers $l$ and $u$,

$$
\sum_{i=l}^{u} f(i) = \begin{cases} 
0 & \text{if } l > u \\
 f(l) + \sum_{i=l+1}^{u} f(i) & \text{otherwise}
\end{cases}
$$
Algorithmic Improvements:

1. perfect
2. shrink
Algorithmic Variations

1. For any \( k \mid n \), \( m = n \text{ div } k \) is also a divisor of \( n \)

2. \( 1 \) is a divisor of every positive number

3. For \( n > 2 \), \( \lfloor \sqrt{n} \rfloor < n \text{ div } 2 \)

4. Hence \( \sum_{k=1}^{n \text{ div } 2} i \text{f divisor}(k) = 1 + \sum_{k=2}^{\lfloor \sqrt{n} \rfloor} i \text{f divisor}2(k) \)
Algorithmic Variations

\[ \text{perfect}(n) \leftrightarrow n = 1 + \sum_{k=2}^{\lfloor \sqrt{n} \rfloor} \text{ifdivisor2}(k) \]

where

\[ \text{ifdivisor2}(k) = \begin{cases} 
k + (n \mod k) & \text{if } k | n \\
0 & \text{otherwise} \end{cases} \]

Are there any glitches? Is it technically correct and complete?
Recap

- Combinations
- Perfect Numbers
- Code Variations
- Algorithmic Variations
Recap: Combinations

\[ n \binom{C}{k} = \frac{n!}{(n-k)!k!} \]

\[ = \frac{n(n-1)\cdots(n-k+1)}{k!} \]

\[ = \frac{n(n-1)\cdots(k+1)}{(n-k)!} \]

\[ = n^{-1} C_{k-1} + n^{-1} C_k \]
Combinations 1

use "fact.sml";
exception invalid_arg;
fun comb_wf (n, k) =
  if n < 0 orelse
    k < 0 orelse
    k > n
  then raise invalid_arg
  else fact (n) div
    (fact (n-k) * fact (k));
exception invalid_arg;

fun comb (n, k) =
  if n < 0 orelse
     k < 0 orelse
     k > n
  then raise invalid_arg
  else if n = 0 orelse
         k = 0 orelse
         n = k
  then 1
  else (* 0<k<n *)
       prod (n, n-k+1) div
       fact (k)
Combinations 3

exception invalid_arg;
fun comb (n, k) =
  if n < 0 orelse
    k < 0 orelse
    k > n
  then raise invalid_arg
  else if n = 0 orelse
    k = 0 orelse
    n = k
  then 1
  else (* 0<k<n *)
    prod (n, k+1) div
    fact (n-k)
Perfect 2

\[ \text{perfect}(n) \]

\[ \iff n = 1 + \sum_{k=2}^{\lfloor \sqrt{n} \rfloor} \text{ifdivisor2}(k) \]

where

\[ \text{ifdivisor2}(k) = \begin{cases} k + m & \text{if } k \mid n \text{ and } k \neq m \\ k & \text{if } k \mid n \text{ and } k = m \\ 0 & \text{otherwise} \end{cases} \]

where \( m = (n \div k) \)
The previous inductive definition used

\[ x^n = (x \times x \times \cdots \times x) \times x \]

\[ n-1 \text{ times} \]

We could associate the product differently
\[ x^n = \begin{cases} 
(x \times x \times \cdots \times x) \\
\text{n/2 times} \\
\times (x \times x \times \cdots \times x) \\
\text{n/2 times} 
\end{cases} 
\]

when \( n \) is even and

\[ x^n = \begin{cases} 
(x \times x \times \cdots \times x) \\
\left\lfloor \frac{n}{2} \right\rfloor \text{ times} \\
\times (x \times x \times \cdots \times x) \\
\left\lfloor \frac{n}{2} \right\rfloor \text{ times} \\
\times x 
\end{cases} 
\]

when \( n \) is odd
Power2: Complete

\[ power2(x, n) = \begin{cases} 
1.0 / power2(x, n) & \text{if } n < 0 \\
1.0 & \text{if } n = 0 \\
(power2(x, \lfloor n/2 \rfloor))^2 & \text{if } \text{even}(n) \\
(power2(x, \lfloor n/2 \rfloor))^2 \times x & \text{otherwise}
\end{cases} \]

where \( \text{even}(n) \iff n \mod 2 = 0. \)
fun power2 (x, n) =
if n < 0
then 1.0/power2 (x, ~n)
else if n = 0
then 1.0
else
let fun even m =
    (m mod 2 = 0);
fun square y = y * y;
val pwr_n_by_2 =
    power2 (x, n div 2);
val sq_pwr_n_by_2 =
    square (pwr_n_by_2)
  in
    if even (n)
    then sq_pwr_n_by_2
    else x * sq_pwr_n_by_2
end
Computation: Issues

1. Correctness
   (a) General correctness
   (b) Technical Completeness
   (c) Termination
General Correctness

1. **Mathematical correctness** should be established for all algorithmic variations.

2. Program Correctness: Mathematically developed code should not be moved around arbitrarily.
   - **Code variations** should also be mathematically proven
Code: Justification

• How does one justify the correctness of
  – this version and
  – this version?

• Can one correct this version?

• But first of all, what is incorrect about this version?

  incorrectness
Recall

• A program is an
  – explicit,
  – unambiguous and
  – technically complete

translation of an algorithm written in mathematical notation.

• Moreover, mathematical notation is
  more concise than a program.

• Hence mathematical notation is
  easier to analyse and diagnose.
Features: Definition before Use

Definition of a name before use:

- *ifdivisor*(\(k\)) is defined first.
- *idivisor*(\(k\)) uses the name \(n\) without defining it.
- \(k\) has been defined (as an argument of *ifdivisor*(\(k\))) before being used.
Run ifdivisor

Standard ML of New Jersey,

```
- fun ifdivisor(k) =
  = if n mod k = 0
  = then k
  = else 0
;
```

```
stdin:18.8 Error: unbound variable
or constructor: n
```
Diagnosis: Features of programs

- So both $\text{sum\_divisors}(l, u)$ and $\text{perfect}(n)$ **may use** $\text{if\_divisor}(k)$.

- $\text{sum\_divisors}(l, u)$ is **defined** before $\text{perfect}(n)$.

- So $\text{perfect}(n)$ **may use** both $\text{if\_divisor}(k)$ and $\text{sum\_divisors}(l, u)$.
Let
\[\text{ifdivisor}(k) = \begin{cases} k & \text{if } k \mid n \\ 0 & \text{otherwise} \end{cases}\]
and \(\text{sum\_divisors}(l, u) = \begin{cases} 0 & \text{if } l > u \\ \text{ifdivisor}(l) + \text{sum\_divisors}(l + 1, u) & \text{otherwise} \end{cases}\)
and \(\text{perfect}(n) \iff n = \text{sum\_divisors}(1, \lfloor n/2 \rfloor)\)
Incorrectness

- \textit{if divisor}(k) has a single argument \( k \)
- But it actually depends upon \( n \) too!
- But that is not made \textit{explicit} in its definition.

Let's make it \textit{explicit}!
ifdivisor3

Let

\[ ifdivisor3(n, k) = \begin{cases} k & \text{if } k \mid n \\ 0 & \text{otherwise} \end{cases} \]

and \( sum\_divisors(l, u) = \begin{cases} 0 & \text{if } l > u \\ ifdivisor3(n, l) + sum\_divisors(l + 1, u) & \text{otherwise} \end{cases} \)

and \( perfect(n) \iff n = sum\_divisors(1, \left\lfloor n/2 \right\rfloor) \)
Run it!

Standard ML of New Jersey
- fun ifdivisor3 (n, k)
  = = if (n mod k = 0)
  = then k
  = else 0;
val ifdivisor3 =
fn : int * int -> int
-
fun sum_divisors (l, u) = if l > u then 0 else if divisor3 (n, l) + sum_divisors (l+1, u);

stdIn:40.18 Error: unbound variable or constructor: n

Now sum_divisors also depends on n!
Hey! Wait a minute!

But \( n \) was defined in \( \text{ifdivisor3}(n, k) \)!

So then where is the problem?

Let’s ignore it for the moment and come back to it later.
The $n$'s

Let

$$if\text{\,divisor}_3(n, k) = \begin{cases} k & \text{if } k | n \\ 0 & \text{otherwise} \end{cases}$$

and $$sum\_divisors2(n, l, u) = \begin{cases} 0 & \text{if } l > u \\ if\text{\,divisor}_3(n, l) + \\ sum\_divisors(l + 1, u) & \text{otherwise} \end{cases}$$

and $$perfect(n) \iff n = sum\_divisors2(n, 1, \lfloor n/2 \rfloor)$$
Scope

• The scope of a name begins from its definition and ends where the corresponding scope ends
• Scopes end with definitions of functions
• Scopes end with the keyword `end` in any `let ... in ... end`
Scope Rules

- Scopes may be **disjoint**
- Scopes may be nested **one completely** within another
- A scope cannot span two disjoint scopes
- Two scopes **cannot** (partly) overlap forward
Disjoint Scopes

let
val x = 10;
fun fun1 y = let ...
in ...
end

fun fun2 z = let ...
in ...
end

fun1 (fun2 x)
end
let
val x = 10;
fun fun1 y =
  let
    val x = 15
  in
    x + y
  end
in
  fun1 x
end
Overlapping Scopes

```plaintext
let
val x = 10;
fun fun1 y =
  ...
  ...
  ...
  ...
end

fun2 x
```

```
fun1 (fun2 x)
```
let

val x = 10;

fun fun1 y =

fun fun2 z =

fun1 (fun2 x)

end
Scope & Names

- A name may occur either as being defined or as a use of a previously defined name.
- The same name may be used to refer to different objects.
- The use of a name refers to the textually most recent definition in the innermost enclosing scope diagram.
let
val x = 10; val z = 5;
fun fun1 y =
    let
        val x = 15
    in
        x + y * z
    end
end

fun1 x

Back to scope names
let

val x = 10; val z = 5;

fun fun1 y =

let
val x = 15
in
x + y * z
end

end

fun1 x

val x = 10;

fun fun1 y =

let
val x = 15
in
x + y * z
end

val z = 5;

* z
let

val x = 10; val z = 5;

fun fun1

val x = 15

in

ex + y * z
end

end

Back to scope names
Names & References

let
val x = 10; val z = 5;
fun fun1
  y =
  let
  val x = 15
  in
  x + y
  end
end

fun1 x
val x = 10;
fun fun1  y =
let
val x = 15
in
x + y * z
end

Back to scope names
let
  val x = 10; val z = 5;
  fun fun1 =
    let
      val x = 15
    in
      x + y
    end
  end
end

Back to scope names
let
val \( x = 10 \); val \( z = 5 \);
fun fun1 y =
  let
  val x = 15
  in
  x + y
  end
val z = 5;
* z

Back to scope names
let
val \( x = 10; \) val \( z = 5; \)
fun \( \text{fun1} \)
y = 
let
val \( x = 15 \)
in
x + y
end
val \( z = 5; \)

Back to scope names
let

val $x = 10;$ val $x = x - 5;$

fun fun1 $y =$ let ...
in ... end

fun fun2 $z =$ let ...
in ... end

in fun1 (fun2 $x$)

end

Back to scope names
let
val \(\textit{x} = 10\); val \(\textit{x} = \textit{x} - 5\);
fun \textit{fun1} y = let ...
in ...
end
fun \textit{fun2} z = let ...
in ...
end
fun1 (fun2 \(\textit{x}\))
end
let
val \( x = 10; \) val \( x = x - 5; \)
fun fun1
  \( y = \) let
  \( \ldots \)
  in
  \( \ldots \)
  end
fun fun2
  \( z = \) let
  \( \ldots \)
  in
  \( \ldots \)
  end
in fun1 (fun2 \( x \))
end

Back to scope names
Definition of Names

Definitions are of the form

\textit{qualifier name} \ldots = \textit{body}

- \texttt{val name} =

- \texttt{fun name ( argnames )} =

- \texttt{local definitions in definition end}
Use of Names

Names are used in expressions. Expressions may occur

• by themselves – to be evaluated
• as the *body* of a definition
• as the *body* of a let-expression

```
let definitions in expression
end
```

use of local
local...in...end

local

exception invalidArg;

fun ifdivisor3 (n, k) =
  if n <= 0 orelse
  k <= 0 orelse
  n < k
  then raise invalidArg
  else if n mod k = 0
  then k
  else 0;
fun sum_div2 (n, l, u) = 
  if n <= 0 orelse 
    l <= 0 orelse 
    l > n orelse 
    u <= 0 orelse 
    u > n 
  then raise invalidArg 
  else if l > u 
  then 0 
  else if divisor3 (n, l) 
    + sum_div2 (n, l+1, u)
fun perfect n = 
  if n <= 0 
  then raise invalidArg 
  else 
    let 
      val nby2 = n div 2 
    in 
      n = sum_div2 (n, 1, nby2) 
    end 
end
local...in...end

Standard ML of New Jersey, - use "perfect2.sml";
[opening perfect2.sml]
GC #0.0.0.0.1.10: (1 ms)
val perfect = fn : int -> bool
val it = () : unit
- perfect 28;
val it = true : bool
- perfect 6;
val it = true : bool
- perfect 8128;
val it = true : bool
Scope & local

local

fun fun1 y = ...

fun fun2 z = fun1

in

fun fun3 x = fun2 ...

fun1 ...

end
Computations: Simple

For most simple expressions it is

- **left to right**, and
- **top to bottom**

except when

- **presence of brackets**
- **precedence of operators**

determine otherwise.

Hence
Simple computations

\[4 + 6 - (4 + 6) \div 2\]
\[= 10 - (4 + 6) \div 2\]
\[= 10 - 10 \div 2\]
\[= 10 - 5\]
\[= 5\]
Computations: Composition

\[ f(x) = x^2 + 1 \]

\[ g(x) = 3 \times x + 2 \]

Then for any value \( a = 4 \)

\[ f(g(a)) = f(3 \times 4 + 2) \]
\[ = f(14) \]
\[ = 14^2 + 1 \]
\[ = 196 + 1 \]
\[ = 197 \]
Composition: Alternative

\[ f(x) = x^2 + 1 \]
\[ g(x) = 3 \times x + 2 \]

Why not

\[ f(g(a)) \]
\[ = g(4)^2 + 1 \]
\[ = (3 \times 4 + 2)^2 + 1 \]
\[ = (12 + 2)^2 + 1 \]
\[ = 14^2 + 1 \]
\[ = 196 + 1 \]
\[ = 197 \]
Compositions: Compare

\[ f(g(a)) = f(3 \times 4 + 2) = f(14) = 14^2 + 1 = 197 \]
\[ f(g(a)) = g(4)^2 + 1 = (3 \times 4 + 2)^2 + 1 = (12 + 2)^2 + 1 = 197 \]
Compositions: Compare

**Question 1**: Which is more correct? Why?

**Question 2**: Which is easier to implement?

**Question 3**: Which is more efficient?
Computations: Composition

A computation of $f(g(a))$ proceeds thus:

- $g(a)$ is evaluated to some value, say $b$
- $f(b)$ is next evaluated
Recursion

\[ f_{act\,L}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 f_{act\,L}(n - 1) \times n & \text{otherwise}
\end{cases} \]

\[ f_{act\,R}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 n \times f_{act\,R}(n - 1) & \text{otherwise}
\end{cases} \]
Recursion: Left

\[ \text{fact}_L(4) \]
\[ = (\text{fact}_L(4 - 1) \times 4) \]
\[ = (\text{fact}_L(3) \times 4) \]
\[ = (((\text{fact}_L(3 - 1) \times 3) \times 4) \]
\[ = (((\text{fact}_L(2) \times 3) \times 4) \]
\[ = (((\text{fact}_L(2 - 1) \times 2) \times 3) \times 4) \]
\[ = \ldots \]
Recursion: Right

\[ factR(4) = (4 \times factR(4 - 1)) = (4 \times factR(3)) = (4 \times (3 \times factR(3 - 1)))) = (4 \times (3 \times (2 \times factR(2)))) = (4 \times (3 \times (2 \times (1 \times factR(1))))) = \ldots \]
So Far-1: Computing

- The general nature of computation
- The notion of primitives, composition & induction
- The notion of an algorithm
- The digital computer & programming language
So Far-2: Algorithms & Programs

- **Algorithms**: Finite mathematical processes
- **Programs**: Precise, unambiguous explications of algorithms
- **Standard ML**: Its primitives
- **Writing** technically complete specifications
So far-3: Top-down Design

integer Square Root

- Begin with the function you need to design
- Write a
  - small compact technically complete definition of the function
  - perhaps using other functions that have not yet been defined
- Each function in turn is defined in a top-down manner
So Far-4: Algorithms to Programs

- Perform top development till you require only the available primitives
- Directly translate the algorithm into a Program
- Use scope rules to localize or generalize

SML code for perfect
So far-5: Caveats

- Don’t arbitrarily vary code from your algorithmic development
  - It might work or
  - It might not work
  - unless properly justified
- May destroy technical completeness
- May create scope violations.
So Far-6: Algorithmic Variations

Algorithmic Variations

- Are safe if developed from first principles. Thus ensuring their
  - mathematical correctness
  - technical completeness
  - termination properties
So Far-7: Computations

- Work within the notion of mathematical equality
  - Simple expressions
  - Composition of functions
  - Recursive computations

- But are generally irreversible
Floating Point

- Each real number \(3E11\) is represented by a pair of integers
  1. **Mantissa**: 3 or 30 or 300 or . . .
  2. **Exponent**: the power of 10 which the mantissa has to be multiplied by

- What is displayed is not necessarily the same as the internal representation.
- There is no unique representation of a real number
Real Operations

Depending upon the operations involved

- Each real number is first converted into a **suitable** representation
- The operation is performed
- The result is converted into a suitable representation for display.

skip to Numerical methods
Real Arithmetic

• for **addition** and **subtraction** the two numbers should have the same exponent for ease of **integer operations** to be performed

• the conversion may involve loss of precision

• for **multiplication** and **division** the exponents may have to be adjusted so as not to cause an **integer overflow** or underflow.
Numerical Methods

- Finite (limited) precision
- Accuracy depends upon available precision
- Whereas integer arithmetic is exact, real arithmetic is not.
- Numerical solutions are a finite approximation of the result
Errors

- Hence an estimate of the error is necessary.
- If $a$ is the “correct” value and $a^*$ is the computed value,

  **absolute error** = $a^* - a$

  **relative error** = $\frac{a^* - a}{a}$
Errors

Errors in floating point computations are mainly due to finite precision Round-off errors. It is impossible to compute the value of a (convergent) infinite series because computations are themselves finite processes. Infinite series
Infinite Series cannot be computed to $\infty$

\[ e^x = \sum_{m=0}^{\infty} \frac{x^m}{m!} \]

\[ \cos x = \sum_{m=0}^{\infty} \frac{(-1)^m x^{2m}}{(2m)!} \]

\[ \sin x = \sum_{m=0}^{\infty} \frac{(-1)^m x^{2m+1}}{(2m+1)!} \]
Truncation Errors

and hopefully it is good enough to restrict it to appropriate values of \( n \)

\[
e^x = \sum_{m=0}^{n} \frac{x^m}{m!}
\]

\[
\cos x = \sum_{m=0}^{n} \frac{(-1)^m x^{2m}}{(2m)!}
\]

\[
\sin x = \sum_{m=0}^{n} \frac{(-1)^m x^{2m+1}}{(2m + 1)!}
\]
Equation Solving

- The fifth most basic operation
- Root finding: A particular form of equation solving

\[ f(x) = 0 \]
Root Finding-1

\[ f(b) \]

\[ a \]

\[ x_0 \]

\[ f(a) \]

\[ b \]
Root Finding-2

\[ f(b) \]

\[ f(a) \]

\[ x_0 \]

\[ a \]

\[ b \]
Root Finding-3

\[ f(b) \]

\[ a \]

\[ x_0 \]

\[ f(a) \]

\[ b \]
Root Finding-4

Rather steep isn’t it?

\[ f(b) \quad \text{and} \quad f(a) \]

\[ x_0 \quad \text{and} \quad \varepsilon \]
Root Finding: Newton’s Method

Consider a function $f(x)$

- smooth and continuously differentiable over $[a, b]$
- with a non-zero derivative $f'(x)$ everywhere in $[a, b]$
- the signs of $f(a)$ and $f(b)$ are different
Root Finding: Newton’s Method

\[ f(b) \]

\[ a \]

\[ f(a) \]

\[ b \]
Root Finding: Newton’s Method
Root Finding: Newton’s Method

The graph illustrates the Newton’s Method for finding roots of a function. The iteration process is shown as:

\[ f(a) \rightarrow f(x_i) \rightarrow f(x_{i+1}) \rightarrow \cdots \]

Where \( f(a) \) and \( f(b) \) are the function values at the initial points, and \( x_i \) and \( x_{i+1} \) are the iterates that approach the root of the function.
Root Finding: Newton’s Method

\[ f(b) \]

\[ f(a) \]

\[ a \]

\[ x_{i+1} \]

\[ x_i \]

\[ b \]
Root Finding: Newton’s Method

![Newton's Method Diagram]
Newton’s Method: Basis

\[ \tan \alpha_i = f'(x_i) = \frac{f(x_i)}{x_i - x_{i+1}} \]

whence

\[ x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \]

Starting from an initial value \( x_0 \in [a, b] \), if the sequence \( f(x_i) \) converges to 0 i.e

\[ f(x_0), f(x_1), f(x_2), \cdots \rightarrow 0 \]
Newton’s Method: Basis

\[ \lim_{n \to \infty} |f(x_n)| = 0 \]

\[ i.e. \forall \varepsilon > 0 : \exists N \geq 0 : \forall n > N : |f(x_n)| < \varepsilon \]

then the sequence

\[ x_0, x_1, x_2, \ldots \]

converges to a root of \( f \).
Newton’ Method: Algorithm

Select a small enough $\varepsilon > 0$ and $x_0$. Then

$$\text{newton}(f, f', a, b, \varepsilon, x_i) =$$

$$\begin{cases} 
  x_i & \text{if } |f(x_i)| < \varepsilon \\
  \text{newton}(f, f', a, b, \varepsilon, x_{i+1}) & \text{otherwise}
\end{cases}$$

where

$$x_0 \in [a, b]$$

and

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \in [a, b]$$
What can go wrong!-1

Oscillations!

\[ f(b) \]

\[ a \]

\[ x_{i+1} \]

\[ x_i \]

\[ x_{i+2} \]
What can go wrong!-2

An intermediate point may lie outside $[a, b]$! The function may not satisfy all the assumptions outside $[a, b]$. There are then no guarantees about the behaviour of the function.
What can go wrong!-2

Interval bounds error!

\[ f(b) \]

\[ a \quad x_{i+1} \quad \alpha_i \quad x_i \quad x_{i+2} \quad b \]

\[ f(a) \]
What can go wrong!-3

The function may be too steep

for the available precision.
What can go wrong!-4

Or too shallow!

\[ f(a) + x_0 \]

\[ f(b) \]
Real Computations & Induction: 1

Newton’s method (when it does work!) computes a sequence

\[ x_0, x_1, x_2, \ldots, x_n \]

of essentially **discrete** values such that even if the sequence is not totally ordered, there is some discrete convergence **measure** viz.

\[ |f(x_i) - 0| \]

which is **well-founded**.
Real Computations & Induction: 2

That is, for some decreasing sequence of integers \( k_i \geq 0 \),

\[
k_0 > k_1 > k_2 > \cdots > k_n = 0
\]

we have

\[
k_i \varepsilon \leq |f(x_i) - 0| < (k_i + 1) \varepsilon
\]

and therefore inductive on integer multiples of \( \varepsilon \)
What’s it good for? 1

Finding the positive $n$-th root $\sqrt[n]{c}$ of a $c > 0$ and $n > 1$ amounts to solving the equation

$$x^n = c$$

which is equivalent to finding the root of $f(x)$ with

$$f(x) = x^n - c$$
$$f'(x) = nx^{n-1}$$

with $[a, b] = [0, c]$ or $[0, \sqrt{c}]$ and an appropriately chosen $\varepsilon$. 
What’s it good for? 2

Finding roots of polynomials.

\[ f(x) = \sum_{i=0}^{n} c_i x^i \]

\[ f'(x) = \sum_{i=1}^{n} ic_i x^{i-1} \]

and

- an appropriately chosen \( \varepsilon \),
- an appropriately chosen \([a, b]\) with one of the limits possibly being \( c_0 \).
newton: Computation

\[ \text{newton}(f, f', a, b, \varepsilon, x_0) \]
\[ \leadsto \text{newton}(f, f', a, b, \varepsilon, x_1) \]
\[ \leadsto \text{newton}(f, f', a, b, \varepsilon, x_2) \]
\[ \leadsto \text{newton}(f, f', a, b, \varepsilon, x_3) \]
\[ \vdots \]
\[ \leadsto \text{newton}(f, f', a, b, \varepsilon, x_n) \]
\[ \leadsto x_n \]
Generalized Composition

Computations

\[ h(x) = f(x, g(x)) \]

where

\[ f(x, y) = \begin{cases} 
0 & \text{if } x < 0 \\
y & \text{otherwise} 
\end{cases} \]

\[ g(x) = \begin{cases} 
0 & \text{if } x = 0 \\
g(x - 1) + 1 & \text{otherwise} 
\end{cases} \]
Two Computations of \( h(1) \)

\[
\begin{align*}
\text{h}(1) & \quad \text{\textasciitilde\textasciitilde} \quad f(1, \text{g}(1)) & \quad \text{\textasciitilde\textasciitilde} \quad f(1, \text{g}(1)) \\
\text{\textasciitilde\textasciitilde} g(1) & \quad \text{\textasciitilde\textasciitilde} \quad f(1, (\text{g}(0) + 1)) \\
\text{\textasciitilde\textasciitilde} g(0) + 1 & \quad \text{\textasciitilde\textasciitilde} \quad f(1, (0 + 1)) \\
\text{\textasciitilde\textasciitilde} 0 + 1 & \quad \text{\textasciitilde\textasciitilde} \quad f(1, 1) \\
\text{\textasciitilde\textasciitilde} 1 & \quad \text{\textasciitilde\textasciitilde} \quad f(1, 1)
\end{align*}
\]
Two Computations of $h(-1)$

\[
\begin{align*}
  h(-1) & \mapsto f(-1, g(-1)) & \mapsto f(-1, g(-1)) & \mapsto f(-1, (g(-2) + 1)) \mapsto f(-1, (g(-3) + 1) + 1) \mapsto \ldots \mapsto \text{FOREVER!}
\end{align*}
\]
Recursive Computations

- Newton’s method
- Factorial
  - fact\(_L\)
  - fact\(_R\)

skip to nonlinear recursion
skip to Recursion Revisited
Recursion: Left

\[ f_{actL}(4) \]
\[ \mapsto (f_{actL}(4-1) \times 4) \]
\[ \mapsto (f_{actL}(3) \times 4) \]
\[ \mapsto (((f_{actL}(3-1) \times 3) \times 4) \]
\[ \mapsto (((f_{actL}(2) \times 3) \times 4) \]
\[ \mapsto (((f_{actL}(2-1) \times 2) \times 3) \times 4) \]
\[ \mapsto \ldots \]
Recursion: Right

\[ \text{factR}(4) \]
\[ \Rightarrow (4 \star \text{factR}(4 - 1)) \]
\[ \Rightarrow (4 \star \text{factR}(3)) \]
\[ \Rightarrow (4 \star (3 \star \text{factR}(3 - 1))) \]
\[ \Rightarrow (4 \star (3 \star \text{factR}(2))) \]
\[ \Rightarrow (4 \star (3 \star (2 \star \text{factR}(2 - 1)))) \]
\[ \Rightarrow \ldots \]
Recursion: Nonlinear

\[ \text{fib}(5) \]
\[ \Rightarrow \text{fib}(4) + \text{fib}(3) \]
\[ \Rightarrow (\text{fib}(3) + \text{fib}(2)) + \text{fib}(3) \]
\[ \Rightarrow (((\text{fib}(2) + \text{fib}(1)) + \text{fib}(2)) + \text{fib}(3) \]
\[ \Rightarrow (((\text{fib}(1) + \text{fib}(0)) + \text{fib}(1)) + \text{fib}(2)) + \text{fib}(3) \]
\[ \Rightarrow (((1 + \text{fib}(0)) + \text{fib}(1)) + \text{fib}(2)) + \text{fib}(3) \]
\[ \Rightarrow \ldots \]

\textit{contd} ...
Some Practical Questions

- What is the essential **difference** between the computations of `newton` and the two factorial programs? Answer: **Constant space vs. Linear space**

- What is the essential **similarity** between the computations of `factL` and `factR`? Answer

- Why can’t we calculate beyond `fib(43)` using the definition Fibonacci, on `ccsun50` or a `P-IV`? Answer
Some Practical Questions

● What does a computation of Fibonacci look like?

● What is the essential difference between the computations of Fibonacci and newton or factL or factR?
Recursion Revisited

- Linear recursions
  - Waxing
  - Waning
- Non-linear recursion
Linear Recursion: Waxing

\[ \text{factL}(4) \]
\[ \leadsto (\text{factL}(3) \times 4) \]
\[ \leadsto (((\text{factL}(2) \times 3) \times 4) \]
\[ \leadsto (((((\text{factL}(1) \times 2) \times 3) \times 4) \]
\[ \leadsto ((((((\text{factL}(0) \times 1) \times 2) \times 3) \times 4) \]

contrast with newton
Recursion: Waning

\[ (((1 \times 1) \times 2) \times 3) \times 4 \]
\[ (((1 \times 2) \times 3) \times 4) \]
\[ ((2 \times 3) \times 4) \]
\[ (6 \times 4) \]
\[ 24 \]

contrast with newton
Nonlinear Recursions

- Each computation of $fib$ has its own waxing and waning.
- There is still an “envelope” which shows waxing and waning.
Fibonacci: contd

\[ \Rightarrow ( ((1 + 1) + fib(1)) + fib(2)) + fib(3) \]
\[ \Rightarrow (2 + fib(1)) + fib(2)) + fib(3) \]
\[ \Rightarrow ((2 + 1) + fib(2)) + fib(3) \]
\[ \Rightarrow \ldots \]
Recursion: Waxing & Waning

- **Waning**: Occurs when an expression is simplified without requiring replacement of names by definitions.

- **Waxing**: Occurs when a name is replaced by its definition.
  - name by value replacements
  - occurs in generalized composition but just once if it is not recursively defined
  - Unfolding recursion
Unfolding Recursion

- may occur several times (terminating), or

- even an infinite number of times leading to nontermination
Non-termination

• Simple expressions **never** lead to nontermination

• (Generalized) composition **never** leads to nontermination

• Recursion **may** lead to non-termination or infinite computations, unless proved **otherwise**
Termination

Since recursion may lead to nontermination

- **Termination** needs to be proved for recursive definitions, and
- for expressions and definitions that use recursively defined names as components.
Proofs of termination

A recursive definition guarantees termination

• if it is inductive, or

• it is well-founded
Proofs of termination

A recursive definition guarantees termination

- if it is **inductive**,
  Examples:
  - Factorial
  - Fibonacci

- it is **well-founded**, though not obviously inductive
Proof of termination: Factorial

Consider $f_{actL}$ defined only for non-negative integers. We prove that it is an algorithm i.e. that it terminates.

**Basis**: For $n = 0$, $f_{actL}(n) = 1$ which is not a recursive definition. Hence it does indeed terminate in a single step.
Proof of termination: Factorial

Induction hypothesis. For some \( n > 0 \), \( \forall k : 0 \leq k \leq n : \text{fact}L(k) \) terminates in \( \propto k \) steps.

Induction step. Then \( \text{fact}L(n + 1) = \text{fact}L(n) \ast (n + 1) \) is guaranteed to terminate in \( \propto (n + 1) \) steps, since \( \text{fact}L(n) \) does so in \( \propto n \) steps.
Fibonacci: Termination

The proof is similar to that of \( \text{fact}L \).

**Basis** For \( n = 0 \) or \( n = 1 \) \( \text{fib}(n) = 1 \).

**Induction hypothesis** For some \( n > 0 \), \( \forall k : 0 \leq k \leq n : \text{fib}(k) \) terminates in \( \propto f(k) \) steps

**Induction Step** Then since each of \( \text{fib}(n) \) and \( \text{fib}(n - 1) \) is guaranteed to terminate in \( \propto f(n) \) and \( \propto f(n - 1) \) steps \( \text{fib}(n + 1) = \text{fib}(n) + \text{fib}(n - 1) \) is also guaranteed to terminate in \( f(n + 1) \propto f(n) + f(n - 1) \) steps.
GCD computations

Euclidean GCD

\[ \text{gcd}(12, 64) \Rightarrow \text{gcd}(64, 12) \Rightarrow \text{gcd}(12, 4) \Rightarrow \text{gcd}(4, 0) \Rightarrow 4 \]
Well-foundedness: GCD

Euclidean GCD

For \( x, y > 0, 0 \leq x \mod y < y \). Hence the sequence of remainders obtained is

- a sequence of non-negative integers, and
- is strictly decreasing

\[ r_1 > r_2 > \cdots > r_{n-1} > r_n = 0 \]
Well-foundedness

A definition is well-founded if it is possible to define a measure (i.e. a function $w$ of its arguments) called the well-founded function such that

1. the well-founded function takes only non-negative integer values
2. with each successive recursive call the value of the well-founded function is guaranteed to decrease by at least 1.
Induction is Well-founded

The well-founded function usually is a measure of the number of computation steps that the algorithm will take to terminate

- **Factorial** $w(n) \propto n$
- **Fibonacci** $w(n) \propto f(n)$

Then
Induction is Well-founded

- $w(n)$ is always non-negative if $factL$ and $fib$ are defined only for non-negative integers.
- The argument to $factL$ and $fib$ in each recursive unfolding is strictly decreasing.
Where it doesn’t work

Such proofs do not work for

- $\text{fact}$ arbitrarily extended to include negative integers. (since $w(n)$ no longer strictly non-negative)
- $\text{fact}(n) = \text{fact}(n+1) \div (n+1)$, even if $n$ is non-negative (since $w(n)$ is no longer decreasing) since the function is no longer well-founded.
Well-foundedness is inductive

But the induction variable is

- hidden or
- too complex to worry about, or
- it serves no useful purpose for the algorithm, except as a counter.
Well-foundedness is inductive

Given any well-founded function \( w(\vec{x}) \) whose values form a decreasing sequence in some algorithm

\[
y_0 > y_1 > \cdots > y_{n-1} > y_n \geq 0
\]

it is possible to put this sequence in 1-1 correspondence with the set \( \{0, \ldots, n\} \) via a function \( \text{ind} \) such that

\[
\text{ind}(w(\vec{x})) = n - i
\]
GCD: Well-foundedness

Well-founded function for $gcd$

$$w(a, b) = b$$
Newton: Well-foundedness

Newton’s Method

Convergence condition

\[ f(x_0), f(x_1), f(x_2), \ldots \rightarrow 0 \]

Compute the discrete value sequence

\[ x_0, x_1, x_2, \ldots x_n \]

such that

\[ k_0 > k_1 > k_2 > \cdots > k_n = 0 \]

where
Newton: Well-foundedness

Newton’s Method

\[ k_i \varepsilon \leq |f(x_i) - 0| < (k_i + 1)\varepsilon \]

and therefore inductive on integer multiples of \( \varepsilon \) Hence

\[ w(x) = \left\lfloor \frac{|f(x)|}{\varepsilon} \right\rfloor \]
Example: Zero

A peculiar way to define the zero function

\[
\text{zero}(x) = \begin{cases} 
\text{zero}(x + 1.0) & \text{if } x \leq -1.0 \\
0.0 & \text{if } -1.0 < x < 1.0 \\
\text{zero}(x - 1.0) & \text{if } x \geq 1.0
\end{cases}
\]

\[
w(x) = \lceil |x| \rceil
\]
is the well-founded function.
Questions

Q: Is it always possible to find a well-founded function for each algorithm?

A: Unfortunately not! However if we can’t then we cannot call it an algorithm!. But if we can then we are guaranteed that the algorithm will terminate.

The Collatz Problem
The Collatz Problem

Does the following algorithm terminate?

\[ \text{collatz}(m) = \begin{cases} 
1 & \text{if } m \leq 1 \\
\text{collatz}(m \text{ div } 2) & \text{if } m \text{ is even} \\
\text{collatz}(3 \times m + 1) & \text{otherwise}
\end{cases} \]

*Unproven Claim.* \( \text{collatz}(m) \rightarrow 1 \) for all \( m \).
Questions

Q: what other uses can well-founded functions be put to?
A: They can be used to estimate the complexity of your algorithm in order of magnitude terms.

**Space Complexity**: The amount of memory space required, as a function of the input

**Time Complexity**: The amount of time (number of computation steps) as a function of the input
Space Complexity

What is the space complexity of

- Newton’s method
- Euclidean GCD
- Factorial
- Fibonacci
Newton’s method (wherever and whenever it works well) requires space to compute:

- the value of $f$ at each point $x_i$
- the value of $f'$ at each point $x_i$
- the value of $x_{i+1}$ from the above

Their absolute space requirements could be different. But . . .
Newton & Euclid: Relative

Newton’s Method
Computation

GCD and Newton’s method (wherever and whenever it works well) require the same amount of space for each recursive unfolding since each fresh unfolding can reuse the space used by the previous one.

Euclidean GCD
Computation
Deriving space requirements

We may use the algorithm itself to derive the space required as follows:

Assume that memory proportional to calculating and outputting the answer is a unit. Then space as a function of the input is given by
GCD: Space

\[ S_{gcd}(a, b) = \begin{cases} 1 & \text{if } b = 0 \\ S_{gcd}(b, a \mod b) & \text{otherwise} \end{cases} \]

This implies (from well-foundedness) that the entire computation ends with the space of a unit.

\[ S_{gcd}(a, b) \propto 1 \]

A similar analysis and result holds for \textit{newton}
Factorial: Space

$S_{\text{fact}L}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
S_{\text{fact}L}(n-1) + 1 & \text{otherwise}
\end{cases}$

The 1 is for output and the +1 is because one needs to store space proportional to remembering “multiply by $n$”.

$S_{\text{fact}L}(n) \propto n$.

A similar analysis and result holds for $\text{fact}R$. 
Fibonacci: Space

\[ S_{fib}(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ S_{fib}(n-1) + S_{fib}(n-2) & \text{if } n > 1 \end{cases} \]
It is easy to see prove by induction that for $n > 1$,

$$S_{fib}(n-1) < S_{fib}(n) \leq 2S_{fib}(n-1)$$

That is, as the value of $n$ increases by 1 the space requirement approximately doubles. Further, it is easy to show by induction that

$$2^{n-2} < S_{fib}(n) \leq 2^{n-1}$$
Recapitulation

• Recursion & nontermination
• Termination & well-foundedness
• Well-foundedness proofs
• Well-foundedness & Complexity
Recapitulation

- Recursion & nontermination
- Termination & well-foundedness
- Well-foundedness proofs
  - By induction
  - well-founded functions
  - By well-founded functions
  - induction as well-foundedness
  - Well-foundedness as induction
- Well-foundedness & Complexity
Time & Space Complexity

Questions

An order of magnitude estimate of the time or space (memory) required (in terms of some large computation steps).

- Newton & Euclid’s GCD
- Deriving space requirements
  - Integer Sqrt
  - Factorial
  - Fibonacci
\( \text{isqrt: Space} \)

Integer Sqrt \quad \text{shrink}

\[
S_{\text{isqrt}}(n) = S_{\text{shrink}}(n, 0, n) \quad \text{for large } n.
\]

\[
S_{\text{shrink}}(n, l, u) = \begin{cases} 
1 & \text{if } l = u \\
S_{\text{shrink}}(n, l+1, u) & \text{if } l < u \\
S_{\text{shrink}}(n, l, u-1) & \text{if } l < u
\end{cases}
\]

Assuming 1 unit of space for output.

By induction on \( |[l, u]| \)

\[
S_{\text{isqrt}}(n) = S_{\text{shrink}}(n, 0, n) \propto 1
\]
Time Complexity

As in the case of space we may use the algorithm itself to derive the time complexity.

- Integer sqrt
- Factorial
- Fibonacci
isqrt: Time Complexity

Assume condition-checking (along with $+1$ or $-1$) takes a unit of time.

Then

$$T_{shrink}(n,l,u) = \begin{cases} 
0 & \text{if } l = u \\
1 + T_{shrink}(n,l+1,u) & \text{if } l < u \\
1 + T_{shrink}(n,l,u-1) & \text{if } l < u 
\end{cases}$$

Then

$$T_{shrink}(n,l,u) \propto |[l, u]| - 1$$

and

$$T_{isqrt}(n) = T_{shrink}(n,0,n) \propto n$$
isqrt2. Time

Assume condition-checking (along with \((l + u) \div 2\)) takes a unit of time. Then

\[ T_{shrink2}(n, l, u) = \begin{cases} 
0 & \text{if } u \leq l \leq u \\
1 + T_{shrink2}(n, m, u) & \text{if } m^2 \leq n \\
1 + T_{shrink2}(n, l, u-1) & \text{if } m^2 > n
\end{cases} \]

If \(2^{k-1} \leq \| [l, u] \| - 1 < 2^k\) then the algorithm terminates in at most \(k\) steps.

Since \(k = \lceil \log_2 \| [l, u] \| - 1 \rceil\),

\[ T_{shrink2}(n, l, u) \propto \lceil \log_2 \| [l, u] \| - 1 \rceil \]

\[ T_{isqrt2}(n) \propto \lceil \log_2 n \rceil \]
$\textit{shrink vs. shrink}_2$: Times

1. The time units are different,
2. But they differ by a constant factor at most.
3. So clearly, for large $n$, $\textit{shrink}_2$ is faster than $\textit{shrink}$
4. But for small $n$, it depends on the constant factor.
5. Implicitly assume that the actual unit of time includes the time required to unfold the recursion.
Factorial: Time Complexity

Here we assume multiplication takes unit time.

$$T_{fact}(n) = \begin{cases} 
0 & \text{if } n = 0 \\
T_{fact}(n-1) + 1 & \text{otherwise}
\end{cases}$$

Then

$$T_{fact}(n) \propto n$$
Fibonacci: Time Complexity

Assuming addition and condition-checking together take a unit of time, we have

\[ T_{fib}(n) = \begin{cases} 
0 & \text{if } n \leq 1 \\
T_{fib}(n-1) + T_{fib}(n-2) & \text{if } n > 1
\end{cases} \]

It follows that

\[ 2^{n-2} < T_{fib}(n) \leq 2^{n-1} \]
### Comparative Complexity

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Space</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$isqrt(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$isqrt2(n)$</td>
<td>$O(1)$</td>
<td>$O(\log_2 n)$</td>
</tr>
<tr>
<td>$factL(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$fib(n)$</td>
<td>$O(2^n)$</td>
<td>$O(2^n)$</td>
</tr>
</tbody>
</table>
Comparisons

For smaller values
Comparisons

For large values
Efficiency Measures: Time

An algorithm for a problem is asymptotically faster or asymptotically more time-efficient than another for the same problem if its time complexity is bounded by a function that has a slower growth rate as a function of the value of its arguments.
Efficiency Measures: Space

Similarly an algorithm is asymptotically more space efficient than another if its space complexity is bounded by a function that has a slower growth rate.
Q: Can fibonacci be speeded up or made more space efficient?

A: Perhaps by studying the nature of the function e.g. $\text{isqrt}2$ vs. $\text{isqrt}$ and attempting more efficient algorithmic variations.
Speeding Up: 2

Q: Are there general methods of speeding up or saving space?

A: Take inspiration from \textit{gcd}, \textit{newton}, \textit{shrink}
Factoring out calculations

\[ \gcd(a_0, b_0) \]
\[ \text{compute } a_1, b_1 \]
\[ \Rightarrow \gcd(a_1, b_1) \]
\[ \text{compute } a_2, b_2 \]
\[ \Rightarrow \gcd(a_2, b_2) \]
\[ \Rightarrow \ldots \]
\[ \Rightarrow \gcd(a_n, b_n) \]
\[ \Rightarrow a_n \]
Tail Recursion: 1

- Factor out calculations and remember only those values that are required for the next recursive call.
- Create a vector of state variables and include them as arguments of the function.
Tail Recursion: 2

- Try to reorder the computation using the state variables so as to get the next state completely defined.

- Redefine the function entirely in terms of the state variables so that the recursive call is the outermost operation.
Factorial: Tail Recursion

\[ \text{factL Waxing} \quad \text{factL Waning} \]

- The recursive call precedes the multiplication operation. *Change it!*
- Define a \textit{state} variable \( p \) which contains the product of all the values that one must remember.
- \textbf{Reorder} the computation so that the computation of \( p \) is performed before the recursive call.
- For that \textbf{redefine} the function in terms of \( p \).
Factorial: Tail Recursion

\[ f_{actL2}(n) = \begin{cases} 
\perp & \text{if } n < 0 \\
1 & \text{if } n = 0 \\
\end{cases} \]

where

\[ f_{actL\_tr}(n, p) = \begin{cases} 
p & \text{if } n = 0 \\
f_{actL\_tr}(n - 1, np) & \text{otherwise} \\
\end{cases} \]
A Computation

\[ \text{fact}\text{L2}(4) \]
\[ \leadsto \text{fact}\text{L}\_\text{tr}(4, 1) \]
\[ \leadsto \text{fact}\text{L}\_\text{tr}(3, 4) \]
\[ \leadsto \text{fact}\text{L}\_\text{tr}(2, 12) \]
\[ \leadsto \text{fact}\text{L}\_\text{tr}(1, 24) \]
\[ \leadsto \text{fact}\text{L}\_\text{tr}(0, 24) \]
\[ \leadsto 24 \]

Reminiscent of \textit{gcd} and \textit{newton}!
Factorial: Issues

• **Correctness**: Prove (by induction on $n$) that for all $n \geq 0$, $\text{factL2}(n) = n!$.

• **Termination**: Prove by induction on $n$ that every computation of $\text{factL2}$ terminates.

• **Space complexity**: Prove that $S_{\text{factL2}}(n) = O(1)$ (as against $S_{\text{factL}}(n) \propto n$).

• **Time complexity**: Prove that $T_{\text{factL2}}(n) = O(n)$.
Fibonacci: Tail Recursion

- Remove **duplicate** computations by defining appropriate state variables
- Let $a$ and $b$ be the consecutive fibonacci numbers $fib(m - 2)$ and $fib(m - 1)$ required for the computation of $fib(m)$.
- The **state** consists of the variables $n, a, b, m$. 
Fibonacci: Tail Recursion

\[ fib_{TR}(n) = \begin{cases} \bot & \text{if } n < 0 \\ 1 & \text{if } 0 \leq n \leq 1 \\ fib_{iter}(n, 1, 1, 1) & \text{otherwise} \end{cases} \]

where

\[ fib_{iter}(n, a, b, m) = \begin{cases} b & \text{if } m \geq n \\ fib_{iter}(n, b, a + b, m + 1) & \text{otherwise} \end{cases} \]
fibTR: SML

local

fun fib_iter (n, a, b, m) =
  (* fib (m) = b , fib (m-1) = a *)
  if m >= n then b
  else fib_iter (n, b, a+b, m+1);

in

fun fibTR (n) =
  if n < 0 then raise negativeArgument
  else if (n <= 1) then 1
  else fib_iter (n, 1, 1, 1, 1)
end;
State in Tail Recursion

- The variables that make up the state bear a definite relation to each other.

- INVARIANCE. That relationship between the state variables does not change throughout the computation of the function.
Invariance

- The **invariant** property of a tail-recursive function must hold **Initially** when it is first invoked, and **Continues** to hold before every successive invocation.

- The **invariant** property characterizes the entire computation and the algorithm and is crucial to the proof of correctness.
Recap

- Asymptotic Complexity:
  - Space
  - Time

- Comparative Complexity

- Comparisons:
  - Small inputs
  - Large inputs
Recursion Transformation

- To achieve constant space and linear time, if possible
- Speeding up using tail recursion
  - Factor out calculations
  - Reorder the computations with state variables
  - Recursion as the outermost operation
Tail Recursion: Examples

- Factorial vs. Factorial: \( \text{factL} \) vs. \( \text{factL2} \) vs.
- Fibonacci vs. Fibonacci: \( \text{fib} \) vs. \( \text{fibTR} \)
# Comparisons

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Space</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(isqrt(n))</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>(isqrt2(n))</td>
<td>(O(1))</td>
<td>(O(\log_2 n))</td>
</tr>
<tr>
<td>(factL(n))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>(factL2(n))</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>(fib(n))</td>
<td>(O(2^n))</td>
<td>(O(2^n))</td>
</tr>
<tr>
<td>(fibTR(n))</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>
Transformation Issues

- **Correctness**: Prove that the new algorithm computes the same function as the original simple algorithm.
- **Termination**: Prove by induction on $n$ that every computation is finite.
- **Space complexity**: Compute it.
- **Time complexity**: Compute it.
Correctness Issues 1

- **Absolute correctness:** For any function \( f \), that an algorithm \( A \) that claims to implement it, prove that
\[
f(x) = A(x)
\]
for all argument values \( x \) for which \( f \) is defined.

- **Transformation correctness:**
Correctness Issues 2

- **Absolute correctness:**
- **Transformation correctness:** For any algorithm $A$ and a transformed algorithm $B$ prove that $A(\vec{x}) = B(\vec{x})$ for all argument values $\vec{x}$ for which $A$ is defined. Then $B$ is absolutely correct provided $A$ is absolutely correct.
Correctness Theorem

Invariant properties \( \text{factL}_2 \)

**Theorem 1** For all \( n \geq 0 \),
\[
\text{factL}_2(n) = n!
\]

**Proof:** For \( n = 0 \), it is clear that \( \text{factL}_2(0) = 1 = 0! \). For \( n > 0 \),
\[
\text{factL}_2(n) = \text{factL}_\text{tr}(n, 1).
\]
The proof is done provided we can show that
\[
\text{factL}_\text{tr}(n, 1) = n!.
\]
Invariants & Correctness 1

Invariant properties $factL2$

- To prove the absolute or transformation correctness of a tail-recursion transformation usually requires an invariant property to be proven about the tail-recursive function.
Invariants & Correctness 2

Invariant properties $fact_{L2}$

• This allows the independent proof of the properties of the tail-recursive function without reference to the function that uses it.

• It reflects the design of the algorithm and its division into sub-problems.
Invariance Lemma: $\text{factL}_\text{tr}$

Invariant properties $\text{factL}_2$

Lemma 2 For all $n \geq 0$ and $p$

$$\text{factL}_\text{tr}(n, p) = (n!)p$$

Proof: By induction on $n$. □

Back to theorem
Invariance: Example

\[ \text{fact}_{L_{tr}}(4, 7) \]
\[ \implies \text{fact}_{L_{tr}}(3, 28) \]
\[ \implies \text{fact}_{L_{tr}}(2, 84) \]
\[ \implies \text{fact}_{L_{tr}}(1, 168) \]
\[ \implies \text{fact}_{L_{tr}}(0, 168) \]
\[ \implies 168 \]

Contrast with a \( \text{fact}_{L_{2}}(4) \) computation
Invariance: Example

So what exactly \textit{is} invariant?

\[
factL_{tr}(4, 7) \quad 168 = 4! \times 7
\]
\[
\rightsquigarrow factL_{tr}(3, 28) \quad 168 = 3! \times 28
\]
\[
\rightsquigarrow factL_{tr}(2, 84) \quad 168 = 2! \times 84
\]
\[
\rightsquigarrow factL_{tr}(1, 168) \quad 168 = 1! \times 168
\]
\[
\rightsquigarrow factL_{tr}(0, 168) \quad 168 = 0! \times 168
\]
\[
\rightsquigarrow 168
\]
Proof

Basis For \( n = 0 \), \( \text{factL_tr}(0, p) = p = (0!)p \).

Induction hypothesis (IH) For all \( k \),
\( 0 < k \leq n \), \( \text{factL_tr}(k, p) = p = (k!)p \)

Induction Step

\[
\text{factL_tr}(n + 1, p) = \text{factL_tr}(n, (n + 1)p) = (n!)(n + 1)p = (n + 1)!p
\]

Back to lemma
Invariance Lemma: \( \text{fib}_\text{iter} \)

**Lemma 3** For all \( n > 1, a, b, m : 1 \leq m \leq n \), if \( a = F(m - 1) \) and \( b = F(m) \), then

\[
\text{INV}: \text{fib}_\text{iter}(n, a, b, m) = F(n)
\]

**Proof:** By induction on \( k = n - m \) \( \Box \)
Proof

**Basis** For $k = 0$, $n = m$, it follows that
\[ \text{fib\_iter}(n, a, b, m) = F(n) \]

**Induction hypothesis (IH)** For all $n > 1$ and $1 \leq m \leq n$, with $n - m \leq k$, INV holds

**Induction Step** Let $1 \leq m < n$ such that $n - m = k + 1$, $F(m) = b$ and $F(m - 1) = a$. Then $F(m + 1) = a + b$ and
\[
\text{fib\_iter}(n, a, b, m) = \text{fib\_iter}(n, b, a + b, m + 1) = F(n) \quad (IH)
\]
Correctness: Fibonacci

\[ \text{Theorem 4 } \text{For all } n \geq 0, \ fibTR(n) = F(n). \]

\textbf{Proof: } For \( 0 \leq n \leq 1 \), it holds trivially. For \( n > 1 \), \( fibTR(n) = fib\_iter(n, 1, 1, 1) = F(n) \), by the invariance lemma, with \( m = 1, a = 1 = F(m - 1) \) and \( b = 1 = F(m) \). \( \square \)
Variants & Invariants

\[ \text{factL3}(n) = \begin{cases} 
\bot & \text{if } n < 0 \\
1 & \text{if } n = 0 \\
\text{factL_tr2}(n, 1, 1) & \text{else}
\end{cases} \]

where
Variants & Invariants

\[ \text{fact}_L^{\text{tr}2}(n, p, m) = \begin{cases} 
p \\
\text{fact}_L^{\text{tr}2}(n, (m+1)p, m+1) 
\end{cases} \]

\[ \text{fact}_L^{\text{tr}2}(n, p, m) = (m!)^p \]

for all \( 1 \leq m \leq n \).
More Invariants

- *shrink* For all $n > 0$, $l$, $u$, if $[l, u] \subseteq [0, n]$,
  \[ l \leq \lfloor \sqrt{n} \rfloor \leq u \]

- *shrink2*
  For all $n > 0$, $l$, $u$, if $[l, u] \subseteq [0, n]$,
  \[ m = \lfloor (l + u)/2 \rfloor \text{ and } l \leq \lfloor \sqrt{n} \rfloor \leq u \]
Fast Powering 1

\[ \text{power}_3(x, n) = \begin{cases} 
1.0 / \text{power}_3(x, -n) & \text{if } n < 0 \\
1.0 & \text{if } n = 0 \\
\text{power}_T R(x, n, 1) & \text{else}
\end{cases} \]

where
Fast Powering 2

\[ \text{power}^2 \]

\[
\text{powerTR}(x, n, p) = \begin{cases} 
  p & \text{if } n = 0 \\
  \text{powerTR}(x^2, n \text{ div } 2, p) & \text{if even}(n) \\
  \text{powerTR}(x^2, n \text{ div } 2, xp) & \text{otherwise}
\end{cases}
\]

where \( \text{even}(n) \iff n \mod 2 = 0 \).

\[ \boxed{\text{powerTR}(x, n, p) = x^n p} \]
Root Finding: Bisection

Newton's Method Algorithm

Select a small enough $\epsilon > 0$ and $x_0$. Then if $sgn(f(a)) \neq sgn(f(b))$, 

$$\text{bisect}(f, a, b, \epsilon) = \begin{cases} 
c & \text{if } |f(c)| < \epsilon \\
\text{bisect}(f, c, b, \epsilon) & \text{if } sgn(f(c)) \neq sgn(f(b)) \\
\text{bisect}(f, a, c, \epsilon) & \text{otherwise}
\end{cases}$$

where $c = (a + b)/2$
Advantage Bisection

More **robust** than Newton’s method

- Requires continuity and change of sign
- Does not require differentiability
- Could change the condition suitably to take care of very shallow curves
- **Oscillations** could occur only if the function is too **steep**.
- An intermediate point can never go outside the interval.
Recap: Tail Recursion

- Asymptotic Complexity:
  - **Time** Linear
  - **Space** Constant

- Correctness: Capture the algorithm through
  - **Invariant** Invariance Lemma
  - **Bound function** Proof by induction
Examples: Invariants

$\text{fact} L_{tr2}$

$\text{shrink} \ & \ shrink^2$

\[
\begin{align*}
l \leq \lfloor \sqrt{n} \rfloor & \leq u \\
m = \lceil (l + u)/2 \rceil \quad \text{and} \quad l \leq \lfloor \sqrt{n} \rfloor & \leq u
\end{align*}
\]

Fast Powering

\[
\text{power}^{TR}(x, n, p) = x^n p
\]
Tuples

\[ \text{divmod}(a, b) = (a \div b, a \mod b) \]

\text{divmod: int * int -> int * int}
Lists

An \textit{\alpha list} represents a sequence of elements of a given type \( \alpha \).

Given a (nonempty) list

- A list is \textbf{ordered}
- There may be \textbf{more than one occurrence of an element} in the list
- \textbf{only the first element} (called the \textit{head}) of the list is immediately accessible.
New Lists

Given a (nonempty) list $L$,

- A new list $M$ may be created from an existing list $L$ by the $tl$ operation.
- New elements can be added (by the operation $cons$) to an existing list, one at a time to create new lists.
- The last element that was added becomes the head of the new list.
- Two lists are equal only if they have the same elements in the same order.
List Operations

- The empty list: \textit{nil} or \texttt{[]}

- Nonempty lists: Given a nonempty list \( L \)

\[
L = [1, 2, 3, 4]
\]

\textbf{head} : \( \textit{hd} : \alpha \text{List} \rightarrow \alpha \)

\[
\textit{hd}(L) = 1
\]

\textbf{tail} : \( \textit{tl} : \alpha \text{List} \rightarrow \alpha \text{List} \)

\[
\textit{tl}(L) = [2, 3, 4]
\]
List Operations: \textit{cons}

1. \( L = [1, 2, 3, 4] \)

\begin{align*}
\text{\textbf{cons}} & : \alpha \times \alpha\textbf{List} \rightarrow \alpha\textbf{List} \\
\text{cons}(0, \texttt{nil}) & = [0] \\
\text{cons}(0, L) & = 0 :: L = [0, 1, 2, 3, 4] \\
1 :: (0 :: L) & = [1, 0, 1, 2, 3, 4]
\end{align*}

back to lists Recap
Generating Primes upto $n$

Definition 5 A positive integer $n > 1$ is **composite** iff it has a **proper** divisor $d \mid n$ with $1 < d < n$. Otherwise it is **prime**.

- $2$ is the smallest (first) prime.
- $2$ is the only even prime.
- No other even number can be a prime.
- All other primes are odd.
More Properties

• An odd number cannot have any even divisors.
• Every number may be expressed uniquely (upto order) as a product of prime factors.
• No divisor of a number can be greater than itself.
• For each divisor $d|n$ such that $d \leq \lfloor \sqrt{n} \rfloor$, $n/d \geq \lfloor \sqrt{n} \rfloor$ is also a divisor.
Composites

- If a number $n$ is composite, then it has a proper divisor $d$, $2 \leq d \leq \lfloor \sqrt{n} \rfloor$.
- If a number $n$ is composite, then it has a prime divisor $p$, $2 \leq p \leq \lfloor \sqrt{n} \rfloor$.
- An odd composite number $n$ has an odd prime divisor $p$, $3 \leq p \leq \lfloor \sqrt{n} \rfloor$. 
Odd Primes

• An odd number $\geq 1$ is a prime iff it has no proper odd divisors.

• An odd number $\geq 1$ is a prime iff it is not divisible by any odd prime smaller than itself.

• An odd number $n > 1$ is a prime iff it is not divisible by any odd prime $\leq \lfloor \sqrt{n} \rfloor$. 
$\text{primesUpto}(n) = \begin{cases} \emptyset & \text{if } n < 2 \\ [(1, 2)] & \text{if } n = 2 \\ \text{primesUpto}(n - 1) & \text{elseif } \text{even}(n) \\ \text{generateFrom} & \text{otherwise} \\ ((1, 2), 3, n, 2) & \end{cases}$

where
generateFrom$(P, m, n, k)$

bound function $n - m$

Invariant

$$2 < m \leq n \land \text{odd}(m)$$

implies

$$P = [(k - 1, p_{k-1}), \cdots, (1, p_1)]$$

and

$$\forall q : p_{k-1} < q < m : \text{composite}(q)$$
\[
\text{generateFrom} \quad \text{generateFrom}(P, m, n, k) = \begin{cases} 
P & \text{if } m > n \\
\text{generateFrom} \quad \text{generateFrom}(((k, m) :: P), m + 2, n, k + 1) & \text{elseif} \\
\text{generateFrom} \quad \text{generateFrom}(P, m + 2, n, k) & \text{else}
\end{cases}
\]

where \( \text{pwrt} = \text{primeWRT}(m, P) \)
Definition 6 A number $m$ is prime with respect to a list $L$ of numbers iff it is not divisible by any of them.

- A number is prime iff it is prime with respect to the list of all primes smaller than itself.
- From properties of odd primes it follows that a number $n$ is prime iff it is prime with respect to the list of all primes $\leq \sqrt{n}$
primeWRT(m, P)

bound function \textit{length}(P)

**Invariant** If \( P = [(i - 1, p_{i-1}), \ldots (1, p_1)] \), for some \( i \geq 1 \) then

- \( p_k \geq m > p_{k-1} \), and
- \( m \) is prime with respect to \([((k - 1, p_{k-1}), \ldots, (i, p_i)]\)
- \( m \) is a prime iff it is a prime with respect to \( P \)
\textit{primeWRT}

\[ \text{\textit{primeWRT}}(m, P) = \begin{cases} 
\text{true} & \text{if } P = \text{nil} \\
\text{false} & \text{elseif } h | m \\
\text{\textit{primeWRT}}(m, \text{tl}(P)) & \text{else} 
\end{cases} \]

where

\[(i, h) = \text{hd}(P)\]

for some \(i > 0\)
Density of Primes

Let $\pi(n)$ denote the number of primes upto $n$. Then

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\pi(n)$</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>25</td>
<td>25.00%</td>
</tr>
<tr>
<td>1000</td>
<td>168</td>
<td>16.80%</td>
</tr>
<tr>
<td>10000</td>
<td>1229</td>
<td>12.29%</td>
</tr>
<tr>
<td>100000</td>
<td>9592</td>
<td>9.59%</td>
</tr>
<tr>
<td>1000000</td>
<td>78,498</td>
<td>7.85%</td>
</tr>
<tr>
<td>10000000</td>
<td>664,579</td>
<td>6.65%</td>
</tr>
<tr>
<td>100000000</td>
<td>5,761,455</td>
<td>5.76%</td>
</tr>
</tbody>
</table>
The Prime Number Theorem

\[ \lim_{n \to \infty} \frac{\pi(n)}{n/\ln n} = 1 \]

Proved by Gauss.

- Shows that the primes get sparser at higher \( n \)
- A larger percentage of numbers as we go higher are composite.

from David Burton: *Elementary Number Theory.*
The Prime Number Theorem

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$\pi(n)$</td>
<td>%</td>
<td>$\lim_{n \to \infty} \frac{\pi(n)}{n/\ln n}$</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>25.00%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>168</td>
<td>16.80%</td>
<td>1.159</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>1229</td>
<td>12.29%</td>
<td>1.132</td>
<td></td>
</tr>
<tr>
<td>100000</td>
<td>9592</td>
<td>9.59%</td>
<td>1.104</td>
<td></td>
</tr>
<tr>
<td>1000000</td>
<td>78,498</td>
<td>7.85%</td>
<td>1.084</td>
<td></td>
</tr>
<tr>
<td>10000000</td>
<td>664579</td>
<td>6.65%</td>
<td>1.071</td>
<td></td>
</tr>
<tr>
<td>100000000</td>
<td>5761455</td>
<td>5.76%</td>
<td>1.061</td>
<td></td>
</tr>
</tbody>
</table>

from David Burton: *Elementary Number Theory.*
## Complexity

<table>
<thead>
<tr>
<th>function</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>primesUpto</code></td>
<td>1</td>
</tr>
<tr>
<td><code>generateFrom</code></td>
<td>(\frac{n}{2})</td>
</tr>
<tr>
<td><code>primeWRT</code></td>
<td>(\sum_{m=3, \text{odd}(m)}^{n} \pi(m))</td>
</tr>
</tbody>
</table>

\(\pi(m)\) denotes the number of primes less than or equal to \(m\).
Diagnosis

For each $m \leq n$,

- $P$ is in **descending** order of the primes
- $m$ is checked for divisibility $\pi(m)$ times
- From properties of odd primes it should not be necessary to check each $m$ more than $\pi(\lfloor \sqrt{m} \rfloor)$ times for divisibility.
- Organize $P$ in **ascending** order instead of **descending**.
Compound Data

- Forming (compound) data structures from simpler ones
- Breaking up compound data into its components.
Recap: Tuples

formation: Cartesian products of types

selection: Selection of individual components

equality: Equality checking

equality errors: Equality errors forward to Lists
Tuple: Formation

Standard ML of New Jersey,
- val a = ("arun", 1<2, 2);
val a = ("arun",true,2)
  : string * bool * int
- val b = ("arun", true, 2);
val b = ("arun",true,2)
  : string * bool * int
Tuples: Selection

- #2 a;
  val it = true : bool
- #1 a;
  val it = "arun" : string
- #3 a;
  val it = 2 : int
- #4 a;
  stdIn:1.1-1.5 Error: operator
    operator domain: {4:'Y; 'Z}
    operand: string * bool
    in expression:
      (fn {4=4,...} => 4) a
Tuples: Equality

- \( a = b \);
val it = true : bool
- \((1 < 2, \text{true}) = (1.0 < 2.0, \text{true})\);
val it = true : bool
- \((\text{true}, 1.0 < 2.4)\)
  = \((1.0 < 2.4, \text{true})\);
val it = true : bool
Tuples: Equality errors

- ("arun", (1, true))
= ("arun", 1, true);

stdin:1.1-29.39 Error: operator and operand do not agree
operator domain: (string * (int * bool)) * (string * (int * bool))
operand: (string * (int * bool)) * (string * int * bool)
in expression:
  ("arun", (1, true))
= ("arun", 1, true)
- ("arun", (1, true))
= (("arun", 1), true);

stdin:1.1-29.39 Error: operator and operand do not agree
Lists: Recap

**formation**: Sequence \( \alpha \) List

**selection**: Selection of individual components

**new lists**: Making new lists from old
Lists: Append

- op @;
val it = fn : 'a list * 'a list
    -> 'a list
- [1,2,3] @ [~1, ~3];
val it = [1,2,3,~1,~3]
  : int list
- [[1,2,3], [~1, ~2]]
@ [[1,2,3], [~1, ~2]]; 
val it = 
[[1,2,3], [~1,~2],
 [1,2,3], [~1,~2]]
  : int list list
**cons** VS. **@**

**cons** is a constant time = $O(1)$ operation. But **@** is linear = $O(n)$ in the length $n$ of the first list. **@** is defined as

$$L@M = \begin{cases} 
M & \text{if } L = \text{nil} \\
h :: (T@M) & \text{if } L = h :: T 
\end{cases}$$
Lists of Functions

- fun add1 x = x + 1;
val add1 = fn : int -> int

- fun add2 x = x + 2;
val add2 = fn : int -> int

- fun add3 x = x + 3;
val add3 = fn : int -> int
Lists of Functions

- val addthree
  = [add1, add2, add3];
val addthree
  = [fn,fn,fn] : (int -> int) list
- fun addall x = [(add1 x), (add2 x)];
val addall = fn : int -> int list
- addall 3;
val it = [4,5,6] : int list
Arithmetic Sequences

\[ AS1(a, d, n) = \begin{cases} 
\emptyset & \text{if } n \leq 0 \\
AS1(a, d, n - 1) & \text{else} \\
[a + (n - 1) \times d] 
\end{cases} \]

<table>
<thead>
<tr>
<th>function</th>
<th>calls</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>( AS1 )</td>
<td>( n )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>( @ )</td>
<td>( n )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>::</td>
<td>( \sum_{i=0}^{n} i )</td>
<td>( O(n^2) )</td>
</tr>
</tbody>
</table>
Tail Recursion

$$AS2(a, d, n) = \begin{cases} \emptyset & \text{if } n \leq 0 \\ AS2\_iter(a, d, n - 1, 0, []) & \text{else} \end{cases}$$

where

for any initial $L_0$ and $n \geq k \geq 0$

$$INV2 : L = L_0 @ [a] @ \ldots @ [a + (k - 1) \times d]$$
Tail Recursion

\[ \text{INV2}: L = L_0[a]@\ldots@[a + (k - 1) \ast d] \]

and bound function

\[
\begin{align*}
\text{AS2_iter}(a, d, n, k, L) &= \\
&= \begin{cases} 
L & \text{if } k \geq n \\
\text{AS2_iter}(a, d, n, k + 1) & \text{else} \\
L[a + k \ast d] & 
\end{cases}
\end{align*}
\]
Tail Recursion

<table>
<thead>
<tr>
<th>function</th>
<th>calls</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AS2$</td>
<td>1</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$AS2_{iter}$</td>
<td>$n$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>@</td>
<td>$n$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>::</td>
<td>$\sum_{i=0}^{n} i$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

So tail recursion simply doesn’t help!
Another Tail Recursion

\[ AS3(a, d, n) = \]

\[
\begin{cases} \\
\text{[]} & \text{if } n \leq 0 \\
AS3\_iter(a, d, n - 1, []) & \text{else}
\end{cases}
\]

where

for any initial \( L_0, n_0 \geq n > 0 \), and

\[ INV3 : L = (a + (n - 1) \ast d) :: \cdots :: \]

\[ \cdots :: (a + (n_0 - 1) \ast d) :: L_0 \]
Another Tail Recursion

\[
INV3 : L = (a + (n - 1) \ast d) :: \cdots \\
\cdots :: (a + (n_0 - 1) \ast d) :: L_0
\]

and bound function \(n\),

\[
AS3_iter(a, d, n, L) =
\begin{align*}
L & \quad \text{if } n \leq 0 \\
AS3_iter(a, d, n - 1, (a + (n - 1) \ast d) :: L) & \quad \text{else}
\end{align*}
\]
## AS3: Complexity

<table>
<thead>
<tr>
<th>function</th>
<th>calls</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS3</td>
<td>1</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>AS3_iter</td>
<td>$n$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>@</td>
<td>0</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>::</td>
<td>$\sum_{i=0}^{n} 1$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Generating Primes: 2

- **primesUpto**
- **invariant**
- **generateFrom**

composites
primes2Upto(n) =

\[
\begin{cases}
[] & \text{if } n < 2 \\
[(1, 2)] & \text{if } n = 2 \\
\text{primes2Upto}(n - 1) & \text{elseif } \text{even}(n) \\
\text{generate2From} & \text{otherwise} \\
\end{cases}
\]

\text{generate2From} = 

\{ 
[(1, 2)], 3, n, 2 \} 

where
generate2From\((P, m, n, k)\)

**bound function** \(n - m\)

**Invariant**

\[
2 < m \leq n \land \text{odd}(m)
\]

implies

\[
P = [(1, p_1), \ldots, (k - 1, p_{k-1})]
\]

and

\[
\forall q : p_{k-1} < q < m : \text{composite}(q)
\]
generate2From

generate2From\((P, m, n, k) =
\begin{cases}
P & \text{if } m > n \\generate2From\((P\text{@}[(k, m)], m + 2, n, k + 1) & \text{elseif} \\
\end{cases}
\text{pwr}t
\end{cases}
\text{else}
\begin{cases}
\end{cases}
\text{else}
\begin{cases}
\end{cases}
\text{where } \text{pwr}t = \text{prime2WR}T(m, P)
prime2WRT(m, P)

Bound function length(P)

Invariant If $P = [(i, p_i), \ldots (k - 1, p_{k-1})]$, for some $i \geq 1$ then

- $p_k \geq m > p_{k-1}$, and
- $m$ is prime with respect to $[(1, p_1), \ldots, (i - 1, p_{i-1})]$
- $m$ is a prime iff it is a prime with respect to $[(1, p_1), \ldots, (j, p_j)]$, where $p_j \leq \lfloor \sqrt{m} \rfloor < p_{j+1}$
prime\text{2W RT} \quad \text{prime\text{2W RT} (m, P) =}

\begin{cases}
\text{true} & \text{if } P = \text{nil} \\
\text{true} & \text{if } h > m \text{ div } h \\
\text{false} & \text{elseif } h | m \\
\text{prime\text{2W RT}} & \text{else}
\end{cases}

\text{where}

(i, h) = hd(P)

\text{for some } i > 0
# Complexity

<table>
<thead>
<tr>
<th>function</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>primes2Upto</code></td>
<td>1</td>
</tr>
<tr>
<td><code>generate2From</code></td>
<td>$n/2$</td>
</tr>
<tr>
<td><code>prime2WRT</code></td>
<td>$\sum_{m=3, \text{odd}(m)}^{n} \pi(\lfloor \sqrt{m} \rfloor)$</td>
</tr>
</tbody>
</table>
primes2: Diagnosis

*generate2From*

- Uses `@` to create an ascending sequence of primes
- For each new prime $p_k$ this operation takes time $O(k)$.
- Can tail recursion be used to reduce the complexity due to `@`?
- Can a more efficient algorithm using `::` instead of `@` be devised (as in the case of $AS3$)?
Compound Data: Summary

- Compound Data:
  - **Tuples**: Cartesian products of different types (ordered)
  - **Lists**: Sequences of the same type of element
  - **Records**: Unordered named aggregations of elements of different types.

- Constructors & Destructors
Records: Constructors

- A record is a set of values drawn from various types such that each component (called a field) has a unique name.

- Each record has a type defined by field names types of fieldnames. The order of presentation of the record fields does not affect its type in any way.
Records

Standard ML of New Jersey,
- val pinky =
{ name = "Pinky", age = 3,
fav_colour = "pink"};
- val pinky = {age=3,
fav_colour="pink",
name="Pinky"}
: {age:int,
fav_colour:string,
name:string
}
Records

- val billu = 
  { age = 1,
    name = "Billu",
    fav_colour = "blue"
  };
- val billu = 
  {age=1,fav_colour="blue",name="Billu"};
- val billu = 
  {age:int, fav_colour:string, name:string}
- pinky = billu;
val it = false : bool
Records: Destructors

```ocaml
#age billu;
val it = 1 : int
-
#fav_colour billu;
val it = "blue" : string
-
#name billu;
val it = "Billu" : string
```
Records: Equality

- val pinky2 =
  { name = "Pinky",
    fav_colour = "pink",
    age = 3
  };
- val pinky2 =
  {age=3,fav_colour="pink",name="Pinky"};
- pinky = pinky2;
val it = true : bool
Tuples & Records

- A $k$-tuple may be thought of as a record whose fields are numbered #1 to #k instead of having names.
- A record may be thought of as a generalization of tuples whose components are named rather than being numbered.
Back to Lists

- Every $L : \alpha List$ satisfies
  \[ L = [] \]

**XOR**

\[ L = hd(L) :: tl(L) \]

- Many functions on lists ($L$) are defined by induction on its length ($|L|$).

\[ f(L) = \begin{cases} 
  c & \text{if } L = [] \\
  g(h, T) & \text{if } L = h :: T 
\end{cases} \]
Lists: Correctness

Hence their properties \( P \) are proved by induction on the length of the list.

**Basis** \[ |L| = 0 \]. Prove \( P([]) \)

**Induction hypothesis** \((IH)\) Assume for some \( |T| = n > 0 \), \( P(T) \) holds.

**Induction Step** Prove \( P(h :: T) \) for \( L = h :: T \) with \( |L| = n + 1 \)
Lists: Case Analysis

inductive defns on lists

• Every list has exactly one of the following forms (patterns)
  - []
  - h::T

• ML provides convenient case analysis based on patterns.

\[
\text{fun } f \ [ ] = c \\
\quad | f \ (h::T) = g \ (h, \ T) \\
\quad ;
\]
Lists: Correctness by Cases

$P$ is proved by case analysis.

**Basis** Prove

$P([])$

**Induction hypothesis** ($IH$) Assume

$P(T)$

**Induction Step** Prove

$P(h :: T)$
List-functions: \( \text{length} \)

\[
\begin{align*}
\text{length} & \quad = 0 \\
\text{length} \; (h :: T) & \quad = 1 + (\text{length} \; T)
\end{align*}
\]
List Functions: \textit{search}

To determine whether $x$ occurs in a list $L$

\[
\begin{align*}
\text{search} (x, []) &= \text{false} \\
\text{search} (x, h :: T) &= \text{true if } x = h \\
\text{search} (x, h :: T) &= \text{search}(x, T) \text{ else}
\end{align*}
\]
List Functions: $search2$

Or even more conveniently

\[
\begin{align*}
    search2 \ (x, \ []) & = \text{false} \\
    search2 \ (x, \ h::T) & = (x = h) \ \text{or} \\
    search2 \ (x, \ T) &
\end{align*}
\]

Time Complexity??
List Functions: ordered

Definition 7 A list $L = [a_0, \ldots, a_{n-1}]$ is ordered by a relation $\leq$ if consecutive elements are related by $\leq$, i.e $a_i \leq a_{i+1}$, for $0 \leq i < n - 1$.

\[
\begin{align*}
\text{ordered } &[] \\
\text{ordered } &[h] \\
\text{ordered } (h_0 :: h_1 :: T) \quad \text{if } h_0 \leq h_1 \text{ and } \\
&\text{ordered}(h_1 :: T)
\end{align*}
\]

Time Complexity??
List Functions: \texttt{insert}

Given an ordered list $L : \alpha \ List$, insert an element $x : \alpha$ at an appropriate position

\[
\begin{align*}
\text{insert} \ (x, \ []) &= [x] \\
\text{insert} \ (x, \ h :: T) &= x :: (h :: T) \\
\text{if} \ x &\leq h \\
\text{insert} \ (x, \ h :: T) &= h :: (\text{insert} \ (x, \ T)) \\
\text{else}
\end{align*}
\]

Time Complexity??
List Functions: \textit{reverse}

Reverse the elements of a list 
$L = [a_0, \ldots, a_{n-1}]$ to obtain 
$M = [a_{n-1}, \ldots, a_0]$.

\[
\begin{align*}
\text{reverse} \; [] & = [] \\
\text{reverse} \; (h :: T) & = (\text{reverse} \; T)@[h]
\end{align*}
\]

Time Complexity?? $O(n^2)$
List Functions: \( reverse2 \)

\[
\begin{align*}
reverse \; [] &= [] \\
reverse \; (h :: T) &= \text{rev} \; ((h :: T), [])
\end{align*}
\]

where

\[
\begin{align*}
\text{rev} \; ([], N) &= N \\
\text{rev} \; (h :: T, N) &= \text{rev} \; (T, h :: N)
\end{align*}
\]

Correctness ?? Time Complexity?? \( O(n) \)
List Functions: $\text{merge}$

Merge two ordered lists $|L| = l$, $|M| = m$ to produce an ordered list $|N| = l + m$ containing exactly the elements of $L$ and $M$. That is if $L = [1, 3, 5, 9, 11]$ and $M = [0, 3, 4, 4, 10]$, then $\text{merge}(L, M) = N$, where $N = [0, 1, 3, 3, 4, 4, 5, 9, 10, 11]$
**List Functions:** $merge$

\[
merge([], M) = M \\
merge(L, []) = L \\
merge(L, M) = \begin{cases} 
  a :: (merge(S, M)) & \text{if } a \leq b \\
  b :: (merge(L, T)) & \text{else}
\end{cases}
\]
List Functions: \( \text{merge contd.} \)

where

\[
\begin{align*}
L &= a :: S \\
M &= b :: T
\end{align*}
\]
ML: \texttt{merge}

\begin{verbatim}
fun merge ([], M) = M
| merge (L, []) = L
| merge (L as a::S, M as b::T) =
  if a <= b
  then a::merge(S, M)
  else b::merge(L, T)
\end{verbatim}
Sorting by Insertion

Given a list of elements to reorder them (i.e. with the same number of occurrences of each element as in the original list) to produce a new ordered list.

Hence \( \text{sort}[10, 8, 3, 6, 9, 7, 4, 8, 1] = [1, 3, 4, 6, 7, 8, 8, 9, 10] \)

\[
\begin{align*}
\text{isort}[] &= [] \\
\text{isort}(h :: T) &= \text{insert}(h, (\text{isort}T))
\end{align*}
\]

Time Complexity??
Sorting by Merging

\[
\begin{align*}
msort [] & = [] \\
msort [a] & = [a] \\
msort L & = \text{merge} \left( (msort M), (msort N) \right)
\end{align*}
\]

where

\[(M, N) = \text{split} \ L\]
Sorting by Merging

where

\[
\begin{align*}
\text{split } &[] = ([], []) \\
\text{split } [a] &= ([a], []) \\
\text{split } (a :: b :: P) &= (a :: \text{Left}, b :: \text{Right})
\end{align*}
\]

where

\[(\text{Left}, \text{Right}) = \text{split } P\]

Time Complexity??
Functions as Data

- Every function is **unary**. A function of many arguments may be thought of as a function of a single argument i.e. a tuple of appropriate type.
- Every function is a **value** of an appropriate type.
- Hence functions are also **data**.
Higher Order Functions

Compound data may be constructed from functions as values using the constructors of the compound data structure.

*Functions may be defined with other functions and/or data as arguments to produce new values or new functions.*
Summary: Compound Data

- Records and tuples
- Lists
  - Correctness
  - Examples
List: Examples

- Length of a list
- Searching a list
- Checking whether a list is ordered
- Reversing a list
- Sorting of lists
Lists: Sorting

- Sorting by insertion
- Sorting by Divide-and-Conquer
Higher Order Functions

- Functions as data
- Higher order functions
An Exmple

List of functions

• $add1 \ x = x + 1$
• $add2 \ x = x + 2$
• $add3 \ x = x + 3$

Suppose we needed to define a long list of length $n$, where the $i$-th element is the function that adds $i + 1$ to the argument.
Currying

\[ \text{addc } y \ x = x + y \]

ML’s response:

\[
\text{val addc = fn : int } \rightarrow \text{ (int } \rightarrow \text{ int) }
\]

Contrast with ML’s response

- \text{op + ;}

\[
\text{val it = fn : int } * \text{ int } \rightarrow \text{ int}
\]

\text{addc} is the \text{curried} version of the bi-

tary operation \text{+}.
Currying: Contd

\[ f : (\alpha \ast \beta \ast \gamma) \to \delta \checkmark \]
\[ f_c : \alpha \to \beta \to \gamma \to \delta \checkmark \]
\[ f_c^1 : (\alpha \ast \beta) \to \gamma \to \delta \checkmark \]
\[ f_c^2 : \alpha \to (\beta \ast \gamma) \to \delta \checkmark \]
Generalization

Then

- \( addc_1 = (addc\ 1) : \text{int} \rightarrow \text{int} \)
- \( addc_2 = (addc\ 2) : \text{int} \rightarrow \text{int} \)
- \( addc_3 = (addc\ 3) : \text{int} \rightarrow \text{int} \)

and for any \( i \),

\[ (addc\ i) : \text{int} \rightarrow \text{int} \]

is the required function.
Generalization: 2

\[
\text{list_adds } n = \begin{cases} 
\emptyset & \text{if } n \leq 0 \\
(\text{list_adds}(n - 1)) \@ ([\text{addc } n]) & \text{else}
\end{cases}
\]

ML's response:

val list_adds = fn : int -> (int -> int) list
Applying a list

\[
\begin{align*}
&\text{applyl} \; \mathbb{L} \; x = \mathbb{L} \\
&\text{applyl} \; (h :: T) \; x = (h \; x) :: (\text{applyl} \; T \; x)
\end{align*}
\]

ML’s response:
val applyl = fn :
('a -> 'b) list ->
'a -> 'b list
Trying it out

\[ \text{interval } x \ n = \ \text{applyl } x \ (\text{list_adds } n) \]

ML’s response:
\[
\begin{align*}
\text{val interval} & = \ \text{fn} : \\
& \quad \text{int} \rightarrow \ \text{int} \rightarrow \ \text{int list} \\
\text{val it} & = [54, 55, 56, 57, 58] : \ \text{int list}
\end{align*}
\]
Associativity

• Application associates to the left.
  \[ f x y = ((f x) y) \]

• \( \rightarrow \) associates to the right.
  \[ \alpha \rightarrow \beta \rightarrow \gamma = \alpha \rightarrow (\beta \rightarrow \gamma) \]

If \( f : \alpha \rightarrow \beta \rightarrow \gamma \rightarrow \delta \)
then \( f a : \beta \rightarrow \gamma \rightarrow \delta \)
and \( f a b : \gamma \rightarrow \delta \)
and \( f a b c : \delta \)
Apply to a list

Apply a list Transpose of a matrix

\[
\text{map} \ f \ [\ ] \ = \ [\ ] \\
\text{map} \ f \ (h :: T) \ = \ (f \ h) :: (\text{map} \ f \ T)
\]

val it = fn : ('a -> 'b) -> \\
\quad 'a list -> 'b list \\
- map addc3 [4, 6, ~1, 0];
val it = [7,9,2,3] : int list \\
- map real [7,9,2,3];
val it = [7.0,9.0,2.0,3.0] \\
: real list
Sequences

Arithmetic sequences-1
Arithmetic sequences-2
Arithmetic sequences-3

$$AS4(a, d, n) =$$

$$\begin{cases} 
\emptyset & \text{if } n \leq 0 \\
 a :: (map (addc d) \ (AS4 (a, d, (n - 1)))) & \text{else}
\end{cases}$$
Further Generalization

Given

\[ f : \alpha \ast \alpha \to \alpha \]

Then

\[ \text{curry2 } f \ x \ y = f(x, y) \]

and

\[ (\text{curry2 } f) : \alpha \to (\alpha \to \alpha) \]

and for any \( d : \alpha \),

\[ ((\text{curry2 } f) \ d) : \alpha \to \alpha \]
Further Generalization

\[ seq(f, a, d, n) = \begin{cases} \emptyset & \text{if } n \leq 0 \\ a :: (map ((curry2 f) d) (seq (f, a, d, n - 1))) & \text{else} \end{cases} \]

is the sequence of length \( n \) generated with \((curry2 f) d\), starting from \( a \).
Sequences

**Arithmetic:** \( AS5(a, d, n) \) = \( \text{seq}(op+, a, d, n) \)

**Geometric:** \( GS1(a, r, n) \) = \( \text{seq}(op\ast, a, r, n) \)

**Harmonic:** \( HS1(a, d, n) \) = \( \text{map reci} (\text{AS5}(a, d, n)) \)

where \( \text{reci} x = 1.0/(\text{real} x) \) gives the reciprocal of a (non-zero) integer.
Efficient Generalization

Let’s not use `map` repeatedly.

\[
\text{seq2}(f, g, a, d, n) = \\
\begin{cases} 
\emptyset \text{ if } n \leq 0 \\
(f \ a) :: (\text{seq2}(f, g(a, d), d, n - 1)) \text{ else} 
\end{cases}
\]

is the sequence of length \(n\) generated with a unary \(f\), a binary \(g\) starting from \(f(a)\).
Sequences: 2

- \( AS_6(a, d, n) = seq2(id, op+, a, d, n) \)
- \( GS_2(a, r, n) = seq2(id, op*, a, r, n) \)
- \( HS_2(a, d, n) = seq2(reci, op+, a, d, n) \)

where \( id \ x = x \) is the identity function.
More Generalizations

Often interested in some particular measure related to a sequence, rather than in the sequence itself, e.g. summations of

- arithmetic, geometric, harmonic sequences
- $e^x$, trigonometric functions up to some $n$-th term
- (Truncated) Taylor and Maclaurin series
More Summations

Wasteful to \textcolor{red}{first} generate the sequence and \textcolor{red}{then} compute the measure

\[ \sum_{i=l}^{u} f(i) \]

where the range \([l, u]\) is defined by a unary \textit{succ} function

\[ sum(f, \text{succ}, l, u) = \]

\[
\begin{cases} 
0 & \text{if } [l, u] = \emptyset \\
 f(l) + sum(f, \text{succ}, \text{succ}(l), u) & \text{else}
\end{cases}
\]
Or Maybe . . . Products

Or may be interested in forming products of sequences.

\[
\prod_{i=l}^{u} f(i)
\]

\[
\text{prod}(f, \text{succ}, l, u) = \begin{cases} 
1 & \text{if } [l, u] = \emptyset \\
 f(l) \ast \text{prod}(f, \text{succ}, \text{succ}(l), u) & \text{else}
\end{cases}
\]
Or Some Other $\otimes$

Or some other binary operation $\otimes$ which has the following properties:

- $\otimes : (\alpha \ast \alpha) \rightarrow \alpha$ is closed
- $\otimes$ is associative i.e.
  \[ a \otimes (b \otimes c) = (a \otimes b) \otimes c \]
- $\otimes$ has an identity element $e$ i.e.
  \[ a \otimes e = a = e \otimes a \]

\[ \otimes \left( f(l) \right) \]
Then if \( f, \text{succ} : \alpha \rightarrow \alpha \)

\[
\text{ser}(\otimes, f, \text{succ}, l, u) =
\begin{cases} 
  e & \text{if } [l, u] = \emptyset \\
  f(l) \otimes \text{ser}(\otimes, f, \text{succ}, \text{succ}(l), u) & \text{else}
\end{cases}
\]
Examples of \( \otimes, e \)

- \( +, 0 \) on integers and reals
- concatenation and the empty string on strings
- `andalso`, `true` on booleans
- `orelse`, `false` on booleans
- \( +, 0 \) on vectors and matrices
- \( *, 1 \) on vectors and matrices
Transpose of a Matrix

Assume a 2-D $r \times c$ matrix is represented by a list of lists of elements. Then

$$\text{transpose } L = \begin{cases} \text{trans } L & \text{if } \text{is2DMatrix}(L) \\ \bot & \text{else} \end{cases}$$

where
Transpose: 0

\[
\begin{bmatrix}
11 & 12 & 13 \\
21 & 22 & 23 \\
31 & 32 & 33 \\
41 & 42 & 43 \\
\end{bmatrix}
\]
### Transpose: 10

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
</tr>
</tbody>
</table>
### Transpose: 01

\[
\begin{bmatrix}
  12 & 13 \\
  22 & 23 \\
  32 & 33 \\
  42 & 43 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
  11 & 21 & 31 & 41 \\
\end{bmatrix}
\]
Transpose: 20

\[
\begin{bmatrix}
12 & 13 \\
22 & 23 \\
32 & 33 \\
42 & 43
\end{bmatrix}
\quad \rightarrow \quad
\begin{bmatrix}
11 & 21 & 31 & 41
\end{bmatrix}
\]
### Transpose: 02

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>22</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>33</td>
<td>43</td>
</tr>
<tr>
<td>3</td>
<td>31</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>41</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The matrix is transposed as follows:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>22</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>33</td>
<td>43</td>
</tr>
<tr>
<td>3</td>
<td>31</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>41</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Transpose: 30

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>43</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{bmatrix}
11 & 21 & 31 & 41 \\
12 & 22 & 32 & 42 \\
\end{bmatrix}
\]
### Transpose: 03

```
[ 0 0 0 0 ]
[ 0 0 0 0 ]
[ 0 0 0 0 ]
[ 0 0 0 0 ]
```

```
[ 11 21 31 41 ]
[ 12 22 32 42 ]
[ 13 23 33 43 ]
```

```
[ ]
[ ]
[ ]
[ ]
```
\[
\begin{align*}
\text{trans} \\
\text{trans} \begin{bmatrix} \end{bmatrix} &= \begin{bmatrix} \end{bmatrix} \\
\text{trans} \begin{bmatrix} \end{bmatrix} \colon TL &= \begin{bmatrix} \end{bmatrix} \\
\text{trans} LL &= (\text{map} \text{ hd} LL) \colon \\
&\quad (\text{trans} (\text{map} \text{ tl} LL)) \\
\end{align*}
\]

and

\[
is2DMatrix = \#1(\text{dimensions} L)
\]

where
is2DMatrix

dimensions \[\cdot\] = (true, 0, 0)

dimensions \([H]\) =
(true, 1, h)

dimensions \((H :: TL)\) =
(b and (h = c), r + 1, c)

where dimensions \(TL = (b, r, c)\)
and \(h = \text{length } H\)
User Defined Types

Many languages allow user-defined data types.

- record types: Pinky and Billu
- Enumerations: aggregates of heterogeneous data.
- other structural constructions (if desperate!)
Enumeration Types

Many languages allow user-defined data types.

• record types: Pinky and Billu

• Enumerations: aggregates of heterogeneous data.
  – days of the week
  – colours
  – geometrical shapes

• other structural constructions (if desperate!)
User Defined Structural Types

Many languages allow user-defined data types.

- **record types**: Pinky and Billu
- **Enumerations**: aggregates of heterogeneous data.
- **other structural constructions** (if desperate!)
  - trees
  - graphs
  - symbolic expressions
Functions vs. data

• Inspired by the list constructors, nil and cons
• Grand Unification of functions and data
  – Functions as data
  – Data as functions
Data as 0-ary Functions

• Every data element may be regarded as a function with 0 arguments

  – **Caution:** A constant function
    \[ f(x) = 5, \text{ for all } x : \alpha \]

  where

  \[ f : \alpha \rightarrow \text{int} \]

  is not the same as a value

  \[ 5 : \text{int} \]

  . Their types are different.
# Data vs. Functions

<table>
<thead>
<tr>
<th>Facilities</th>
<th>Functions</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>primitive</td>
<td>operations</td>
<td>values</td>
</tr>
<tr>
<td>user-defined</td>
<td>functions</td>
<td>constructors</td>
</tr>
<tr>
<td>composition</td>
<td>application</td>
<td>alternative</td>
</tr>
<tr>
<td>recursion</td>
<td>recursion</td>
<td>recursion</td>
</tr>
</tbody>
</table>
# Data vs. Functions: Recursion

<table>
<thead>
<tr>
<th>Recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basis</td>
</tr>
<tr>
<td>naming</td>
</tr>
<tr>
<td>composition</td>
</tr>
<tr>
<td>induction</td>
</tr>
</tbody>
</table>
Lists

datatype 'a list =
  nil | cons of 'a * 'a list

Every $\alpha \text{list}$ is either
nil: (Basis, name)
| : or (alternative)
cons: constructed inductively from
an element of type 'a and another
list of type 'a list using the con-
structor cons
Constructors

- Inspired by the list constructors

  \( \text{nil} : \alpha \text{ list} \)

  \( \text{cons} : \alpha \times \alpha \text{ list} \rightarrow \alpha \text{ list} \)

- combine heterogeneous types: \( \alpha \) and \( \alpha \text{ list} \)

- allows recursive definition by a form of induction

  \textbf{Basis} : \textit{nil}

  \textbf{Induction} : \textit{cons}
Shapes

A non-recursive data type

datatype shape =

  CIRCLE of real

  | RECTANGLE of real * real

  | TRIANGLE of real * real * real
Shapes: Triangle Inequality

fun isTriangle
  (TRIANGLE (a, b, c)) =
  (a+b>c) andalso
  (b+c>a) andalso
  (c+a>b)
| isTriangle _ = false
Shapes: Area

exception notShape;

fun area (CIRCLE (r)) = 3.14159 * r * r
| area (RECTANGLE (l,b)) = l*b
| area (s as TRIANGLE (a, b, c)) =
Shapes: Area

if isTriangle (s) then
let val s = (a+b+c)/2.0
in Math.sqrt
    (s*(s-a)*(s-b)*(s-c))
end
else raise notShape;
ML: Try out

- use "shapes.sml"

[opening shapes.sml]

datatype shape

  = CIRCLE of real
  |
  RECTANGLE of real * real
  |
  TRIANGLE of real * real * real

val isTriangle =

  fn : shape -> bool

exception notShape

val area = fn : shape -> real
ML: Try out (contd.)

val it = () : unit
- area (TRIANGLE (2.0,1.0,3.0))

uncaught exception notShape
raised at: shapes.sml:22.17–22.25
- area
  (TRIANGLE (3.0, 4.0, 5.0));
val it = 6.0 : real

Back to User defined types
Enumeration Types

- Enumeration types are non-recursive datatypes with

- 0-ary constructors

```
datatype working = MON | TUE
                 | WED | THU | FRI;
datatype weekends = SAT | SUN
datatype weekdays = working
                  | weekends;
```

Back to User defined types
Recursive Data Types

• But the really interesting types are the recursive data types

  Back to Lists

• As with lists proofs of correctness on recursive data types depend on a case-analysis of the structure (basis and inductive constructors)

  Correctness on lists
Resistors: Datatype

datatype resist =
    RES of real |
    SER of resist * resist |
    PAR of resist * resist
Resistors: Equivalent

fun value (RES (r)) = r
| value (SER (R1, R2)) = value (R1) + value (R2)
| value (PAR (R1, R2)) =
  let val r1 = value (R1);
  val r2 = value (R2)
  in (r1*r2)/(r1+r2)
end;
val R = PAR(
    SER(
        PAR(
            RES (5.0),
            RES (4.0)
        ),
        SER(
            RES (5.0),
            RES (2.0)
        )
    ),
    RES (3.0)
);
Resistors: ML session
- use "resistors.sml";

[opening resistors.sml]
datatype resist = PAR of resist * resist
  | RES of real
  | SER of resist * resist

val value = fn : resist -> real
val R = PAR (SER (PAR #,SER #),RES 3.0)
val it = () : unit
- value R;
val it = 2.26363636364 : real

Resistance Expressions
User Defined Types

- Records
- Structural Types
  - Constructors
    * Non-recursive
    * Enumeration Types
  - Recursive datatypes
    * Resistance circuits
Resistors: Grouping

R1: 5.0, 4.0
R2: 5.0, 2.0
R3: 5.0
R4: 3.0

Diagram showing the grouping of resistors.
Resistors: In Pairs

val R1 = PAR (RES 5.0, RES 4.0) : resist
val R2 = SER (RES 5.0, RES 2.0) : resist
val R3 = SER (R1, R2);
val R4 = PAR (R3, RES(3.0));
Resistor: Values

- value R1;
  val it = 2.22222222222 : real
- value R2;
  val it = 7.0 : real
- value R3;
  val it = 9.22222222222 : real
- value R4;
  val it = 2.26363636364 : real
Resistance Expressions

A resistance expression

Circuit Diagram
Resistance Expressions

A resistance expression

Circuit Diagram
Arithmetic Expressions

ML arithmetic expressions:
\[ ((5 \times \sim 4) + \sim (5 - 2)) \text{ div } 3 \]
are represented as trees
Arithmetic Expressions: 0

```
5 ~ 5 - 2 4 * 5 + 2 div 3
```
Arithmetic Expressions: 1

```
\text{div}
\text{+}
\text{*}
\text{~}
\text{4}
\text{5}
\text{2}
\text{3}
```

The diagram represents the arithmetic expression: 

\( \text{div} (\text{+} (\text{*} (\text{~} 5), 4), 2) - 3 \)

This expression illustrates the use of various arithmetic operations, including division, addition, multiplication, and subtraction.
Arithmetic Expressions: 2

```
5 * -4 + 3 / 3
```
Arithmetic Expressions: 3

\[
\begin{align*}
5 \times (-4) & + 3 \div 3 \\
\end{align*}
\]
Arithmetic Expressions: 4

-20 + (-3) = 3
Arithmetic Expressions: 5

\[ \frac{-20}{3} - 3 \]
Arithmetic Expressions: 6

\[-23 \div 3\]
Arithmetic Expressions: 7

\[-23 \div 3\]
Arithmetic Expressions: 8
Binary Trees

datatype 'a bintree = Empty | Node of 'a * 'a bintree * 'a bintree
Arithmetic Expressions: 0

Arithmetic Expressions

\[
\begin{align*}
\text{div} & \\
+ & \\
\ast & \\
\sim & \\
5 & \\
\sim & \\
- & \\
3 & \\
4 & \\
5 & \\
2 & \\
\end{align*}
\]
Trees: Traversals

- preorder
- inorder
- postorder
Recursive Data Types: Correctness

Correctness on lists by cases

$P$ is proved by case analysis.
Data Types: Correctness

**Basis** Prove $P(c)$ for each non-recursive constructor $c$

**Induction hypothesis (IH)** Assume $P(T)$ for all elements of the data type less than a certain depth

**Induction Step** Prove $P(r(T_1, \ldots, T_n))$ for each recursive constructor $r$
Summary: Functional Model

- **Stateless** (as is most mathematics)
- Notion of value is paramount
  - Integers, reals, booleans, strings and characters are all values
  - Every function is also a value
  - Every complex piece of data is also a value
- No concept of storage (except for space complexity calculations)
CPU & Memory: Simplified

- CPU
- Memory
- Peripherals:
  - Printer
  - Disk
  - Keyboard
  - Screen
Resource Management

Operating System

CPU

Memory

Peripherals
- Printer
- Disk
- Keyboard
- Screen

Resource Management

CPU

Memory

Peripherals
- Printer
- Disk
- Keyboard
- Screen

Operating System
Shell: User Interface

Operating System

Shell

CPU

Memory

Peripherals

Printer

Disk

Keyboard

Screen

CPU

Memory

Peripherals

Printer

Disk

Keyboard

Screen
GUI: User Interface

Graphical User Interface (GUI)

Operating System

Shell

CPU

Memory

Peripherals

Printer

Disk

Keyboard

Screen

Operating System

Shell

Graphical User Interface (GUI)
Memory Model: Simplified

1. A sequence of storage cells
2. Each cell is a container of a single unit of information.
   • integer, real, boolean, character or string
3. Each cell has a unique name, called its address
4. The memory cell addresses range from 0 to (usually) $2^k - 1$ (for some $k$)
Memory

0 1 2 3

32K−1
The Imperative Model

- Memory or Storage made explicit
- Notion of state (of memory)
  - State is simply the value contained in each cell.
    - state : Addresses → Values
- State changes
State Changes: $\sigma$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;#a&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume all other cells are filled with null.
State

The state $\sigma$

- $\sigma(12) = 4 : \text{int}$
- $\sigma(20) = \text{null}$
- $\sigma(43) = \text{true : bool}$
- $\sigma(66) = "\#a" : \text{char}$
State Changes

- A state change takes place when the value in some cell changes.
- The contents of only one cell may be changed at a time.
### State Changes: $\sigma$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>true</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;#a&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume all other cells are filled with `null`
State Changes: $\sigma_1$

Assume all other cells are filled with `null`
State Changes: $\sigma_2$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Assume all other cells are filled with null

- First
- Prev
- Next
- Last
- Go Back
- Full Screen
- Close
- Quit
Imperative Languages

- How is the memory accessed?
  - Through system calls to the OS.

- How are memory cells identified?
  - Use Imperative variables.
  - Each such variable is a name mapped to an address.

- How are state changes accomplished?
  - By the assignment command.
# Imperative vs Functional Variables

<table>
<thead>
<tr>
<th>Functional</th>
<th>Imperative</th>
</tr>
</thead>
<tbody>
<tr>
<td>name of a value</td>
<td>name of an address</td>
</tr>
<tr>
<td>constant</td>
<td>could change with time</td>
</tr>
</tbody>
</table>

The value contained in an imperative variable $x$ is denoted $!x$. 
Assignment Commands

Let \( x \) and \( y \) be imperative variables. Consider the following commands. Assuming \( x = 1 \) and \( y = 2 \).
Assignment Commands

Store the value \(5\) in \(x\).

\[
x := 5
\]
Assignment Commands

Copy the value contained in $y$ into $x$.

$x := y$

$$x \ 2 \ y \ 2$$
Increment the value contained in \( x \) by 1.

\[
x := x + 1
\]
Assignment Commands

Store the product of the values in $x$ and $y$ in $y$.

\[ y := x \times y \]
Assignment
Commands: Swap

Swap the values in $x$ and $y$.

Swapping values implies trying to make two state changes simultaneously!

Requires a new memory cell $t$ to temporarily store one of the values.
Swap

How does one get a new memory cell?

```
val t = ref 0
```

Then the rest is easy

```
val t = ref 0;

let

  t := !x;
  x := !y;
  y := !t;
```
Swap

Could be made simpler!

```ocaml
val t = ref (!x);
x := !y;
y := !t;
```
Swap

Could use a temporary functional variable $t$ instead of an imperative variable

```plaintext
val t = !x;
x := !y;
y := t;
```
Imperative vs Functional

- Functional Model
- Memory/Store Model
- Imperative Model
- State Changes
- Accessing the store
Features of the Store

Memory is treated as a datatype with constructors

Allocation \( \text{ref} : \alpha \rightarrow \alpha \text{ ref} \)

Dereferencing \( ! : \alpha \text{ ref} \rightarrow \alpha \)

Updation \( \text{:=:} : \alpha \text{ ref} \ast \alpha \rightarrow \text{unit} \)

Deallocation of memory is automatic!
References: Experiments

- val a = ref 0;
val a = ref 0 : int ref
References: Experiments

\[-\text{val } b = \text{ref } 0;\]
\[\text{val } b = \text{ref } 0 : \text{int ref}\]
References:
Experiments

- \( a = b; \)
val it = false : bool
- \( !a = !b; \)
val it = true : bool
Aliases

- val x = ref 0;
val x = ref 0 : int ref
References: Experiments

- val y = x;
val y = ref 0 : int ref
References: Aliases

```ml
- x := !x + 1;
val it = () : unit
```
References: Experiments

- !y;
val it = 1 : int
- x = y;
val it = true : bool
After Garbage Collection

GC #0.0.0.0.2.45: (0 ms)
Side Effects

- **Assignment** does not produce a value
- It produces only a state change (side effect)
- But side-effects are compatible with functional programming since it is provided as a new data type with constructors and destructors.
Imperative ML

- Does not provide direct access to memory addresses
- Does not allow for uninitialized imperative variables
- Provides a type with every memory location
- Manages the memory completely automatically
Imperative ML

- Does not provide direct access to memory addresses
  - Prevents the use of memory addresses as integers that can be manipulated by the user program
- Does not allow for uninitialized imperative variables
- Provides a type with every memory location
- Manages the memory completely automatically
Imperative ML

- Does not provide direct access to memory addresses
- Does not allow for uninitialized imperative variables
  - Most imperative languages keep declarations separate from initializations
- Provides a type with every memory cell
- Manages the memory completely automatically
Imperative ML

- Does not provide direct access to memory addresses
- Does not allow for uninitialized imperative variables
  - A frequent source of surprising results in most imperative language programs
- Provides a type with every memory cell
- Manages the memory completely automatically
Nasty Surprises

Separation of declaration from initialization

- Uninitialized variables
- Execution time errors if not detected by compiler, since every memory location contains some data
- Might use a value stored previously in that location by some imperative variable that no longer exists.
- Errors due to type violations.
Imperative ML

- Does not provide direct access to memory addresses
- Does not allow for uninitialized imperative variables
- Provides a type with every memory cell
- **Manages** the memory completely automatically and securely.
Imperative ML

- Does not provide direct access to memory addresses
- Does not allow for uninitialized imperative variables
- Provides a type with every memory cell
- Manages the memory completely automatically and securely
  - Memory has to be managed by the user program in most languages
  - Prone to various errors
Common Errors

- Memory access errors due to integer arithmetic, especially in large structures (arrays)
- Dangling references on deallocation of aliased memory
Aliasing & References

Before deallocation:

\[\begin{array}{ccc}
0 & 0 & 1 \\
a & b & x & y
\end{array}\]
Dangling References

Deallocate $x$ through a system call

$y$ is left dangling!
val z = ref 12;

By sheer coincidence \( y = 12 \)
Imperative Commands: Basic

A Command is an ML expression that creates a side effect and returns an empty tuple $(()) : \text{unit}$. 

Assignment

print
Imperative Commands: Compound

Any complex ML expression or function definition whose type is of the form $\alpha \rightarrow \text{unit}$ is a compound command.

- **Predefined** ML compound commands
- Could be user-defined. After all, *everything is a value!*
Predefined Compound Commands

**branching** if e then c₁ else c₀.

**cases** case e of p₁ ⇒ c₁ | · · · | pₙ ⇒ cₙ

**Sequencing** (c₁; c₂; · · ·; cₙ). Sequencing is associative

**looping** while e do c₁ is defined recursively as

if e then (c₁; while e do c₁) else ()
Why Imperative

• Historical reasons: Early machine instruction set.

• Programming evolved from the machine architecture.

• Legacy software:
  – numerical packages
  – operating systems

• Are there any real benefits of imperative programming?
Arrays

An array of length \( n \) is a contiguous sequence of \( n \) memory cells

\[ C_0, C_1, \ldots, C_{n-1} \]
Indexing Arrays

For any array

• $i, 0 \leq i < n$ is the index of cell $C_i$.

• $C_i$ is at a distance of $i$ cells away from $C_0$. 
Indexing Arrays

\[ C_0, C_i, C_{n-1} \]
Indexing Arrays

- The **start** address of the array and the **address** of $C_0$ are the same (say $a_0$)

- The address $a_i$ of cell $C_i$ is $a_i = a_0 + i$
Physical Addressing

If each element occupies \( s \) physical memory locations, then

\[
a_i = a_0 + i \times s
\]
Arrays

A 2-dimensional array of

- $r$ rows numbered 0 to $r - 1$
- each row containing $c$ elements numbered 0 to $c - 1$

is also a contiguous sequence of $rc$ memory cells

$$C_{0,0}, C_{0,1}, \ldots, C_{0,c-1}, C_{1,0}, \ldots, C_{r-1,c-1}$$
2D Arrays

A 2 dimensional-array is represented as an array of length $r \times c$, where

- $a_{00}$ is the start address of the array, and
- the address of the $(i, j)$-th cell is given by

$$a_{ij} = a_{00} + (c \times i + j)$$

- the physical address of the $(i, j)$-th cell is given by

$$a_{ij} = a_{00} + (c \times i + j) \times s$$
2D Arrays: Indexing

• The index \((i, j)\) of a 2D array may be thought of as being similar to a 2-digit number in base \(c\).

• The successor of index \((i, j)\) is the successor of a number in base \(c\) i.e.

\[
\text{succ}(i, j) = \begin{cases} 
(i + 1, 0) & \text{if } j = n - 1 \\
(i, j + 1) & \text{else}
\end{cases}
\]
Ordering of indices

There is a natural “<” ordering on indices given by

\[(i, j) < (k, l) \iff \begin{cases} 
  i < k \\
  i = k \text{ and } j < l
\end{cases}\]
## Arrays vs. Lists

<table>
<thead>
<tr>
<th>Lists</th>
<th>Arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unbounded lengths</td>
<td>Fixed length</td>
</tr>
<tr>
<td>Insertions possible</td>
<td>Very complex</td>
</tr>
<tr>
<td>Indirect access</td>
<td>Direct access</td>
</tr>
</tbody>
</table>

- Lists allow unbounded lengths and insertions, but access is indirect.
- Arrays have a fixed length and direct access, but inserting elements is very complex.

---

**Home Page**

---

**Title Page**

---

**Go Back**

---

**Full Screen**

---

**Close**

---

**Quit**
Arrays: Physical

\[ a_0, a_{n-1}, a_i \]
Lists: Physical
Logical Arguments

Examples.

- Saintly and Rich
- About cats
- About God
- Russell’s argument
Saintly and Rich

hy1 The landed are rich.

hy2 One cannot be both saintly and rich.

conc The landed are not saintly
About Cats

**hy1** *Tame cats are non-violent and vegetarian.*

**hy2** *Non-violent cats would not kill mice.*

**hy3** *Vegetarian cats are bottle-fed.*

**hy4** *Cats eat meat.*

**conc** *Cats are not tame.*
About God

hy1 God is omniscient and omnipotent.

hy2 An omniscient being would know there is evil.

hy3 An omnipotent being would prevent evil.

hy4 There is evil.

conc There is no God
Russell's Argument

**hy1** If we can directly know that God exists, then we can know God exists by experience.

**hy2** If we can indirectly know that God exists, then we can know God exists by logical inference from experience.

**hy3** If we can know that God exists, then we can directly know that God exists, or we can indirectly know that God exists.
Russell’s Argument

hy4 If we cannot know God empirically, then we cannot know God by experience and we cannot know God by logical inference from experience.

hy5 If we can know God empirically, then “God exists” is a scientific hypothesis and is empirically justifiable.

hy6 “God exists” is not empirically justifiable.

conc We cannot know that God exists.
Russell’s Argument

**hy1** If we can directly know that God exists, then we can know God exists by experience.

**hy2** If we can indirectly know that God exists, then we can know God exists by logical inference from experience.

**hy3** If we can know that God exists, then (we can directly know that God exists, or we can indirectly know that God exists).
Russell’s Argument

**hy4** If we cannot know God empirically, then (we cannot know God by experience and we cannot know God by logical inference from experience.)

**hy5** If we can know God empirically, then (“God exists” is a scientific hypothesis and is empirically justifiable.)

**hy6** “God exists” is not empirically justifiable.

**conc** We cannot know that God exists.
Propositions

A **proposition** is a sentence to which a **truth value** may be assigned.

In any real or imaginary world of facts a proposition has a truth value, **true** or **false**.

An **atom** is a simple proposition that has no propositions as components.
## Compound Propositions

Compound propositions may be formed from **atoms** by using the following operators/constructors.

<table>
<thead>
<tr>
<th>operator</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>¬</td>
</tr>
<tr>
<td>and</td>
<td>∧</td>
</tr>
<tr>
<td>or</td>
<td>∨</td>
</tr>
<tr>
<td>if...then...</td>
<td>⇒</td>
</tr>
<tr>
<td>equivalent</td>
<td>⇔</td>
</tr>
</tbody>
</table>
Valuations

Given truth values to individual atoms the truth values of compound propositions are evaluated as follows:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$\neg p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
# Valuations

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$p \land q$</th>
<th>$p \lor q$</th>
<th>$p \Rightarrow q$</th>
<th>$p \iff q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
Tautology

A (compound) proposition is a tautology if it is true regardless of what truth values are assigned to its atoms.

Examples.

• $p \lor \neg p$
• $(p \land q) \Rightarrow p$
• $(p \land \neg p) \Rightarrow q$
Properties

• Every proposition may be expressed in a logically equivalent form using only the operators \( \neg \), \( \land \) and \( \lor \)

\[(p \iff q) = (p \Rightarrow q) \land (q \Rightarrow p)\]

\[(p \Rightarrow q) = (\neg p \lor q)\]

• De Morgan’s laws may be applied to push \( \neg \) inward

\[\neg (p \land q) = \neg p \lor \neg q\]

\[\neg (p \lor q) = \neg p \land \neg q\]
Negation Normal Form

- Double negations may be removed since
  \[ \neg \neg p = p \]

- Every proposition may be expressed in a form containing only \( \land \) and \( \lor \) with \( \neg \) appearing only in front of atoms.
Literals & Clauses

- A **literal** is either an **atom** or ¬ applied to an **atom**
- ∨ is commutative and associative
- A **clause** is of the form ∨ \( j=1 \)^m l_j, where each \( l_j \) is a **literal**.
Conjunctive Normal Form

- \( \lor \) may be distributed over \( \land \)
  \[ p \lor (q \land r) = (p \lor q) \land (p \lor r) \]

- \( \land \) is commutative and associative.

- Every proposition may be expressed in the form \( \land_{i=1}^{n} q_i \), where each \( q_i \) is a clause.
Validity

- A logical argument consists of a number of hypotheses and a single conclusion \([h_1, \ldots, h_n] | c\).
- A logical argument is valid if the conclusion logically follows from the hypotheses.
Logical Validity

The conclusion logically follows from the given hypotheses if for any truth assignment to the atoms, either some hypothesis $h_i$ is false or whenever every one of the hypotheses is true the conclusion is also true.
Validity & Tautologies

• A **tautology** is a valid argument in which there is a **conclusion without any hypothesis**.

• A **logical argument** $[h_1, \ldots, h_n]|c$, is **valid if and only if**

  $$(h_1 \land \ldots \land h_n) \Rightarrow c$$

  is a tautology
Problem

Given an argument $[h_1, \ldots, h_n] \models c$,

- determine whether $(h_1 \land \ldots \land h_n) \Rightarrow c$ is a tautology, and

- If it is not a tautology, to determine what truth assignments to the atoms make it false.
Tautology Checking

A proposition in CNF \((\bigwedge_{i=1}^{n} p_i)\)

- is a tautology if and only if every proposition \(p_i\), \(1 \leq i \leq m\), is a tautology.

- otherwise at least one clause \(p_i\) must be false

- Clause \(p_i = \bigvee_{j=1}^{m} l_{ij}\) is false if and only if every literal \(l_{ij}\), \(1 \leq j \leq m\) is false
Falsifying

For a proposition in CNF ($\bigwedge_{i=1}^{n} p_i$) that is not a tautology

- A clause $p_i = \bigvee_{j=1}^{m} l_{ij}$ is false
- A truth assignment that falsifies the argument
  - sets the atoms that occur negatively in $p_i$ to true,
  - sets every other atom to false
Tautology Checking

- Logical arguments
- Propositional forms
- Propositions
- Compound Propositions
- Truth table
- Tautologies
Normal Forms

• Properties
• Negation Normal Form
• Conjunctive Normal Forms
• Valid Propositional Arguments as tautologies
• The problem
Top-down Development

- Transform the argument into a single proposition.
- Transform the single proposition into one in CNF

Check whether every clause is a tautology
If any clause is not a tautology, find the truth assignment(s) that make it false
The Signature

signature PropLogic =

sig
datatype Prop = ??
type Argument =
    Prop list * Prop
val falsifyArg : Argument -> Prop list list
val Valid :
    Argument -> bool * Prop list list

...
The Core subproblem

- Representing propositions
- Transformation of propositions into CNF
  - Transform into Negation Normal Form (NNF)
  - Transform NNF into Conjunctive Normal Form (CNF)
The datatype

datatype Prop =
  ATOM of string
  NOT of Prop
  AND of Prop * Prop
  OR of Prop * Prop
  IMP of Prop * Prop
  EQL of Prop * Prop
Convert to CNF

Convert a given proposition into CNF

fun cnf (P) =
    conj_of_disj (nnf (rewrite (P)));

where

- \texttt{rewrite} eliminates $\iff$ and $\Rightarrow$
- \texttt{nnf} converts into NNF
- \texttt{conj\_of\_disj} converts into CNF
Rewrite into NNF

- Eliminate $\iff$ and then $\implies$
- Push $\neg$ inward using De Morgan’s laws and eliminate double negations.
and ⇒ Elimination

fun rewrite (ATOM a) = ATOM a
| rewrite (IMP (P, Q)) = OR (NOT (rewrite(P)), rewrite(Q))
| rewrite (EQL (P, Q)) = rewrite (AND (IMP(P, Q), IMP(Q, P)))
| ...

Proposition made up of only ¬, ∧ and ∨.
Push inward

fun nnf (ATOM a) = 
ATOM a

| nnf (NOT (ATOM a)) = 
NOT (ATOM a)

| nnf (NOT (NOT (P))) = 
nnf (P)
Push \( \rightarrow \) inward

\[
\begin{align*}
nnf (\neg (\land (P, Q))) & = nnf (\lor (\neg (P), \\
& \quad \neg (Q))) \\
nnf (\neg (\lor (P, Q))) & = nnf (\land (\neg (P), \\
& \quad \neg (Q))) \\
& \cdots
\end{align*}
\]

Proposition made up of only \( \land \) and \( \lor \) applied to positive or negative literals.
fun conj_of_disj (AND (P, Q)) = 
  AND (conj_of_disj (P),
       conj_of_disj (Q))
| conj_of_disj (OR (P, Q)) =
  distOR (conj_of_disj (P),
         conj_of_disj (Q))
| conj_of_disj (P) = P

where distOR is
Push \( \lor \) inward

Use distributivity of \( \lor \) over \( \land \)

\[
\text{fun distOR} \ (P, \ AND \ (Q, \ R)) = \\
\ AND \ (\text{distOR} \ (P, \ Q), \\
\ AND \ (\text{distOR} \ (P, \ R))
\]

\[
\text{distOR} \ (\AND \ (Q, \ R), \ P) = \\
\ AND \ (\text{distOR} \ (Q, \ P), \\
\ AND \ (\text{distOR} \ (R, \ P))
\]

\[
\text{distOR} \ (P, \ Q) = \ OR \ (P, \ Q)
\]
Tautology & Falsification

Falsifying a proposition

• A proposition \( Q \) in CNF, not a tautology if and only if at least one of the clauses can be made false, by a suitable truth assignment.

• The list of atoms which are set true to falsify a clause is called a falsifier.

• A proposition is a tautology if and only if there is no falsifier!