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.

March 19, 2007
# Contents

1 Computing: The Functional Way
   1.1 Introduction to Computing .................................................. 3
   1.2 Our Computing Tool .............................................................. 26
   1.3 Primitives: Integer & Real ..................................................... 53
   1.4 Example: Fibonacci ............................................................... 80
   1.5 Primitives: Booleans .............................................................. 103

2 Algorithms: Design & Refinement ............................................ 114
   2.1 Technical Completeness & Algorithms ..................................... 114
   2.2 Algorithm Refinement ............................................................ 148
   2.3 Variations: Algorithms & Code .............................................. 180
   2.4 Names, Scopes & Recursion .................................................... 208

3 Introducing Reals ................................................................. 242
   3.1 Floating Point ........................................................................ 242
   3.2 Root Finding, Composition and Recursion .................................. 264

4 Correctness, Termination & Complexity .................................... 294
   4.1 Termination and Space Complexity .......................................... 294
   4.2 Efficiency Measures and Speed-ups ........................................ 333
   4.3 Invariance & Correctness ......................................................... 364

5 Compound Data ......................................................................... 390
   5.1 Tuples, Lists & the Generation of Primes .................................. 390
   5.2 Compound Data & Lists ........................................................... 429
   5.3 Compound Data & List Algorithms .......................................... 457

6 Higher Order Functions & Structured Data ................................. 486
   6.1 Higher Order Functions ........................................................... 486
   6.2 Structured Data ...................................................................... 513
6.3 User Defined Structured Data Types .................................................. 547

7 Imperative Programming: An Introduction .............................................. 570
   7.1 Introducing a Memory Model .......................................................... 570
   7.2 Imperative Programming: ............................................................... 599
   7.3 Arrays ......................................................................................... 626

8 A large Example: Tautology Checking ..................................................... 640
   8.1 Large Example: Tautology Checking ............................................... 640
   8.2 Tautology Checking Contd. .............................................................. 665

9 Lecture-wise Index to Slides .................................................................... 680

1.1. Introduction to Computing

1. Introduction
2. Computing tools
3. Ruler and Compass
4. Computing and Computers
5. Primitives
6. Algorithm
7. Problem: Doubling a Square
8. Solution: Doubling a Square
9. Execution: Step 1
10. Execution: Step 2
11. Doubling a Square: Justified
12. Refinement: Square Construction
13. Refinement 2: Perpendicular at a point
14. Solution: Perpendicular at a point
15. Perpendicular at a point: Justification

Next: Our Computing Tool
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.
Problem: Doubling a Square

Given a square, construct another square of twice the area of the original square.
Solution: Doubling a Square

Assume given a square $\square ABCD$ of side $a > 0$.

1. Draw the diagonal $AC$.

2. Complete the square $\square ACEF$ on side $AC$. 
Execution: Step 1

Draw the diagonal $\overline{AC}$.
Execution: Step 2

Complete the square $\Box ACEF$ on side $\overline{AC}$. 

![Square and triangles diagram]
Doubling a Square: Justified

Assume given a square $\square ABCD$ of side $a > 0$.

1. Draw the diagonal $\overline{AC}$. $AC = \sqrt{2}a$

2. Complete the square $\square ACEF$ on side $\overline{AC}$. Area of $\square ACEF = 2a^2$. 
Refinement: Square

Given a line segment of length \( b > 0 \) construct a square of side \( b \).

Assume given a line segment \( PQ \) of length \( b \).

1. Construct two lines \( l_1 \) and \( l_2 \) perpendicular to \( PQ \) passing through \( P \) and \( Q \) respectively.

2. On the same side of \( PQ \) mark points \( R \) on \( l_1 \) and \( S \) on \( l_2 \) such that \( PR = PQ = QS \).

3. Draw \( RS \). \( \square PQSR \) is a square of side \( b \).
Square on Segment: 0

Assume given a line segment $PQ$ of length $b$. 

\[ P \quad \underline{\hphantom{aaa}} \quad Q \]
Square on Segment: 1

Construct two lines $l_1$ and $l_2$ perpendicular to $PQ$ passing through $P$ and $Q$ respectively.
Square on Segment: 2

On the same side of $\overline{PQ}$ mark points $R$ on $l_1$ and $S$ on $l_2$ such that $PR = PQ = QS$. 

![Diagram of square on segment with points R, S, and Q on lines l1 and l2](image)
Square on Segment: 3

Draw $\overline{RS}$. $\square PQSR$ is a square of side $b$.

Square Construction algorithm
Perpendicular at a point

Given a line, draw a perpendicular to it passing through a given point on it. Assume given a line $l$ containing a point $X$. 
Solution: Perpendicular at a point

1. Choose a length $c > 0$. With $X$ as centre mark off points $Y$ and $Z$ on $l$ on either side of $X$, such that $YX = c = XZ$. $YZ = 2c$.

2. Draw Circles $C_1(Y, 2c)$ and $C_2(Z, 2c)$ respectively.

3. Join the points of intersection of the two circles.
Perpendicular at a Point: 1

Choose a length \( c > 0 \). With \( X \) as centre mark off points \( Y \) and \( Z \) on \( l \) on either side of \( X \), such that \( YX = c = XZ \). \( YZ = 2c \).
Perpendicular at a Point: 2

Draw Circles $C_1(Y, 2c)$ and $C_2(Z, 2c)$ respectively.
Perpendicular at a Point: 3

Choose a length $c > 0$. With $X$ as centre mark off points $Y$ and $Z$ on $l$ on either side of $X$, such that $YX = c = XZ$. $YZ = 2c$. 

![Diagram of perpendicular at a point]
Perpendicular at a point: Justification

1. The two circles intersect at points $U$ and $V$ on either side of line $l$.

2. $\overleftrightarrow{UV}$ is a perpendicular bisector of $YZ$.

3. Since $YX = c = XZ$ and $YZ = 2c$, $\overleftrightarrow{UV}$ is perpendicular to $l$ and passes through $X$.

Back to square 1
1.2. Our Computing Tool

1. The Digital Computer: Our Computing Tool
2. Algorithms
3. Programming Language
4. Programs and Languages
5. Programs
6. Programming
7. Computing Models
8. Primitives
9. Primitive expressions
10. Methods of combination
11. Methods of abstraction
12. The Functional Model
13. Mathematical Notation 1: Factorial
14. Mathematical Notation 2: Factorial
15. Mathematical Notation 3: Factorial
16. A Functional Program: Factorial

Previous: Introduction to Computing
17. A Computation: Factorial
18. A Computation: Factorial
19. A Computation: Factorial
20. A Computation: Factorial
22. A Computation: Factorial
23. A Computation: Factorial
24. Standard ML
25. SML: Important Features

Next: Primitives: Integer & Real
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, subtraction, 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.
1.3. Primitives: Integer & Real

1. Algorithms & Programs
2. SML: Primitive Integer Operations 1
3. SML: Primitive Integer Operations 1
4. SML: Primitive Integer Operations 1
5. SML: Primitive Integer Operations 1
6. SML: Primitive Integer Operations 1
7. SML: Primitive Integer Operations 1
8. SML: Primitive Integer Operations 2
9. SML: Primitive Integer Operations 2
10. SML: Primitive Integer Operations 2
11. SML: Primitive Integer Operations 2
12. SML: Primitive Integer Operations 2
13. SML: Primitive Integer Operations 3
14. SML: Primitive Integer Operations 3
15. SML: Primitive Integer Operations 3
16. SML: Primitive Integer Operations 3
17. SML: Primitive Integer Operations 3
18. Quotient & Remainder
19. SML: Primitive Real Operations 1
20. SML: Primitive Real Operations 1
21. SML: Primitive Real Operations 1
22. SML: Primitive Real Operations 2
23. SML: Primitive Real Operations 3
24. SML: Primitive Real Operations 4
25. SML: Precision
26. Fibonacci Numbers
27. Euclidean Algorithm

Next: Technical Completeness & Algorithms
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: 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
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
SML: Primitive Integer Operations 2

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

- \( x \times y; \)
- \( \text{val it = 30 : int} \)
- \( \text{val a = 25;} \)
- \( \text{val a = 25 : int} \)
- \( \text{val b = 7;} \)
- \( \text{val b = 7 : int} \)
- \( \text{val q = a \ div \ b;} \)
- \( \text{val q = 3 : int} \)
- \( \text{val r = a \ mod \ b;} \)
- \( \text{GC \ #0.0.0.0.2.45: (0 ms)} \)
- \( \text{val r = 4 : int} \)
- 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
- 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
val q = a div b;
val q = 3 : int
val r = a mod b;
val r = 4 : int
val it = true : bool
val c = ~7:
val c = ~7 : int
val q1 = a div c;
val q1 = ~4 : int
val r = a mod b;
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

1. $a = b \times q + r$

2. \[
\begin{cases}
0 \leq r < b & \text{when } b > 0 \\
b < r \leq 0 & \text{when } b < 0
\end{cases}
\]

So $0 \leq |r| < |b|$ always.
sml

Standard ML of New Jersey,
- val real_a = real a;
val real_a = 25.0 : real
-
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 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
val it = 3.57142857143 : real
val 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)
SML: Primitive Real Operations 3

- val c = a/b;
val c = 3.57142857143 : real
- trunc(c);
val it = 3 : int
- trunc (c + 0.5);
val it = 4 : int
-
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 = 299999999996.9 : real
pi + d;
val it = 300000000003.1 : real
SML: Precision

- \( \pi + d \times 10.0; \)
val it = 300000000003.0 : 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 = 2999999999997.0 : real

-
1.4. Example: Fibonacci

1. Fibonacci Numbers: 1
2. Fibonacci Numbers: 2
3. Fibonacci Numbers: 3
4. Fibonacci Numbers: 4
5. Fibonacci Numbers: 5
6. Is $F_a(n, 1, 1) = F(n)$?
7. Trial & Error
8. Generalization
9. Proof
10. Another Generalization
11. Try Proving it!
12. Another Generalization
13. Try Proving it!
14. Complexity
15. Complexity
16. Time Complexity: $R$
17. Time Complexity
18. Time Complexity: $R$
19. Bound on $\mathcal{R}$

20. Other Bounds: $C_F$

21. Other Bounds: $A_F$
Fibonacci Numbers: 1

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

fun fib (n) = 
  if (n = 0) orelse (n = 1) then 1
  else fib (n-1) + fib (n-2);
Fibonacci Numbers: 2

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

Alternatively,

\[
F(n) = \begin{cases} 
1 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
F(n - 1) + F(n - 2) & \text{if } n > 1
\end{cases}
\]
Fibonacci Numbers: 3

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

Alternatively,

```haskell
fun fib (n) = 
  if (n = 0) then 1
  else if (n = 1) then 1
  else fib (n-1) + fib (n-2);
```
Fibonacci Numbers: 4

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

Alternatively,

\[
F(n) = Fa(n, 1, 1)
\]

where

\[
Fa(n, a, b) = \begin{cases} 
    a & \text{if } n = 0 \\
    b & \text{if } n = 1 \\
    Fa(n - 1, b, a + b) & \text{if } n > 1
\end{cases}
\]
Fibonacci Numbers: 5

fun fib_a (n, a, b) = 
  if (n = 0) then a 
  else if (n = 1) then b 
  else fib_a (n, b, a+b);

fun fib (n) = fib_a (n, 1, 1);
Is $F_a(n, 1, 1) = F(n)$?

Intuition $F_a$ is a generalization of $F$.

Question 1 What does it actually generalize to?

Question 2 Does the generalization have a non-recursive form?

Trial & Error Can we use trial and error to get a non-recursive form?
Trial & Error

\[ Fa(2, a, b) = a + b \]
\[ Fa(3, a, b) = Fa(2, b, a + b) = a + 2b \]
\[ Fa(4, a, b) = Fa(3, b, a + b) = Fa(2, a + b, a + 2b) = 2a + 3b \]
\[ Fa(5, a, b) = Fa(2, a + 2b, 2a + 3b) = 3a + 5b \]
Generalization

- \( F_a(0, a, b) = a \)
- \( F_a(1, a, b) = b \)
- \( F_a(n, a, b) = aF(n - 2) + bF(n - 1) \)

When \( a = 1 \) and \( b = 1 \), for all \( n \geq 0 \),
  \( F_a(n, a, b) = F(n) \)

**Theorem 1** For all integers \( a, b \) and \( n > 1 \),

\[
F_a(n, a, b) = aF(n - 2) + bF(n - 1)
\]
Proof by Induction on $n > 1$

Proof:

**Basis** For $n = 2$, $F_a(2, a, b) = a + b = aF(0) + bF(1)$

**Induction hypothesis (IH)** Assume

$F_a(k, a, b) = aF(k - 2) + bF(k - 1)$, for some $k > 1$ and all integers $a, b$

**Induction Step**

$F_a(k + 1, a, b)$

$= F_a(k, b, a + b)$ Definition of $F_a$

$= bF(k - 2) + (a + b)F(k - 1)$ IH

$= aF(k - 1) + b(F(k - 2) + F(k - 1))$

$= aF(k - 1) + bF(k)$ Definition of $F$
Another Generalization

Try to prove a different and more direct theorem.

**Theorem 2** For all integers $n > 1$,

$$F_a(n, 1, 1) = F(n)$$
Try Proving it!

Proof: By induction on \( n > 1 \).

Basis For \( n = 0 \) and \( n = 1 \), \( F_a(0, 1, 1) = 1 = F_a(1) \)

Induction hypothesis (IH) Assume \( F_a(k, 1, 1) = F(k) \), for some \( k > 1 \)

Induction Step

\[
F_a(k + 1, 1, 1) = F_a(k, 1, 2) \quad \text{Definition of } F_a \\
= \ ? \ ? \ ? \quad \text{IH}
\]

STUCK!
Another Generalization

Try to prove a different and more direct theorem.

**Theorem 3** For all integers \( n \geq 1 \) and \( j \geq 1 \),

\[
F_a(n, F(j - 1), F(j)) = F(n + j - 1)
\]

**Proof:** By induction on \( n \geq 1 \), for all values of \( j \geq 1 \).

**Corollary 4** For all integers \( n \geq 1 \),

\[
F_a(n, F(0), F(1)) = F(n)
\]
Try Proving it!

**Basis** For \( n = 1 \), \( F_a(1, F(j - 1), F(j)) = F(j) \)

**Induction hypothesis (IH)** For some \( k > 1 \) and all \( j \geq 1 \),
\[
F_a(k, F(j - 1), F(j)) = F(k + j - 1)
\]

**Induction Step** We need to prove
\[
F_a(k + 1, F(j - 1), F(j)) = F(k + j).
\]

\[
F_a(k + 1, F(j - 1), F(j)) = F_a(k, F(j), F(j - 1) + F(j))
\]
\[
= F_a(k, F(j), F(j + 1))
\]
\[
= F_a(k + (j + 1) - 1) \quad \text{IH}
\]
\[
= F_a(k + j)
\]
Complexity

- Time complexity:
  - No of additions: $A_F(n)$
  - No of comparisons: $C_F(n)$
  - No of recursive calls to $F$: $R_F(n)$

- Space complexity:
Complexity

- Time complexity:
- Space complexity:
  - left-to-right evaluation: $\mathcal{LR}_F(n)$
  - arbitrary evaluation: $U_F(n)$
Time Complexity: $\mathcal{R}$

- Hardware operations like addition and comparisons are usually very fast compared to software operations like recursion unfolding.
- The number of recursion unfoldings also includes comparisons and additions.
Time Complexity

- It is enough to put bounds on the number of recursion unfoldings and not worry about individual hardware operations.
- Similar theorems may be proved for any operation by counting and induction.

So we concentrate on $R$. 
Time Complexity: $\mathcal{R}$

- $\mathcal{R}_F(0) = \mathcal{R}_F(1) = 0$
- $\mathcal{R}_F(n) = 2 + \mathcal{R}_F(n - 1) + \mathcal{R}_F(n - 2)$ for $n > 1$

To solve the equation as initial value problem and obtain an upper bound we guess the following theorem.

**Theorem 5** $\mathcal{R}_F(n) \leq 2^{n-1}$ for all $n > 2$

**Proof:** By induction on $n > 2$. \(\square\)
Bound on $\mathcal{R}$

**Basis** $n = 3$. $\mathcal{R}_F(3) = 2 + 2 + 0 \leq 2^{3-1}$

**Induction hypothesis (IH)** For some $k > 2$, $\mathcal{R}_F(k) \leq 2^{k-1}$

**Induction Step** If $n = k + 1$ then $n > 3$

\[
\begin{align*}
\mathcal{R}_F(n) &= 2 + \mathcal{R}_F(n - 2) + \mathcal{R}_F(n - 1) \\
&\leq 2 + 2^{n-3} + 2^{n-2} \quad \text{(IH)} \\
&\leq 2.2^{n-3} + 2^{n-2} \quad \text{for } n > 3, \ 2^{n-3} \geq 2 \\
&= 2^{n-2} + 2^{n-2} \\
&= 2^{n-1}
\end{align*}
\]
Other Bounds: $C_F$

One comparison for each call.

• $C_F(0) = C_F(1) = 1$

• $C_F(n) = 1 + C_F(n-1) + C_F(n-2)$ for $n > 1$

**Theorem 6** $C_F(n) \leq 2^n$ for all $n \geq 0$. 
Other Bounds: $A_F$

No additions for the basis and one addition in each call.

- $A_F(0) = A_F(1) = 0$
- $A_F(n) = 1 + A_F(n - 1) + A_F(n - 2)$ for $n > 1$

**Theorem 7** $A_F(n) \leq 2^{n-1}$ for all $n > 0$. 
1.5. Primitives: Booleans

1. Boolean Conditions
2. Booleans in SML
3. Booleans in SML
4. $\land$ vs. andalso
5. $\lor$ vs. orelse
6. SML: orelse
7. SML: andalso
8. and, andalso, ⊥
9. or, orelse, ⊥
10. Complex Boolean Conditions
Boolean Conditions

- Two (truth) value set: \{true, false\}
- Boolean conditions are those statements or names which can take only truth values.
  Examples: \(n < 0\), true,

- Negation operator: **not**
  Examples: not \((n < 0)\), not true, not false
Booleans in SML

Standard ML of New Jersey,
- val tt = true;
val tt = true : bool
- not(tt);
val it = false : bool
- val n = 10;
val n = 10 : int
- n < 10;
val it = false : bool
- not (n<10);
val it = true : bool
-
Booleans in SML

Examples:
- \((n \geq 10)\) andalso \((n=10)\);
  val it = true : bool
- \(n < 0\) orelse \(n \geq 10\);
  val it = true : bool
- \(\neg ((n \geq 10) = \text{andalso} (n=10)) = \text{andalso} (n=10)\)
  \(= \text{andalso} (n=10)\) = orelse \(n < 0\) orelse \(n \geq 10\);
  val it = true : bool
-
\( \land \) vs. `andalso`

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td>( q )</td>
<td>( p \land q )</td>
<td>( p \text{ andalso} q )</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>
\( \lor \) vs. `orelse`

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>( p \lor q )</th>
<th><code>p orelse q</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
SML: `orelse`

Standard ML of New Jersey,
- `val tt = true;
val tt = true : bool`
- `val ff = false;
val ff = false : bool`
- `fun gtz n = if n=1 then true =
  else gtz (n-1);
val gtz = fn : int -> bool`
- `tt orelse (gtz 0);
val it = true : bool`
-
SML: andalso

- (gtz 0) orelse tt;

Interrupt
- ff andalso (gtz 0);
val it = false : bool
- (gtz 0) andalso ff;

Interrupt
-
and, andalso, ⊥

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>q</td>
<td>p ∧ q</td>
<td>p andalso q</td>
</tr>
<tr>
<td>true</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>⊥</td>
<td>true</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>false</td>
<td>⊥</td>
<td>false</td>
<td>⊥</td>
</tr>
<tr>
<td>⊥</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

∧ is commutative whereas andalso is not.
or, \texttt{orelse}, \bot

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$p \lor q$</th>
<th>$p \texttt{orelse} q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>\bot</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>\bot</td>
<td>true</td>
<td>true</td>
<td>\bot</td>
</tr>
<tr>
<td>false</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
</tr>
<tr>
<td>\bot</td>
<td>false</td>
<td>\bot</td>
<td>\bot</td>
</tr>
</tbody>
</table>

$\lor$ is commutative whereas $\texttt{orelse}$ is not.
Complex Boolean Conditions

Assume $p$ and $q$ are boolean conditions

\[ p \text{ orelse } q \equiv \text{if } p \text{ then true else } q \]

\[ p \text{ andalso } q \equiv \text{if } p \text{ then } q \text{ else false} \]
2. Algorithms: Design & Refinement

2.1. Technical Completeness & Algorithms

1. Recapitulation: Integers & Real
2. Recap: Integer Operations
3. Recapitulation: Real Operations
4. Recapitulation: Simple Algorithms
5. More Algorithms
6. Powering: Math
7. Powering: SML
8. Technical completeness
9. What SML says
10. Technical completeness
11. What SML says ... contd
12. Powering: Math 1
13. Powering: SML 1
14. Technical Completeness
15. What SML says
16. Powering: Integer Version
17. Exceptions: A new primitive
18. Integer Power: SML
19. Integer Square Root 1
20. Integer Square Root 2
21. An analysis
22. Algorithmic idea
23. Algorithm: isqrt
24. Algorithm: shrink
25. SML: shrink
26. SML: intsqr
27. Run it!
28. SML: Reorganizing Code
29. Intsqr: Reorganized
30. shrink: Another algorithm
31. Shrink2: SML
32. Shrink2: SML ... contd
Recapitulation: Integers & Real

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

● Primitive Integer Operations
  – Naming, +, −, ∼
  – 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**
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

```ml
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

$$0.0^n = \frac{1.0}{(0.0^m)} = \frac{1.0}{0.0}$$

Division by zero!
What SML says

- \texttt{power (2.5, \sim 2)};
  \texttt{val it = 0.16 : real}
- \texttt{power (\sim 2.5, \sim 2)};
  \texttt{val it = 0.16 : real}
- \texttt{power (0.0, 2)};
  \texttt{val it = 0.0 : real}
- \texttt{power (0.0, \sim 2)};
  \texttt{val it = inf : real}

\textbf{SML is somewhat more understandable than most languages}
Powering: Integer Version

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

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

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

exception negExponent;
exception zeroPowerZero;
fun intpower (x, n) =
  if n < 0
  then raise negExponent
else if n = 0
  then if x=0
    then raise zeroPowerZero
  else 1
else intpower (x, n-1) * x
- `intpower(3, 4);`
  `val it = 81 : int`
- `intpower(\sim3, 5);`
  `val it = \sim243 : int`
- `intpower(3, \sim4);`

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

- `intpower(0, 0);`

uncaught exception `zeroPowerZero`
raised at: `stdin:24.26-24.39`
\[ isqrt(n) = \lfloor \sqrt{n} \rfloor \]

- 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
Suppose `Real.Math.sqrt` were not available to us!

The function \( isqrt(n) \) of a non-negative integer \( n \) is the integer \( k \geq 0 \) such that \( k^2 \leq n < (k + 1)^2 \).

That is,

\[
isqrt(n) = \begin{cases} \bot & \text{if } n < 0 \\ k & \text{otherwise} \end{cases}
\]

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 \( isqrt(n) = 0 \).

Otherwise with \([l, u] = [0, n]\) and

\[
\begin{align*}
    l^2 &\leq n < u^2
\end{align*}
\]

use one or both of the following to shrink the interval \([l, u]\).

- if \((l + 1)^2 \leq n\) then try \([l + 1, u]\)
  otherwise \(l^2 \leq n < (l + 1)^2\) and \(k = l\)

- if \(u^2 > n\) then try \([l, u - 1]\)
  otherwise \((u - 1)^2 \leq n < u^2\) and \(k = u - 1\)
Algorithm: isqrt

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

where
Algorithm: shrink

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

shrink
Run it!

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

```haskell
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

\[
\text{shrink2}(n, l, u) = \begin{cases} 
  l & \text{if } l = u \text{ or } u = l + 1 \\
  \text{shrink2}(n, m, u) & \text{if } l < u \\
  \text{shrink2}(n, l, m) & \text{if } l < u \\
  \bot & \text{if } l > u \\
\end{cases} \\
\text{where } m = \left(\frac{l + u}{2}\right)
\]
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
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;
2.2. Algorithm Refinement

1. Recap: More Algorithms
2. Recap: Power
3. Recap: Technical completeness
4. Recap: More Algorithms
5. Intsqrt: Reorganized
6. Intsqrt: Reorganized
7. Some More Algorithms
8. Combinations: Math
9. Combinations: Details
10. Combinations: SML
11. Perfect Numbers
12. Refinement
13. Perfect Numbers: SML
14. $\sum_{i}^{u} \text{ifdivisor}(k)$
15. SML: sum_divisors
16. ifdivisor and ifdivisor
17. SML: Assembly 1
18. SML: Assembly 2
19. Perfect Numbers . . . contd.
20. Perfect Numbers . . . contd.
21. SML: Assembly 3
22. Perfect Numbers: Run
23. Perfect Numbers: Run
24. SML: Code variations
25. SML: Code variations
26. SML: Code variations
27. Summation: Generalizations
28. Algorithmic Improvements:
29. Algorithmic Variations
30. Algorithmic Variations
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? \textbf{YES}
- $n$ will be integer? \textbf{YES}
- $n$ will be non-negative? \textbf{NO}
- $x \neq 0$? \textbf{NO}

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

\textbf{INFINITE COMPUTATION}
Recap: More Algorithms

- $x^n$ for real and integer $x$
- Integer square root
  - Analysis
  - Algorithmic idea
  - Algorithm
  - where
  - and let ... in ... end
exception negError;
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

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

- Combinations
- Perfect Numbers
Combinations: Math

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

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

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

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

Since we already have the function `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 \( 0 \leq k \leq n \)

\[
n C_k = \begin{cases} 
  \bot & \text{if } n < 0 \text{ or } k < 0 \text{ or } k > n \\
  1 & \text{if } n = 0 \text{ or } k = 0 \text{ or } k = n \\
  n^{-1}C_{k-1} + n^{-1}C_k & \text{otherwise}
\end{cases}
\]
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);
Perfect Numbers

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

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

$$\text{perfect}(n)$$

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

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

where
Refinement

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

2. $\sum_{k=1}^{n-1} \text{ifdivisor}(k)$ 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 \[
\sum_{k=l}^{u} \text{ifdivisor}(k)
\]

fun \texttt{sum\_divisors} (l, u) =
  if l > u
  then 0
  else ifdivisor (l) +
  \texttt{sum\_divisors} (l+1, u)

\text{where} \texttt{ifdivisor}(k) \text{ still needs to be defined}
ifdivisor and ifdivisor

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

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

Not technically complete!
However ...
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
    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 \ldots contd.

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

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

Hence

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

Hence

\[
\text{perfect}(n) \quad \iff \quad n = \sum_{k=1}^{n-1} \text{if divisor}(k)
\]

\[
\iff \quad n = \sum_{k=1}^{n} \frac{\text{div} 2}{\text{if divisor}(k)}
\]

where

\[
\text{if divisor}(k) = \begin{cases} 
  k & \text{if } k \mid n \\
  0 & \text{otherwise}
\end{cases}
\]
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 $k \in [l, u] = [1, n \div 2]$ 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 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
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?

```ml
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 $f : \mathbb{Z} \to \mathbb{Z}$ 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. perfect2
2. shrink2
Algorithmic Variations

1. For any $k | n$, $m = n \text{ div } k$ 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 $\sum_{k=1}^{n} \text{div } 2$ if divisor$(k) = 1 + \sum_{k=2}^{\lfloor \sqrt{n} \rfloor} \text{ifdivisor}2(k)$
Algorithmic Variations

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

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

where

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

Are there any glitches? Is it technically correct and complete?
2.3. Variations: Algorithms & Code
Recap

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

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

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

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

\[ = n^{-1}C_{k-1} + n^{-1}C_k \]
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)
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 \( m = (n \ \text{div} \ k) \)
Power 2

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
A Faster Power

\[ x^n = \underbrace{x \times x \times \cdots \times x}_{n/2 \text{ times}} \times \underbrace{x \times x \times \cdots \times x}_{n/2 \text{ times}} \]

when \( n \) is even and

\[ x^n = \underbrace{x \times x \times \cdots \times x}_{\lfloor n/2 \rfloor \text{ times}} \times \underbrace{x \times x \times \cdots \times x}_{\lfloor n/2 \rfloor \text{ times}} \times x \]

when \( n \) is odd
Power2: Complete

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

where \( \text{even}(n) \iff n \mod 2 = 0. \)
Power2: SML

fun power2 (x, n) =
  if n < 0
  then 1.0/power2 (x, ~n)
  else if n = 0
  then 1.0
  else
Power2: SML

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?
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

Incorrect version

Definition of a name before use:

• $\text{ifdivisor}(k)$ is defined first.
• $\text{idivisor}(k)$ uses the name $n$ without defining it.
• $k$ has been defined (as an argument of $\text{ifdivisor}(k)$) 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

incorrect version

• 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

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

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

and \( perfect(n) \iff n = sum\_divisors(1, \lfloor n/2 \rfloor) \)
Incorrectness

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

Let’s make it explicit!
ifdivisor3

Let

$$ifdivisor3(n, k) = \begin{cases} k & \text{if } k|n \\ 0 & \text{otherwise} \end{cases}$$

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 $perfect(n) \iff n = sum\_divisors(1, \lfloor n/2 \rfloor)$
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
Try it!

- fun sum_divisors (l, u) =
  = if l > u
  = then 0
  = else ifdivisor3 (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 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

\[ \text{ifdivisor}_3(n, k) = \begin{cases} k & \text{if } k \mid n \\ 0 & \text{otherwise} \end{cases} \]

and $\text{sum\_divisors}_2(n, l, u) = \begin{cases} 0 & \text{if } l > u \\ \text{ifdivisor}_3(n, l) + \ \text{sum\_divisors}(l + 1, u) & \text{otherwise} \end{cases}$

and $\text{perfect}(n) \iff n = \text{sum\_divisors}_2(n, 1, \lfloor n/2 \rfloor)$
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
2.4. Names, Scopes & Recursion

1. Disjoint Scopes
2. Nested Scopes
3. Overlapping Scopes
4. Spannning
5. Scope & Names
6. Names & References
7. Names & References
8. Names & References
9. Names & References
10. Names & References
11. Names & References
12. Names & References
13. Names & References
14. Names & References
15. Names & References
16. Definition of Names
17. Use of Names
18. local...in...end
19. local...in...end
20. local...in...end
21. local...in...end
22. Scope & local
23. Computations: Simple
24. Simple computations
25. Computations: Composition
26. Composition: Alternative
27. Compositions: Compare
28. Compositions: Compare
29. Computations: Composition
30. Recursion
31. Recursion: Left
32. Recursion: Right
Disjoint Scopes

\[
\begin{aligned}
\text{let} & \\
\text{val } x = 10; & \\
\text{fun } \text{fun1} & \\
\text{y = } & \\
\text{let} & \\
\text{...} & \\
\text{in} & \\
\text{...} & \\
\text{end} & \\
\text{fun } \text{fun2} & \\
\text{z = } & \\
\text{let} & \\
\text{...} & \\
\text{in} & \\
\text{...} & \\
\text{end} & \\
\text{in} & \\
\text{fun1 } (\text{fun2 } x) & \\
\text{end} & 
\end{aligned}
\]
let
val x = 10;
fun fun1
let
val x = 15
in
x + y
end
in
fun1 x
end
Overlapping Scopes

```
let
val x = 10;
fun fun1  y =
...
...
...
...
fun1 (fun2 x)
end
```
val x = 10;

fun1 (fun2 x)
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

val z = 5;

* z
let
val \textbf{x} = 10; val \underline{z} = 5;
fun \textbf{fun1} \ y =
  let
  val \underline{x} = 15
  in
  x + y * z
  end
end
fun1 \ x
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
end

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

Back to scope names
let
val \( \mathbf{x} = 10; \) val \( \mathbf{z} = 5; \)
fun \( \text{fun1} \)
\[
\text{fun fun1 } \ y = \ \\
\begin{array}{l}
\text{let} \\
\text{val } \mathbf{x} = 15 \\
\text{in} \\
\mathbf{x} + \ y \ * \ \mathbf{z} \\
\text{end}
\end{array}
\]
val \( \mathbf{z} = 5; \)
end
\]
let

val x = 10; val z = 5;

fun fun1

end

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 fun1
  x
fun fun1  y =
  let
    val x = 15
    in
    x + y
  end
val z = 5;
* z
end
Back to scope names
\[
\text{let } \begin{align*}
\text{val } & \mathbf{x} = 10; \quad \text{val } \mathbf{x} = \mathbf{x} - 5; \\
\text{fun } & \text{fun1 } \begin{align*}
\text{y } = & \begin{align*}
\text{let } & \ldots \\
\text{in } & \ldots \\
\text{end }
\end{align*} \\
\text{fun } & \text{fun2 } \begin{align*}
\text{z } = & \begin{align*}
\text{let } & \ldots \\
\text{in } & \ldots \\
\text{end }
\end{align*} \\
\text{in } & \text{fun1 } (\text{fun2 } \mathbf{x}) \\
\text{end }
\end{align*}
\end{align*}
\]
let
val \(x = 10\); val \(x = x - 5\);

fun fun1
  \(y = \)
    let
      ... 
    in
      ...
    end

fun fun2
  \(z = \)
    let
      ... 
    in
      ...
    end

d in fun1 (fun2 \(x\))
end
let
val \(x = 10\); val \(x = x - 5\);
fun fun1
  \(y = \ldots\)
in
fun fun2
  \(z = \ldots\)
in
  \(\ldots\)
end
in fun1 (fun2 \(x\))
end
Definition of Names

Definitions are of the form

\[ \text{qualifier } \underline{\text{name}} \ldots = \text{body} \]

- \[ \text{val } \underline{\text{name}} = \]
- \[ \text{fun } \underline{\text{name}} \left( \underline{\text{argnames}} \right) = \]
- \[ \text{local definitions in definition end} \]
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

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)
local...in...end

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
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

\[
\begin{align*}
\text{local} & \quad \text{fun fun1} \quad y = \ldots \\
& \quad \text{fun fun2} \quad z = \ldots \\
\text{fun fun3} & \quad x = \ldots \\
\text{end} & \quad \text{fun2} \ldots \\
& \quad \text{fun1} \\
& \quad \text{fun1} 
\end{align*}
\]
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

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

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

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

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

\[ fact_L(4) = (fact_L(4 - 1) \times 4) = (fact_L(3) \times 4) = ((fact_L(3 - 1) \times 3) \times 4) = ((fact_L(2) \times 3) \times 4) = (((fact_L(2 - 1) \times 2) \times 3) \times 4) = \ldots \]
Recursion: Right

\[ \text{factR}(4) \]
\[ = (4 \times \text{factR}(4 - 1)) \]
\[ = (4 \times \text{factR}(3)) \]
\[ = (4 \times (3 \times \text{factR}(3 - 1))) \]
\[ = (4 \times (3 \times \text{factR}(2))) \]
\[ = (4 \times (3 \times (2 \times \text{factR}(2 - 1)))) \]
\[ = \ldots \]
3. Introducing Reals

3.1. Floating Point

1. So Far-1: Computing
2. So Far-2: Algorithms & Programs
3. So far-3: Top-down Design
4. So Far-4: Algorithms to Programs
5. So far-5: Caveats
6. So Far-6: Algorithmic Variations
7. So Far-7: Computations
8. Floating Point
9. Real Operations
10. Real Arithmetic
11. Numerical Methods
12. Errors
13. Errors
14. Infinite Series
15. Truncation Errors
16. Equation Solving
17. Root Finding-1
18. Root Finding-2
19. Root Finding-3
20. Root Finding-4
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

- 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

Perfect Numbers
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 and 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
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

\[ f(b) \]

\[ a \]

\[ f(a) \]

\[ x_0 \]

\[ b \]
Root Finding-2
Root Finding-3

The diagram illustrates a function $f(x)$ with a root $x_0$ between $a$ and $b$. The function values at $a$ and $b$ are $f(a)$ and $f(b)$ respectively.
Root Finding-4

Rather **steep** isn’t it?

![Graph showing a function with points a, b, f(a), f(b), x₀, and ε.](image)
3.2. Root Finding, Composition and Recursion

1. Root Finding: Newton's Method
2. Root Finding: Newton's Method
3. Root Finding: Newton's Method
4. Root Finding: Newton's Method
5. Root Finding: Newton's Method
6. Root Finding: Newton’s Method
7. Newton’s Method: Basis
8. Newton’s Method: Basis
10. What can go wrong!-1
11. What can go wrong!-2
12. What can go wrong!-2
13. What can go wrong!-3
14. What can go wrong!-4
15. Real Computations & Induction: 1
16. Real Computations & Induction: 2
17. What’s it good for? 1
18. What's it good for? 2
19. newton: Computation
20. Generalized Composition
21. Two Computations of \( h(1) \)
22. Two Computations of \( h(-1) \)
23. Recursive Computations
24. Recursion: Left
25. Recursion: Right
26. Recursion: Nonlinear
27. Some Practical Questions
28. Some Practical Questions
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

\[ f(b) \]

\[ f(a) \]

\[ a \]

\[ b \]

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

\[ f(b) \]

\[ a \]

\[ x_{i+1} \]

\[ x_i \]

\[ f(a) \]

\[ b \]
Root Finding: Newton’s Method

\[ f(b) \]

Graph showing the iteration process of Newton's Method with points \( a \), \( x_i \), \( x_{i+1} \), and \( b \).
Root Finding: Newton’s Method

\[ f(b) \]

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

\[ f(a) \]
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.

\[ f(x_0), f(x_1), f(x_2), \ldots \to 0 \]
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’s 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!

![Oscillation Diagram]
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!
What can go wrong!-3

The function may be too steep

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

Or too shallow!

\[ f(a) \quad f(b) \quad x_0 \quad a \quad b \]

\[ \int_{a}^{b} f(x) \, dx \]

\[ f(a) \quad f(b) \]

\[ x_0 \]
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 \( \epsilon \),

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

\[ \text{newton}(f, f', a, b, \varepsilon, x_0) \]
\[ \mapsto \text{newton}(f, f', a, b, \varepsilon, x_1) \]
\[ \mapsto \text{newton}(f, f', a, b, \varepsilon, x_2) \]
\[ \mapsto \text{newton}(f, f', a, b, \varepsilon, x_3) \]
\[ \vdots \]
\[ \mapsto \text{newton}(f, f', a, b, \varepsilon, x_n) \]
\[ \mapsto x_n \]
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 $h(1)$

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

$h(-1)$
\[\leadsto f(-1, g(-1))\]
\[\leadsto 0\]
DONE!

$h(-1)$
\[\leadsto f(-1, g(-1))\]
\[\leadsto f(-1, (g(-2) + 1))\]
\[\leadsto \ldots\]
\[\leadsto \text{FOREVER!}\]
Recursive Computations

- Newton’s method
- Factorial
  - $\text{fact}_L$
  - $\text{fact}_R$

skip to nonlinear recursion
skip to Recursion Revisited
Recursion: Left

\[
factL(4) \\
\Rightarrow (factL(4 - 1) \times 4) \\
\Rightarrow (factL(3) \times 4) \\
\Rightarrow ((factL(3 - 1) \times 3) \times 4) \\
\Rightarrow (((factL(2) \times 3) \times 4) \\
\Rightarrow (((factL(2 - 1) \times 2) \times 3) \times 4) \\
\Rightarrow \ldots
\]
Recursion: Right

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

Fibonacci

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

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?
4. Correctness, Termination & Complexity

4.1. Termination and Space Complexity

1. Recursion Revisited
2. Linear Recursion: Waxing
3. Recursion: Waning
4. Nonlinear Recursions
5. Fibonacci: contd
6. Recursion: Waxing & Waning
7. Unfolding Recursion
8. Non-termination
9. Termination
10. Proofs of termination
11. Proofs of termination: Induction
12. Proof of termination: Factorial
13. Proof of termination: Factorial
14. Fibonacci: Termination
15. GCD computations
16. Well-foundedness: GCD
17. Well-foundedness
18. Induction is Well-founded
19. Induction is Well-founded
20. Where it doesn’t work
21. Well-foundedness is inductive
22. Well-foundedness is inductive
23. GCD: Well-foundedness
24. Newton: Well-foundedness
25. Newton: Well-foundedness
26. Example: Zero
27. Questions
28. The Collatz Problem
29. Questions
30. Space Complexity
31. Newton & Euclid: Absolute
32. Newton & Euclid: Relative
33. Deriving space requirements
34. GCD: Space
35. Factorial: Space
36. Fibonacci: Space
37. Fibonacci: Space
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

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

contrast with newton
Nonlinear Recursions

Fibonacci

- Each computation of \( \text{fib} \) has its own waxing and waning
- There is still an “envelope” which shows waxing and waning.
Fibonacci: contd

\[ (((1 + 1) + \text{fib}(1)) + \text{fib}(2)) + \text{fib}(3) \]
\[ \Rightarrow (2 + \text{fib}(1)) + \text{fib}(2) + \text{fib}(3) \]
\[ \Rightarrow ((2 + 1) + \text{fib}(2)) + \text{fib}(3) \]
\[ \Rightarrow \ldots \]
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: Induction

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 \( factL \) defined only for non-negative integers. We prove that it is an algorithm i.e. that it terminates.

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

**Induction hypothesis**. For some $n > 0$, $\forall k : 0 \leq k \leq n : f_{actL}(k)$ terminates in $\propto k$ steps.

**Induction step**. Then $f_{actL}(n + 1) = f_{actL}(n) \ast (n + 1)$ is guaranteed to terminate in $\propto (n + 1)$ steps, since $f_{actL}(n)$ 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 \( n > 0 \),
\[
\forall k : 0 \leq k \leq n : \text{fib}(k) \text{ terminates in } \propto f(k) \text{ steps}
\]

**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

\[ \text{gcd}(12, 64) \]
\[ \rightsquigarrow \text{gcd}(64, 12) \]
\[ \rightsquigarrow \text{gcd}(12, 4) \]
\[ \rightsquigarrow \text{gcd}(4, 0) \]
\[ \rightsquigarrow 4 \]
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 \( \text{w}(n) \) no longer strictly non-negative)

• \( \text{fact}(n) = \text{fact}(n+1) \div (n+1) \), even if \( n \) is non-negative (since \( \text{w}(n) \) 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

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

\[ w(x) = \left\lfloor |x| \right\rfloor \] 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.* \( \text{collatz}(m) \to 1 \text{ for all } m. \)
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 & Euclid: Absolute

Newton’s Method
Computation

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 . . .

Euclidean GCD
Computation
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 newton
Factorial: Space

\[ S_{factL}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
S_{factL}(n-1) + 1 & \text{otherwise}
\end{cases} \]

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

\[ S_{factL}(n) \propto n. \]

A similar analysis and result holds for \(factR\).
Fibonacci: Space

\[ S_{fib}(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
S_{fib}(n-1) + S_{fib}(n-2) & \text{if } n > 1 
\end{cases} \]
Fibonacci: Space

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

\[
2^{n-2} < S_{fib}(n) \leq 2^{n-1}
\]
4.2. Efficiency Measures and Speed-ups

1. Recapitulation  
2. Recapitulation  
3. Time & Space Complexity  
4. $\sqrt{\text{isqrt}}$: Space  
5. Time Complexity  
6. $\sqrt{\text{isqrt}}$: Time Complexity  
7. $\sqrt{\text{isqrt2}}$: Time  
8. $\text{shrink vs. shrink2}$: Times  
9. Factorial: Time Complexity  
10. Fibonacci: Time Complexity  
11. Comparative Complexity  
12. Comparisons  
13. Comparisons  
14. Efficiency Measures: Time  
15. Efficiency Measures: Space  
16. Speeding Up: 1  
17. Speeding Up: 2
18. Factoring out calculations
19. Tail Recursion: 1
20. Tail Recursion: 2
21. Factorial: Tail Recursion
22. Factorial: Tail Recursion
23. A Computation
24. Factorial: Issues
25. Fibonacci: Tail Recursion
26. Fibonacci: Tail Recursion
27. fibTR: SML
28. State in Tail Recursion
29. Invariance
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
\textit{isqrt}: Space

Integer Sqrt \quad 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**

**Integer Sqrt shrink**

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

Then $T_{\text{shrink}}(n,l,u) =$

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

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

$$T_{\text{isqrt}}(n) = T_{\text{shrink}}(n,0,n) \propto n$$
Assume condition-checking (along with \((l + u) \text{ div } 2\)) takes a unit of time. Then

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

If \(2^{k-1} \leq [\lfloor l, u \rfloor] - 1 < 2^k\) then the algorithm terminates in at most \(k\) steps. Since \(k = \lceil \log_2 [\lfloor l, u \rfloor] - 1 \rceil\),

\[
T_{\text{shrink}2}(n,l,u) \propto \lceil \log_2 [\lfloor l, u \rfloor] - 1 \rceil
\]

\[
T_{\text{isqrt}2}(n) \propto \lceil \log_2 n \rceil
\]
1. The time units are **different**, 
2. But they differ by a **constant** factor at most. 
3. So clearly, for **large** $n$, $ shrink2 $ is **faster** than $ 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.

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

Then

\[ T_{\text{fact}L}(n) \propto n \]
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

\[ O(n^2) \]
\[ O(2^n) \]
\[ O(n \log n) \]
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.
Speeding Up: 1

Q: Can fibonacci be speeded up or made more space efficient?

A: Perhaps by studying the nature of the function e.g. \textit{isqrt2} vs. \textit{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 $gcd$, $newton$, $shrink$
Factoring out calculations

$$gcd(a_0, b_0)$$
compute $a_1, b_1$

$\leadsto gcd(a_1, b_1)$
compute $a_2, b_2$

$\leadsto gcd(a_2, b_2)$

$\leadsto \ldots$

$\leadsto gcd(a_n, b_n)$

$\leadsto a_n$
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

\[ \text{fact}_L \text{ Waxing} \quad \text{fact}_L \text{ Waning} \]

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

\[
factL2(n) = \begin{cases} 
\bot & \text{if } n < 0 \\
1 & \text{if } n = 0 \\
factL_tr(n, 1) & \text{otherwise}
\end{cases}
\]

where

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

\[
\begin{align*}
\text{factL2}(4) & \rightarrow \text{factL_tr}(4, 1) \\
& \rightarrow \text{factL_tr}(3, 4) \\
& \rightarrow \text{factL_tr}(2, 12) \\
& \rightarrow \text{factL_tr}(1, 24) \\
& \rightarrow \text{factL_tr}(0, 24) \\
& \rightarrow 24
\end{align*}
\]

Reminiscent of \texttt{gcd} and \texttt{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

\[ \text{fibTR}(n) = \begin{cases} \perp & \text{if } n < 0 \\ 1 & \text{if } 0 \leq n \leq 1 \\ \text{fib_iter}(n, 1, 1, 1) & \text{otherwise} \end{cases} \]

where

\[ \text{fib_iter}(n, a, b, m) = \begin{cases} b & \text{if } m \geq n \\ \text{fib_iter}(n, b, a + b, m + 1) & \text{otherwise} \end{cases} \]
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)
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
4.3. Invariance & Correctness

1. Recap
2. Recursion Transformation
3. Tail Recursion: Examples
4. Comparisons
5. Transformation Issues
6. Correctness Issues 1
7. Correctness Issues 2
8. Correctness Theorem
9. Invariants & Correctness 1
10. Invariants & Correctness 2
11. Invariance Lemma: $factL_{tr}$
12. Invariance: Example
13. Invariance: Example
14. Proof
15. Invariance Lemma: $fib_{iter}$
16. Proof
17. Correctness: Fibonacci
18. Variants & Invariants
19. Variants & Invariants
20. More Invariants
21. Fast Powering 1
22. Fast Powering 2
23. Root Finding: Bisection
24. Advantage Bisection
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 $f(\vec{x}) = A(\vec{x})$ for all argument values $\vec{x}$ 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 factL2

Theorem 8 For all $n \geq 0$, $\text{factL2}(n) = n!$

Proof: For $n = 0$, it is clear that $\text{factL2}(0) = 1 = 0!$. For $n > 0$, $\text{factL2}(n) = \text{factL_tr}(n, 1)$. The proof is done provided we can show that $\text{factL_tr}(n, 1) = n!$. □
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 \( \text{fact}L2 \)

- 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: \( factL_{tr} \)

Invariant properties \( factL_2 \)

**Lemma 9** For all \( n \geq 0 \) and \( p \)

\[
 factL_{tr}(n, p) = (n!)p
\]

**Proof:** By induction on \( n \).
Invariance: Example

\[ \text{factL2} \]

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

Contrast with a \( \text{factL2}(4) \) computation
Invariance: Example

So what exactly is invariant?

\[ \text{fact}_L(4, 7) \quad 168 = 4! \times 7 \]
\[ \Rightarrow \quad \text{fact}_L(3, 28) \quad 168 = 3! \times 28 \]
\[ \Rightarrow \quad \text{fact}_L(2, 84) \quad 168 = 2! \times 84 \]
\[ \Rightarrow \quad \text{fact}_L(1, 168) \quad 168 = 1! \times 168 \]
\[ \Rightarrow \quad \text{fact}_L(0, 168) \quad 168 = 0! \times 168 \]
\[ \Rightarrow \quad 168 \]
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**
\[
\begin{align*}
\text{factL_tr}(n + 1, p) \\
= & \text{factL_tr}(n, (n + 1)p) \\
= & (n!)(n + 1)p \\
= & (n + 1)!p
\end{align*}
\]

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

Lemma 10 \textit{For all} \( n > 1, a, b, m : 1 \leq m \leq n \), \( \text{if} \ a = F(m - 1) \ \text{and} \ b = F(m) \), \( \text{then} \)

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

\textit{Proof:} By induction on \( k = n - m \) \( \square \)
Proof

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

**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

$$
\text{fib}_\text{iter}(n, a, b, m) = \text{fib}_\text{iter}(n, b, a + b, m + 1) = F(n) \quad (IH)
$$
Correctness: Fibonacci

\[ fibTR(n) = F(n). \]

**Theorem 11** For all \( n \geq 0 \),

\[ fibTR(n) = F(n). \]

**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

\[ f_{actL3}(n) = \begin{cases} 
\bot & \text{if } n < 0 \\
1 & \text{if } n = 0 \\
f_{actL_{tr}2}(n, 1, 1) & \text{else}
\end{cases} \]

where
Variants & Invariants

\( \text{factL}_2 \)

\[
\text{factL}_{tr2}(n, p, m) = \begin{cases} 
  p & \text{if } n = m \\
  \text{factL}_{tr2}(n, (m + 1)p, m + 1) & \text{else}
\end{cases}
\]

\[\text{factL}_{tr2}(n, p, m) = (m!)p\]

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

- **shrink** For all $n > 0$, $l$, $u$, if $[l, u] \subseteq [0, n]$,

  $l \leq \lfloor \sqrt{n} \rfloor \leq u$

- **shrink 2**

  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

\[ \text{power3}(x, n) = \begin{cases} 
1.0 / \text{power3}(x, -n) & \text{if } n < 0 \\
1.0 & \text{if } n = 0 \\
\text{powerTR}(x, n, 1) & \text{else}
\end{cases} \]

where
Fast Powering 2

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

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

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

Newton’s Method

Algorithm

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

$$\text{bisect}(f, a, b, \varepsilon) =$$

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

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.
5. Compound Data

5.1. Tuples, Lists & the Generation of Primes

1. Recap: Tail Recursion
2. Examples: Invariants
3. Tuples
4. Lists
5. New Lists
6. List Operations
7. List Operations: \textit{cons}
8. Generating Primes upto \(n\)
9. More Properties
10. Composites
11. Odd Primes
12. \textit{primesUpto}(n)
13. \textit{generateFrom}(P, m, n, k)
14. \textit{generateFrom}
15. \textit{primeWRT}(m, P)
16. \textit{primeWRT}(m, P)
17. \textit{primeWRT}
18. Density of Primes
19. The Prime Number Theorem
20. The Prime Number Theorem
21. Complexity
22. Diagnosis
Recap: Tail Recursion

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

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

\[ \text{factL_tr2} \]

\[ \text{shrink \& shrink2} \]

\[ l \leq \lfloor \sqrt{n} \rfloor \leq u \]

\[ m = \lfloor (l + u)/2 \rfloor \quad \text{and} \quad l \leq \lfloor \sqrt{n} \rfloor \leq u \]

Fast Powering

\[ \text{powerTR}(x, n, p) = x^n p \]
Tuples: Formation

Simplest form of compound data: Cartesian products.

- Each element of a cartesian product is a **tuple**
- Tuples may be constructed as we do in mathematics, simply by enclosing the elements (separated by commas) in a pair of parentheses.

```scala
val a = (2, 3.0, false);
val a = (2,3.0,false) : int * real * bool
```
Tuples: Decomposition

- Individual components of a tuple may be taken out too.

- #1 a;
  val it = 2 : int
- #2 (a);
  val it = 3.0 : real
- #3 a;
  val it = false : bool
Tuples: \texttt{divmod}

Standard ML of New Jersey, ... 
- \texttt{fun divmod (a, b) =}
  \[(a \texttt{ div} b, a \texttt{ mod} b)\];
\texttt{val divmod =}
\texttt{fn : int * int \rightarrow int * int}
- \texttt{val dm = divmod (24, 9);}
\texttt{val dm = (2, 6) : int * int}
- \#1 \texttt{dm;}
\texttt{val it = 2 : int}
- \#2 \texttt{dm;}
\texttt{val it = 6 : int}
Constructors & Destructors

Every way of constructing compound data from simpler data elements has

Constructors: Operators which construct compound data from simpler ones (for tuples it is simply \((, , \text{ and } )\)).

Destructors: Operators which allow us to extract the individual components of a compound data item (for tuples they are \#1, \#2 ... depending upon how many components it consists of).
Tuples: Identity

Every tuple that has been broken up into its components using its destructors can be put together back again using its constructors.

Given a tuple \( \mathbf{a} \in A_1 \times A_2 \times \ldots \times A_n \), we have

\[
\mathbf{a} = (\#1 \, \mathbf{a}, \#2 \, \mathbf{a}, \ldots, \#n \, \mathbf{a})
\]
Lists

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

Given a (nonempty) list

- A list is ordered
- There may be more than one occurrence of an element in the list
- only the first element (called the 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 \([\ ]\)

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

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

**head**: \( \textit{hd} : \alpha\text{List} \to \alpha \)

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

**tail**: \( \textit{tl} : \alpha\text{List} \to \alpha\text{List} \)

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

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

\textbf{cons}: \( \text{cons} : \alpha \times \alpha \text{List} \rightarrow \alpha \text{List} \)

\[
\text{cons}(0, \text{nil}) = [0]
\]

\[
\text{cons}(0, L) = 0 :: L = [0, 1, 2, 3, 4]
\]

\[
1 :: (0 :: L) = [1, 0, 1, 2, 3, 4]
\]

back to lists Recap
Polynomial Evaluation

Evaluating a polynomial

\[ p(x) = \sum_{i=0}^{n} a_i x^i \]

given

- its coefficients as a list \([a_n, \ldots, a_0]\) of values from the highest degree term to the constant \(a_0\).
- a value for the variable \(x\).

Assume an empty list of coefficients yields a value 0.
Naive Solution

\[ poly_0(L, x) = \begin{cases} 
0 & \text{if } L = \text{nil} \\
hx^n + poly_0(T, x) & \text{if } L = h :: T 
\end{cases} \]

where \( n = |L| - 1 \).

fun \( poly_0([], x) = 0.0 \)
| \( poly_0((h::T), x) = h \ast \text{power}(x, n) + poly_0(T, x) \)
Complexity of $poly_0$

Space. $O(n)$ to store both the list and the intermediate computations.

Additions. $O(n)$ additions.

Multiplications. $n(n-1)/2$ by the simplest powering algorithm.

Multiplications. $O(\log_2(n) + O(\log_2(n-1) + \cdots + O(\log_2(1)) \leq O(n \log_2(n))$ by the fast powering algorithm.
Arden’s Rule

Factor out the multiplications to get

\[ p(x) = \ldots((a_n x + a_{n-1}) x + a_{n-2})x + \ldots)x + a_0 \]

and define a tail-recursive function which requires only \( n \) multiplications.

\[ poly_1(L, x) = poly_{TR}(0, L, x) \]

where

\[ poly_{TR}(p, L, x) = \begin{cases} 
  p & \text{if } L = \text{nil} \\
  poly_{TR}(px + h, T, x) & \text{if } L = h :: T
\end{cases} \]
poly1 in SML

local

fun poly_TR (p, [], x) = p
| poly_TR (p, (h::T), x) = poly_TR (p*x + h, T, x)
in
fun poly L x = poly_TR (0.0, L, x)
end;

Question 1. What is the right theorem to prove that poly_TR is the right generalization for the problem?

Ans.
**poly1 in SML**

```sml
clocal
  fun poly_TR (p, [], x) = p
  | poly_TR (p, (h::T), x) = poly_TR (p*x + h, T, x)
in
  fun poly L x = poly_TR (0.0, L, x)
end;
```

**Ans.** \[ poly_TR(p, L, x) = px^{n+1} + \sum_{i=0}^{n} a_i x^i \]
Reverse Input

Supposing the coefficients were given in reverse order \([a_0, \ldots, a_n]\). Reversing this list will be an extra \(O(n)\) time and space. Though the asymptotic complexity does not change much, it is more interesting to work directly with the given list.
\( \text{revpoly0} \)

\[
\text{revpoly0}(L, x) = \begin{cases} 
0 & \text{if } L = \text{nil} \\ 
h + x \times \text{revpoly0}(T, x) & \text{if } L = h :: T 
\end{cases}
\]

fun \text{revpoly0} ([], x) = 0.0

| \text{revpoly0} ((h::T), x) = h + x \times \text{revpoly0} (T, x)
Tail Recursive

\[ \text{revpoly1}(L, x) = \text{revpoly1}_TR(L, x, 1, x) \]

where

\[ \text{revpoly1}_TR(L, x, p, s) = \begin{cases} 
  s & \text{if } L = \text{nil} \\
  \text{revpoly1}(T, x, px, s + ph) & \text{if } L = h :: T 
\end{cases} \]
Tail Recursion: SML

local

fun revpoly1_TR([],x,p,s)=s
    | revpoly1_TR((h::T),x,p,s)=
        revpoly1_TR(T,x,p*x,s+p*h)

in

fun revpoly1 (L, x) =
    revpoly1_TR (L, x, 1.0, 0.0)

end
Complexity of \( \text{revpoly}_1 \)

Space. \( O(n) \) space to store the list

Multiplications. \( 2n \) multiplications

Additions. \( n \) additions.
Generating Primes upto $n$

**Definition 12** A positive integer $n > 1$ is **composite** iff it has a **proper divisor** $d | n$ 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 positive integer 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 \( > 1 \) is a prime iff it has no proper odd divisors.
- An odd number \( > 1 \) 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 \( \leq \lfloor \sqrt{n} \rfloor \).
\[ \text{primesUpto}(n) \]

\[ \text{primesUpto}(n) = \begin{cases} 
[] & \text{if } n < 2 \\
[(1, 2)] & \text{if } n = 2 \\
\text{primesUpto}(n - 1) & \text{elseif } \text{even}(n) \\
\text{generateFrom} \quad & \text{otherwise} \\
([(1, 2]), 3, n, 2) & 
\end{cases} \]

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

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

where \( \text{pwrt} = \text{primeWRT}(m, P) \)
Definition 13 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 $\leq \sqrt{n}$.
$\text{primeWRT}(m, P)$

**bound function** $\text{length}(P)$

**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 $P$
primeWRT

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

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>664579</td>
<td>6.65%</td>
</tr>
<tr>
<td>100000000</td>
<td>5761455</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

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\pi(n)$</th>
<th>%</th>
<th>$\lim_{n \to \infty} \frac{\pi(n)}{n/\ln n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>25</td>
<td>25.00%</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>168</td>
<td>16.80%</td>
<td>1.159</td>
</tr>
<tr>
<td>10000</td>
<td>1229</td>
<td>12.29%</td>
<td>1.132</td>
</tr>
<tr>
<td>100000</td>
<td>9592</td>
<td>9.59%</td>
<td>1.104</td>
</tr>
<tr>
<td>1000000</td>
<td>78,498</td>
<td>7.85%</td>
<td>1.084</td>
</tr>
<tr>
<td>10000000</td>
<td>664579</td>
<td>6.65%</td>
<td>1.071</td>
</tr>
<tr>
<td>100000000</td>
<td>5761455</td>
<td>5.76%</td>
<td>1.061</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>( n/2 )</td>
</tr>
<tr>
<td><code>primeWRT</code></td>
<td>( \sum_{m=3, \text{odd}(m)}^{n} \pi(m) )</td>
</tr>
</tbody>
</table>
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**.
5.2. Compound Data & Lists

1. Compound Data
2. Recap: Tuples
3. Tuple: Formation
4. Tuples: Selection
5. Tuples: Equality
6. Tuples: Equality errors
7. Lists: Recap
8. Lists: Append
9. \textit{cons vs. @} 
10. Lists of Functions
11. Lists of Functions
12. Arithmetic Sequences
13. Tail Recursion
14. Tail Recursion Invariant
15. Tail Recursion
16. Another Tail Recursion: \textit{AS3}
17. Another Tail Recursion: \textit{AS3\_iter}
18. AS3: Complexity
19. Generating Primes: 2
20. \texttt{primes2Upto(n)}
21. \texttt{generate2From(P, m, n, k)}
22. \texttt{generate2From}
23. \texttt{prime2WRT(m, P)}
24. \texttt{prime2WRT}
25. \texttt{primes2: Complexity}
26. \texttt{primes2: Diagnosis}
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,
- \( \text{val } a = ("arun", 1<2, 2); \)
  \( \text{val } a = ("arun", \text{true}, 2) \)
  : \text{string} * \text{bool} * \text{int}
- \( \text{val } b = ("arun", \text{true}, 2); \)
  \( \text{val } b = ("arun", \text{true}, 2) \)
  : \text{string} * \text{bool} * \text{int}
Tuples: Selection

- #2 a;
val it = true : bool
- #1 a;
val it = "arun" : string
- #3 a;
val it = 2 : int
- #4 a;

```plaintext
stdIn:1.1-1.5 Error: operator domain: {4:'Y; 'Z}
operand: string * bool * int
in expression:
  (fn {4=4, ...} => 4) a
```

Tuples: Equality

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

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

stdIn:1.1-29.39 Error: operator and operand don't agree [tycon mismatch]
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 don't agree [tycon mismatch]
Lists: Recap

formation: Sequence \( \alpha \) List

selection: Selection of individual components

new lists: Making new lists from old
Lists: Append

- \( \text{op @} \)
- \( \text{val it = fn : 'a list * 'a list -> 'a list} \)
- \( [1,2,3] @ [\sim 1, \sim 3]; \)
- \( \text{val it = [1,2,3,\sim 1,\sim 3]} \)
- \( : \text{int list} \)
- \( [[1,2,3],[\sim 1,\sim 2]] \)
- \( @ [[1,2,3],[\sim 1,\sim 2]] ; \)
- \( \text{val it = } \)
- \( [[1,2,3],[\sim 1,\sim 2],\)
- \( [1,2,3],[\sim 1,\sim 2]] \)
- \( : \text{int list list} \)
**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), (add3 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

\[ AS_2(a, d, n) = \begin{cases} \emptyset & \text{if } n \leq 0 \\ AS_2_{\text{iter}}(a, d, n - 1, 0, []) & \text{else} \end{cases} \]

where

for any initial \( L_0 \) and \( n \geq k \geq 0 \)

\[ INV_2 : L = L_0 @ [a] @ \ldots @ [a + (k - 1) \ast d] \]
Tail Recursion: Invariant

\[ INV_2 : L = L_0 @ [a] @ \ldots @ [a + (k - 1) \times d] \]

and bound function \( n - k \)

\[
AS_2\_iter(a, d, n, k, L) =
\begin{cases} 
L & \text{if } k \geq n \\
AS_2\_iter(a, d, n, k + 1) & \text{else} \\
L @ [a + k \times d] &
\end{cases}
\]
## Tail Recursion: Complexity

<table>
<thead>
<tr>
<th>function</th>
<th>calls</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AS^2$</td>
<td>1</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$AS^2_{\text{iter}}$</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: \( AS^3 \)

\[
AS^3(a, d, n) =
\begin{cases} 
[] & \text{if } n \leq 0 \\
AS^3_{\text{iter}}(a, d, n - 1, []) & \text{else}
\end{cases}
\]

where

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

\[
INV^3: L = (a + (n - 1) \times d) :: \cdots :: \cdots :: (a + (n_0 - 1) \times d) :: L_0
\]
Another Tail Recursion:

\[ \text{INV}_3 : L = (a + (n - 1) \times d) :: \cdots \]

\[ \cdots :: (a + (n_0 - 1) \times d) :: L_0 \]

and bound function \( n \),

\[
\text{AS3}_{\text{iter}}(a, d, n, L) =
\begin{cases} 
L & \text{if } n \leq 0 \\
\text{AS3}_{\text{iter}}(a, d, n - 1, \text{else} \ (a + (n - 1) \times d) :: L) 
\end{cases}
\]
### 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

composites

- \textit{primesUpto}
- invariant
- \textit{generateFrom}
primes2Upto(n) =

\[
\begin{cases}
\emptyset & \text{if } n < 2 \\
[(1, 2)] & \text{if } n = 2 \\
\text{primes2Upto}(n - 1) & \text{elseif } \text{even}(n) \\
\text{generate2From} & \text{otherwise} \\
\left([(1, 2)], 3, n, 2\right)
\end{cases}
\]

where
\( \text{generate2From}(P, m, n, k) \)

**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

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

where \( pwrt = \text{prime2WRT}(m, P) \)
\text{prime2WRT}(m, P)

\textbf{bound function} \text{length}(P)

\textbf{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 \( p_j \leq \lfloor \sqrt{m} \rfloor < p_{j+1} \)
$prime2WRT$

$prime2WRT(m, P) =$

\[
\begin{cases}
  \text{true} & \text{if } P = \text{nil} \\
  \text{true} & \text{if } h > m \div h \\
  \text{false} & \text{elseif } h \mid m \\
  prime2WRT & \text{else}
\end{cases}
\]

$(m, \text{tl}(P))$

where

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

for some $i > 0$
**primes2**: Complexity

<table>
<thead>
<tr>
<th>function</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>primes2Upto</td>
<td>1</td>
</tr>
<tr>
<td>generate2From</td>
<td>(\frac{n}{2})</td>
</tr>
<tr>
<td>prime2WithRT</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)?
5.3. Compound Data & List Algorithms

1. Compound Data: Summary
2. Records: Constructors
3. Records: Example 1
4. Records: Example 2
5. Records: Destructors
6. Records: Equality
7. Tuples & Records
8. Back to Lists
9. Lists: Correctness
10. Lists: Case Analysis
11. Lists: Correctness by Cases
12. List-functions: \textit{length}
13. List Functions: \textit{search}
14. List Functions: \textit{search2}
15. List Functions: \textit{ordered}
16. List Functions: \textit{insert}
17. List Functions: \textit{reverse}
18. List Functions: reverse
19. List Functions: merge
20. List Functions: merge
22. ML: merge
23. Sorting by Insertion
24. Sorting by Merging
25. Sorting by Merging
26. Functions as Data
27. Higher Order Functions
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: Example 1

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: Example 2

- val billu = 
  { age = 1, 
    name = "Billu",
    favColour = "blue"
  };
- val billu = 
  {age=1,favColour="blue",name="Billu"}:
  {age:int, favColour: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"}
{age:int, fav_colour:string, name:string}
- 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 $L : \alpha \text{List}$ satisfies

\[
L = []
\]

**XOR**

\[
L = \text{hd}(L) :: \text{tl}(L)
\]

- 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.

fun f [] = c
  | f (h::T) = g (h, T)
;
Lists: Correctness by Cases

Lists-correctness

$P$ is proved by case analysis.

**Basis** Prove

\[ P([]) \]

**Induction hypothesis (IH)** Assume

\[ P(T) \]

**Induction Step** Prove

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

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

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

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

Or even more conveniently

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

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

**Definition 14** A list \( L = [a_0, \ldots, a_{n-1}] \) is \textit{ordered} by a relation \( \leq \) if consecutive elements are related by \( \leq \), i.e \( a_i \leq a_{i+1} \), for \( 0 \leq i < n - 1 \).

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

Time Complexity??
List Functions: $\text{insert}$

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

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

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

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

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: $merge$

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

where

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

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

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

Time Complexity??
Sorting by Merging

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

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.
6. Higher Order Functions & Structured Data

6.1. Higher Order Functions

1. Summary: Compound Data
2. List: Examples
3. Lists: Sorting
4. Higher Order Functions
5. An Example
6. Currying
7. Currying: Contd
8. Generalization
9. Generalization: 2
10. Applying a list
11. Trying it out
12. Associativity
13. Apply to a list
14. Sequences
15. Further Generalization
16. Further Generalization
17. Sequences
18. Efficient Generalization
19. Sequences: 2
20. More Generalizations
21. More Summations
22. Or Maybe . . . Products
23. Or Some Other \( \otimes \)
24. Other \( \otimes \)
25. Examples of \( \otimes, e \)
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 Example

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:

```ml
val addc = fn :
  int -> (int -> int)

Contrast with ML’s response

- op +;
val it = fn : int * int -> int

\text{addc} \text{ is the curried version of the binary operation } +.\]
Currying: Contd

\[ f : (\alpha \ast \beta \ast \gamma) \rightarrow \delta \checkmark \]

\[ f_c : \alpha \rightarrow \beta \rightarrow \gamma \rightarrow \delta \checkmark \]

\[ f_c' : (\alpha \ast \beta) \rightarrow \gamma \rightarrow \delta \checkmark \]

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

Then

- \( \text{addc1} = (\text{addc 1}): \text{int} \to \text{int} \)
- \( \text{addc2} = (\text{addc 2}): \text{int} \to \text{int} \)
- \( \text{addc3} = (\text{addc 3}): \text{int} \to \text{int} \)

and for any \( i \),

\( (\text{addc } i): \text{int} \to \text{int} \)

is the required function.
Generalization: 2

\[
\text{list}\_\text{adds} \ n = \begin{cases} \
\emptyset & \text{if } n \leq 0 \\
(\text{list}\_\text{adds}(n - 1)) \circ (\text{addc} \ n) & \text{else}
\end{cases}
\]

ML’s response:

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

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

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:

\[
\text{val interval} = \text{fn}: \int \to \int \to \int \text{ list} \\
- \text{interval } 53 \ 5; \\
\text{val it} = [54, 55, 56, 57, 58] : \int \text{ list}
\]
Associtativity

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

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

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

Apply a list Transpose of a matrix

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

val it = fn : ('a -> 'b) -> \\
' 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

\[ AS4(a, d, n) = \begin{cases} \text{if } n \leq 0 \\ a :: (\text{map} (\text{addc } d) \quad \text{else} \\
(AS4(a, d, (n - 1))) \end{cases} \]
Further Generalization

Given

\[ f : \alpha \times \alpha \rightarrow \alpha \]

Then

\[ curry2 f \; x \; y = f(x, y) \]

and

\[ (curry2 f) : \alpha \rightarrow (\alpha \rightarrow \alpha) \]

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

\[ ((curry2 f) \; d) : \alpha \rightarrow \alpha \]
Further Generalization

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

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

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

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

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

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

Let’s not use \texttt{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

- \( AS_6(a, d, n) = seq_2(id, op+, a, d, n) \)
- \( GS_2(a, r, n) = seq_2(id, op*, a, r, n) \)
- \( HS_2(a, d, n) = seq_2(reci, op+, a, d, n) \)

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 upto some $n$-th term
- (Truncated) Taylor and Maclaurin series
More Summations

Wasteful to first generate the sequence and then compute the measure

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

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

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

Or some other binary operation \( \odot \) which has the following properties:

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

\[
\begin{array}{c}
  \bigotimes \\
  f(l) \\
  i=l
\end{array}
\]
Then if \( f, \text{succ} : \alpha \rightarrow \alpha \)

\[
\text{ser}(\bigotimes, f, \text{succ}, l, u) = \begin{cases} 
  e & \text{if } [l, u] = \emptyset \\
  f(l) \otimes \text{ser}(\bigotimes, f, \text{succ}, \text{succ}(l), u) & \text{else}
\end{cases}
\]
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
- $\ast$, 1 on vectors and matrices
6.2. Structured Data

1. Transpose of a Matrix
2. Transpose: 0
3. Transpose: 10
4. Transpose: 01
5. Transpose: 20
6. Transpose: 02
7. Transpose: 30
8. Transpose: 03
9. *trans*
10. *is2DMatrix*
11. User Defined Types
12. Enumeration Types
13. User Defined Structural Types
14. Functions vs. data
15. Data as 0-ary Functions
16. Data vs. Functions
17. Data vs. Functions: Recursion
18. Lists
19. Constructors
20. Shapes
21. Shapes: Triangle Inequality
22. Shapes: Area
23. Shapes: Area
24. ML: Try out
25. ML: Try out (contd.)
26. Enumeration Types
27. Recursive Data Types
28. Resistors: Datatype
29. Resistors: Equivalent
30. Resistors
31. Resistors: Example
32. Resistors: ML session
Transpose of a Matrix

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

\[
\text{transpose } L = \begin{cases} 
\text{trans } L & \text{if } \text{is2DMatrix}(L) \\
\perp & \text{else}
\end{cases}
\]

where
### Transpose: 0

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

\[
\begin{bmatrix}
\end{bmatrix}
\]
Transpose: 10

\[
\begin{bmatrix}
11 & 12 & 13 \\
21 & 22 & 23 \\
31 & 32 & 33 \\
41 & 42 & 43 \\
\end{bmatrix}
\]
### Transpose: 01

\[
\begin{bmatrix}
12 & 13 \\
22 & 23 \\
32 & 33 \\
42 & 43 \\
\end{bmatrix}
\quad \rightarrow \quad
\begin{bmatrix}
11 & 21 & 31 & 41 \\
\end{bmatrix}
\]
Transpose: 20

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

\[
\begin{bmatrix}
11 & 21 & 31 & 41 \\
\end{bmatrix}
\]
Transpose: 02

\[
\begin{bmatrix}
  [ \\
  13 ] \\
  [ \\
  23 ] \\
  [ \\
  33 ] \\
  [ \\
  43 ] \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
  [ 11 21 31 41 ] \\
  [ 12 22 32 42 ] \\
\end{bmatrix}
\]
### Transpose: 30

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

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

\[
\begin{bmatrix}
11 & 21 & 31 & 41 \\
12 & 22 & 32 & 42 \\
13 & 23 & 33 & 43
\end{bmatrix}
\]
\begin{equation*}
\begin{aligned}
\text{trans} & \quad \text{is 2DMatrix} = \#1(\text{dimensions } L) \\
\text{trans } [] &= [] \\
\text{trans } [] :: TL &= [] \\
\text{trans } LL &= (\text{map hd } LL) :: (\text{trans } (\text{map tl } LL)) \\
\end{aligned}
\end{equation*}

and

\begin{equation*}
\text{is2DMatrix} = \#1(\text{dimensions } L)
\end{equation*}

where
is2DMatrix

\[
\begin{align*}
\text{dimensions } [] & = (\text{true}, 0, 0) \\
\text{dimensions } [H] & = (\text{true}, 1, h) \\
\text{dimensions } (H :: TL) & = (b \text{ and } (h = c), r + 1, c)
\end{align*}
\]

where \( \text{dimensions } TL = (b, r, c) \)
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 as Structured Data

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

Every \( \alpha \) list is either

- \( \text{nil} : (\text{Basis, name}) \)
- \( : \text{or (alternative)} \)
- \( \text{cons} : \text{constructed inductively from an element of type } \alpha \text{ and another list of type } \alpha \text{ list using the constructor}\) 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

**Basis**: \( \text{nil} \)

**Induction**: \( \text{cons} \)
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

```haskell
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

```plaintext
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;
Resistors

5.0

4.0

5.0

2.0

3.0

+

−
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): resist

val it = (): unit

- value R;

val it = 2.26363636364: real

-
### 6.3. User Defined Structured Data Types

1. User Defined Types
2. Resistors: Grouping
3. Resistors: In Pairs
4. Resistor: Values
5. Resistance Expressions
6. Resistance Expressions
7. Arithmetic Expressions
8. Arithmetic Expressions: 0
9. Arithmetic Expressions: 1
10. Arithmetic Expressions: 2
11. Arithmetic Expressions: 3
12. Arithmetic Expressions: 4
13. Arithmetic Expressions: 5
15. Arithmetic Expressions: 7
16. Arithmetic Expressions: 8
17. Binary Trees
18. Arithmetic Expressions: 0
19. Trees: Traversals
20. Recursive Data Types: Correctness
21. Data Types: Correctness
User Defined Types

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

R1

R2

R3

R4

5.0
4.0
5.0 2.0
3.0
+ −
R1
R2
R3
R4
Resistors: In Pairs

val R1 = PAR (RES 5.0, RES 4.0)
val R2 = SER (RES 5.0, RES 2.0)
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:

\[((5 \times \sim 4) + \sim(5 - 2)) \div 3\]

are represented as trees
Arithmetic Expressions: 1

\[
\begin{align*}
\text{div} & \quad + \\
\ast & \quad \sim \\
5 & \quad 4 \\
\sim & \quad 5 \\
\sim & \quad 2 \\
\end{align*}
\]
Arithmetic Expressions: 2

```
5 * -4 + 3  div 3
<table>
<thead>
<tr>
<th></th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>
```

```
Arithmetic Expressions: 3

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

\[ \frac{-20}{3} + \frac{-3}{3} \]
Arithmetic Expressions: 5

\[
\text{div} \quad \begin{array}{c}
+ \\
-20 \quad -3
\end{array} 3
\]
Arithmetic Expressions: 6

\[ \frac{-23}{3} \]
Arithmetic Expressions: 7

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

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

Arithmetic Expressions

```
5 ~ 4
   ~ 2
   ~ 5
   + 3
   div
```

5 4 2 3
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 $r$
7. Imperative Programming: An Introduction

7.1. Introducing a Memory Model

1. Summary: Functional Model
2. CPU & Memory: Simplified
3. Resource Management
4. Shell: User Interface
5. GUI: User Interface
6. Memory Model: Simplified
7. Memory
8. The Imperative Model
9. State Changes: $\sigma$
10. State
11. State Changes
12. State Changes: $\sigma$
13. State Changes: $\sigma_1$
14. State Changes: $\sigma_2$
15. Languages
16. User Programs
17. Imperative Languages
18. Imperative vs Functional Variables
19. Assignment Commands
20. Assignment Commands
21. Assignment Commands
22. Assignment Commands
23. Assignment Commands
24. Assignment Commands: Swap
25. Swap
26. Swap
27. Swap
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
Resource Management

Operating System

CPU

Memory

Peripherals

Printer
Disk
Keyboard
Screen
Shell: User Interface

- CPU
- Memory
- Peripherals: Printer, Disk, Keyboard, Screen
- Operating System
- Shell
GUI: User Interface

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.
  - \textit{state} : Addresses $\rightarrow$ Values
- State changes
State Changes: $\sigma$

Assume all other cells are filled with `null`
State

The state $\sigma$

- $\sigma(12) = 4 : \text{int}$
- $\sigma(20) = \text{null}$
- $\sigma(43) = \text{true : bool}$
- $\sigma(66) = \text{” #a” : char}$
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>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>true</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: $\sigma_1$

<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>0</td>
<td>1</td>
<td>2</td>
<td>3</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>

- $\text{true}$
- "#a"
- Assume all other cells are filled with `null`
State Changes: $\sigma_2$

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.1</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>&quot;#a&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Assume all other cells are filled with null.
Languages

CPU

Memory

Peripherals

Printer
Disk
Keyboard
Screen

Operating System

Programming Language Interface
User Programs

Operating System

Programming Language Interface

User Programs
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 constant</td>
<td>name of an address 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 $!y = 2$. 

\[
\begin{array}{ccc}
& x & 1 \\
& y & 2 \\
\end{array}
\]
Assignment Commands

Store the value 5 in $x$.

$x := 5$

x 5  y 2
Assignment Commands

Copy the value contained in $y$ into $x$.

\[ x := !y \]
Assignment Commands

Increment the value contained in $x$ by 1.

$x := x + 1$

\[
\begin{array}{c|c|c}
\text{x} & \text{3} & \text{y} & \text{2} \\
\end{array}
\]
Assignment Commands

Store the product of the values in $x$ and $y$ in $y$.

$$y := !x \ast !y$$
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?

\[ \textit{val } t = \textit{ref } 0 \]

Then the rest is easy

\[
\begin{align*}
\text{val } t &= \text{ref } 0; \\
t &= !x; \\
x &= !y; \\
y &= !t;
\end{align*}
\]
Swap

Could be made simpler!

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;
```
7.2. Imperative Programming:

1. Imperative vs Functional
2. Features of the Store
3. References: Experiments
4. References: Experiments
5. References: Experiments
6. Aliases
7. References: Experiments
8. References: Aliases
9. References: Experiments
10. After Garbage Collection
11. Side Effects
12. Imperative ML
13. Imperative ML
14. Imperative ML
15. Imperative ML
16. Nasty Surprises
17. Imperative ML
18. Imperative ML
19. Common Errors
20. Aliasing & References
21. Dangling References
22. New Reference
23. Imperative Commands: Basic
24. Imperative Commands: Compound
25. Predefined Compound Commands
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
- val b = ref 0;
val b = ref 0 : int ref
References: Experiments

- \( a = b \);
- \( \neg a = \neg b \);
- \( \text{val it = false : bool} \)
- \( \text{val it = true : bool} \)
Aliases

- val x = ref 0;
val x = ref 0 : int ref
References: Experiments

- val y = x;
val y = ref 0 : int ref
References: Aliases

\[-x := !x + 1;\]
\[val \ it = () : \text{unit}\]
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:
Dangling References

Deallocate $x$ through a system call

$y$ is left dangling!
By sheer coincidence \( y = 12 \)
Imperative Commands: Basic

A **Command** is an ML expression that creates a **side effect** and returns an empty tuple `() : 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** \( \text{if } e \text{ then } c_1 \text{ else } c_0 \).

**cases** \( \text{case } e \text{ of } p_1 \Rightarrow c_1 | \cdots | p_n \Rightarrow c_n \)

**Sequencing** \( (c_1; c_2; \ldots; c_n) \). Sequencing is associative

**looping** \( \text{while } e \text{ do } c_1 \) is defined recursively as

\( \text{if } e \text{ then } (c_1; \text{while } e \text{ do } c_1) \text{ else ()} \)
7.3. Arrays

1. Why Imperative
2. Arrays
3. Indexing Arrays
4. Indexing Arrays
5. Indexing Arrays
6. Physical Addressing
7. Arrays
8. 2D Arrays
9. 2D Arrays: Indexing
10. Ordering of indices
11. Arrays vs. Lists
12. Arrays: Physical
13. Lists: Physical
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

\[ a_0 \rightarrow C_0 \rightarrow C_i \rightarrow C_{n-1} \rightarrow a_{n-1} \rightarrow a_i \]
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 $rc$

$C_{0,0}, C_{0,1}, \ldots, C_{0,c-1}, C_{1,0}, \ldots, C_{r-1,c-1}$
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

\[(i, j) < (k, l) \iff (i < k) \text{ or } (i = k \text{ and } j < l)\]
# 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>
Arrays: Physical

\[ a_0, a_{n-1}, a_i \]
Lists: Physical
8. A large Example: Tautology Checking

8.1. Large Example: Tautology Checking

1. Logical Arguments
2. Saintly and Rich
3. About Cats
4. About God
5. Russell’s Argument
6. Russell’s Argument
7. Russell’s Argument
8. Russell’s Argument
9. Propositions
10. Compound Propositions
11. Valuations
12. Valuations
13. Tautology
14. Properties
15. Negation Normal Form
16. Literals & Clauses
17. Conjunctive Normal Form
18. Validity
19. Logical Validity
20. Validity & Tautologies
21. Problem
22. Tautology Checking
23. Falsifying
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>true</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>
</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 ¬, ∧ and ∨

\[(p \iff q) = (p\implies q) \land (q\implies p)\]

\[(p\implies q) = (\neg p \lor q)\]

• 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 \( \neg \) applied to an atom.
- \( \lor \) is commutative and associative.
- A clause is of the form \( \lor_{j=1}^{m} l_j \), 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 \( \bigwedge_{i=1}^{n} q_i \), 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 \([h_1, \ldots, h_n] | c\),

- determine whether \((h_1 \land \cdots \land h_n) \Rightarrow c\)
  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
8.2. Tautology Checking Contd.

1. Tautology Checking
2. Normal Forms
3. Top-down Development
4. The Signature
5. The Core subproblem
6. The datatype
7. Convert to CNF
8. Rewrite into NNF
9. $\leftrightarrow$ and $\Rightarrow$ Elimination
10. Push $\neg$ inward
11. Push $\neg$ inward
12. conj.of.disj
13. Push $\lor$ inward
14. Tautology & Falsification
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
...
end;
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

- *rewrite* eliminates $\iff$ and $\Rightarrow$
- *nnf* converts into NNF
- *conj_of_disj* converts into CNF
Rewrite into NNF

• Eliminate $\equiv$ and then $\implies$

• Push $\neg$ inward using De Morgan’s laws and eliminate double negations.
and ⇒ Elimination

\[
\begin{align*}
\text{fun rewrite (ATOM a) } &= \text{ATOM a} \\
&\mid \text{ rewrite (IMP (P, Q)) } = \text{OR (NOT (rewrite(P)), rewrite(Q))} \\
&\mid \text{ rewrite (EQL (P, Q)) } = \text{rewrite (AND (IMP(P, Q), IMP(Q, P)))} \\
&\mid \ldots
\end{align*}
\]

Proposition made up of only $\neg$, $\land$ and $\lor$. 
Push \rightarrow {\text{inward}}

\text{fun} \ \text{nff} (\text{ATOM} \ a) = \\
\text{ATOM} \ a \\
| \ \text{nff} (\text{NOT} (\text{ATOM} \ a)) = \\
\text{NOT} (\text{ATOM} \ a) \\
| \ \text{nff} (\text{NOT} (\text{NOT} (P))) = \\
\text{nff} (P)
Push \( \rightarrow \) inward

| \( \text{nff} (\text{NOT} (\text{AND} (P, Q))) = \text{nff} (\text{OR} (\text{NOT} (P), \text{NOT} (Q))) \) |
| \( \text{nff} (\text{NOT} (\text{OR} (P, Q))) = \text{nff} (\text{AND} (\text{NOT} (P), \text{NOT} (Q))) \) |
| ... |

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 $\land$

fun distOR (P, AND (Q, R)) = AND (distOR (P, Q),
                  distOR (P, R))

| distOR (AND (Q, R), P) = AND (distOR (Q, P),
                  distOR (R, P))

| 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!
9. Lecture-wise Index to Slides

Index

Introduction to Computing (3-8)

1. Introduction
2. Computing tools
3. Ruler and Compass
4. Computing and Computers
5. Primitives
6. Algorithm
7. Problem: Doubling a Square
8. Solution: Doubling a Square
9. Execution: Step 1
10. Execution: Step 2
11. Doubling a Square: Justified
12. Refinement: Square Construction
13. Refinement 2: Perpendicular at a point
14. Solution: Perpendicular at a point
15. Perpendicular at a point: Justification
Our Computing Tool

1. The Digital Computer: Our Computing Tool
2. Algorithms
3. Programming Language
4. Programs and Languages
5. Programs
6. Programming
7. Computing Models
8. Primitives
9. Primitive expressions
10. Methods of combination
11. Methods of abstraction
12. The Functional Model
13. Mathematical Notation 1: Factorial
14. Mathematical Notation 2: Factorial
15. Mathematical Notation 3: Factorial
16. A Functional Program: Factorial
17. A Computation: Factorial
18. A Computation: Factorial
19. A Computation: Factorial
20. A Computation: Factorial
22. A Computation: Factorial
23. A Computation: Factorial
24. Standard ML
25. SML: Important Features

Next: Primitives: Integer & Real

Primitives: Integer & Real (34-60)

Previous: Primitives: Integer & Real

1. Algorithms & Programs
2. SML: Primitive Integer Operations 1
3. SML: Primitive Integer Operations 1
4. SML: Primitive Integer Operations 1
5. SML: Primitive Integer Operations 1
6. SML: Primitive Integer Operations 1
7. SML: Primitive Integer Operations 1
8. SML: Primitive Integer Operations 2
9. SML: Primitive Integer Operations 2
10. SML: Primitive Integer Operations 2
11. SML: Primitive Integer Operations 2
12. SML: Primitive Integer Operations 2
13. SML: Primitive Integer Operations 3
14. SML: Primitive Integer Operations 3
15. SML: Primitive Integer Operations 3
16. SML: Primitive Integer Operations 3
17. SML: Primitive Integer Operations 3
18. Quotient & Remainder
19. SML: Primitive Real Operations 1
20. SML: Primitive Real Operations 1
21. SML: Primitive Real Operations 1
22. SML: Primitive Real Operations 2
23. SML: Primitive Real Operations 3
24. SML: Primitive Real Operations 4
25. SML: Precision
26. Fibonacci Numbers
27. Euclidean Algorithm

Next: Technical Completeness & Algorithms
1. Recapitulation: Integers & Real
2. Recap: Integer Operations
3. Recapitulation: Real Operations
4. Recapitulation: Simple Algorithms
5. More Algorithms
6. Powering: Math
7. Powering: SML
8. Technical completeness
9. What SML says
10. Technical completeness
11. What SML says ... contd
12. Powering: Math 1
13. Powering: SML 1
14. Technical Completeness
15. What SML says
16. Powering: Integer Version
17. Exceptions: A new primitive
18. Integer Power: SML
19. Integer Square Root 1
20. Integer Square Root 2
21. An analysis
22. Algorithmic idea
23. Algorithm: isqrt
24. Algorithm: shrink
25. SML: shrink
26. SML: intsqt
27. Run it!
28. SML: Reorganizing Code
29. Intsqrt: Reorganized
30. shrink: Another algorithm
31. Shrink2: SML
32. Shrink2: SML ... contd

Algorithm Refinement (93-122)

1. Recap: More Algorithms
2. Recap: Power
3. Recap: Technical completeness
4. Recap: More Algorithms
5. Intsqrt: Reorganized
6. Intsqrt: Reorganized
7. Some More Algorithms
8. Combinations: Math
9. Combinations: Details
10. Combinations: SML
11. Perfect Numbers
12. Refinement
13. Perfect Numbers: SML
14. $\sum_i^n \text{ifdivisor}(k)$
15. SML: sum_divisors
16. \text{ifdivisor} and \text{ifdivisor}
17. SML: Assembly 1
18. SML: Assembly 2
19. Perfect Numbers \ldots contd.
20. Perfect Numbers \ldots contd.
21. SML: Assembly 3
22. Perfect Numbers: Run
23. Perfect Numbers: Run
24. SML: Code variations
25. SML: Code variations
26. SML: Code variations
27. Summation: Generalizations
28. Algorithmic Improvements:
29. Algorithmic Variations
30. Algorithmic Variations

Variations: Algorithms & Code (123-149)
1. Recap
2. Recap: Combinations
3. Combinations 1
4. Combinations 2
5. Combinations 3
6. Perfect 2
7. Power 2
8. A Faster Power
9. Power2: Complete
10. Power2: SML
11. Power2: SML
12. Computation: Issues
13. General Correctness
14. Code: Justification
15. Recall
16. Features: Definition before Use
17. Run ifdivisor
18. Diagnosis: Features of programs
20. Incorrectness
21. ifdivisor3
22. Run it!
23. Try it!
24. Hey! Wait a minute!
25. The n’s
26. Scope
27. Scope Rules

Names, Scopes & Recursion (150-181)

1. Disjoint Scopes
2. Nested Scopes
3. Overlapping Scopes
4. Spannning
5. Scope & Names
6. Names & References
7. Names & References
8. Names & References
9. Names & References
10. Names & References
11. Names & References
12. Names & References
13. Names & References
14. Names & References
15. Names & References
16. Definition of Names
17. Use of Names
18. local...in...end
19. local...in...end
20. local...in...end
21. local...in...end
22. Scope & local
23. Computations: Simple
24. Simple computations
25. Computations: Composition
26. Composition: Alternative
27. Compositions: Compare
28. Compositions: Compare
29. Computations: Composition
30. Recursion
31. Recursion: Left
32. Recursion: Right

Floating Point (182-201)

1. So Far-1: Computing
2. So Far-2: Algorithms & Programs
3. So far-3: Top-down Design
4. So Far-4: Algorithms to Programs
5. So far-5: Caveats
6. So Far-6: Algorithmic Variations
7. So Far-7: Computations
8. Floating Point
9. Real Operations
10. Real Arithmetic
11. Numerical Methods
12. Errors
13. Errors
14. Infinite Series
15. Truncation Errors
16. Equation Solving
17. Root Finding-1
18. Root Finding-2
19. Root Finding-3
20. Root Finding-4

Root Finding, Composition and Recursion (202-229)

1. Root Finding: Newton’s Method
2. Root Finding: Newton’s Method
3. Root Finding: Newton’s Method
4. Root Finding: Newton’s Method
5. Root Finding: Newton’s Method
6. Root Finding: Newton’s Method
7. Newton’s Method: Basis
8. Newton’s Method: Basis
10. What can go wrong!-1
11. What can go wrong!-2
12. What can go wrong!-2
13. What can go wrong!-3
14. What can go wrong!-4
15. Real Computations & Induction: 1
16. Real Computations & Induction: 2
17. What’s it good for? 1
18. What’s it good for? 2
19. newton: Computation
20. Generalized Composition
21. Two Computations of \( h(1) \)
22. Two Computations of \( h(-1) \)
23. Recursive Computations
24. Recursion: Left
25. Recursion: Right
26. Recursion: Nonlinear
27. Some Practical Questions
28. Some Practical Questions

Termination and Space Complexity  (230-266)

1. Recursion Revisited
2. Linear Recursion: Waxing
3. Recursion: Waning
4. Nonlinear Recursions
5. Fibonacci: contd
6. Recursion: Waxing & Waning
7. Unfolding Recursion
8. Non-termination
9. Termination
10. Proofs of termination
11. Proofs of termination: Induction
12. Proof of termination: Factorial
13. Proof of termination: Factorial
14. Fibonacci: Termination
15. GCD computations
16. Well-foundedness: GCD
17. Well-foundedness
18. Induction is Well-founded
19. Induction is Well-founded
20. Where it doesn’t work
21. Well-foundedness is inductive
22. Well-foundedness is inductive
23. GCD: Well-foundedness
24. Newton: Well-foundedness
25. Newton: Well-foundedness
26. Example: Zero
27. Questions
28. The Collatz Problem
29. Questions
30. Space Complexity
31. Newton & Euclid: Absolute
32. Newton & Euclid: Relative
33. Deriving space requirements
34. GCD: Space
35. Factorial: Space
36. Fibonacci: Space
37. Fibonacci: Space

Efficiency Measures and Speed-ups (267-295)
1. Recapitulation
2. Recapitulation
3. Time & Space Complexity
4. \textit{isqrt}: Space
5. Time Complexity
6. \textit{isqrt}: Time Complexity
7. \textit{isqrt2}: Time
8. \textit{shrink} \textit{vs. shrink2}: Times
9. Factorial: Time Complexity
10. Fibonacci: Time Complexity
11. Comparative Complexity
12. Comparisons
13. Comparisons
14. Efficiency Measures: Time
15. Efficiency Measures: Space
16. Speeding Up: 1
17. Speeding Up: 2
18. Factoring out calculations
19. Tail Recursion: 1
20. Tail Recursion: 2
21. Factorial: Tail Recursion
22. Factorial: Tail Recursion
23. A Computation
24. Factorial: Issues
25. Fibonacci: Tail Recursion
26. Fibonacci: Tail Recursion
27. fibTR: SML
28. State in Tail Recursion
29. Invariance

Invariance & Correctness (296-319)

1. Recap
2. Recursion Transformation
3. Tail Recursion: Examples
4. Comparisons
5. Transformation Issues
6. Correctness Issues 1
7. Correctness Issues 2
8. Correctness Theorem
9. Invariants & Correctness 1
10. Invariants & Correctness 2
11. Invariance Lemma: factL_tr
12. Invariance: Example
13. Invariance: Example
14. Proof
15. Invariance Lemma: $fib_{iter}$
16. Proof
17. Correctness: Fibonacci
18. Variants & Invariants
19. Variants & Invariants
20. More Invariants
21. Fast Powering 1
22. Fast Powering 2
23. Root Finding: Bisection
24. Advantage Bisection

Tuples, Lists & the Generation of Primes (320-341)

1. Recap: Tail Recursion
2. Examples: Invariants
3. Tuples
4. Lists
5. New Lists
6. List Operations
7. List Operations: $cons$
8. Generating Primes upto $n$
9. More Properties
10. Composites
11. Odd Primes
12. $primesUpto(n)$
13. $generateFrom(P, m, n, k)$
14. $generateFrom$
15. $primeWRT(m, P)$
16. $primeWRT(m, P)$
17. $primeWRT$
18. Density of Primes
19. The Prime Number Theorem
20. The Prime Number Theorem
21. Complexity
22. Diagnosis

Compound Data & Lists

1. Compound Data
2. Recap: Tuples
3. Tuple: Formation
4. Tuples: Selection
5. Tuples: Equality
6. Tuples: Equality errors
7. Lists: Recap

(342-367)
8. Lists: Append
9. *cons vs. @*
10. Lists of Functions
11. Lists of Functions
12. Arithmetic Sequences
13. Tail Recursion
14. Tail Recursion Invariant
15. Tail Recursion
16. Another Tail Recursion: *AS3*
17. Another Tail Recursion: *AS3_iter*
18. *AS3*: Complexity
19. Generating Primes: 2
20. *primes2Upto(n)*
21. *generate2From(P, m, n, k)*
22. *generate2From*
23. *prime2WRT(m, P)*
24. *prime2WRT*
25. *primes2*: Complexity
26. *primes2*: Diagnosis

**Compound Data & List Algorithms**

1. Compound Data: Summary

(368-394)
2. Records: Constructors
3. Records: Example 1
4. Records: Example 2
5. Records: Destructors
6. Records: Equality
7. Tuples & Records
8. Back to Lists
9. Lists: Correctness
10. Lists: Case Analysis
11. Lists: Correctness by Cases
12. List-functions: length
13. List Functions: search
14. List Functions: search2
15. List Functions: ordered
16. List Functions: insert
17. List Functions: reverse
18. List Functions: reverse2
19. List Functions: merge
20. List Functions: merge
22. ML: merge
23. Sorting by Insertion
24. Sorting by Merging
25. Sorting by Merging
26. Functions as Data
27. Higher Order Functions

Higher Order Functions

1. Summary: Compound Data
2. List: Examples
3. Lists: Sorting
4. Higher Order Functions
5. An Example
6. Currying
7. Currying: Contd
8. Generalization
9. Generalization: 2
10. Applying a list
11. Trying it out
12. Associativity
13. Apply to a list
14. Sequences
15. Further Generalization
16. Further Generalization
17. Sequences
18. Efficient Generalization
19. Sequences: 2
20. More Generalizations
21. More Summations
22. Or Maybe . . . Products
23. Or Some Other $\otimes$
24. Other $\otimes$
25. Examples of $\otimes$, $e$

Structured Data (420-451)

1. Transpose of a Matrix
2. Transpose: 0
3. Transpose: 10
4. Transpose: 01
5. Transpose: 20
6. Transpose: 02
7. Transpose: 30
8. Transpose: 03
9. $trans$
10. $is2DMatrix$
11. User Defined Types
12. Enumeration Types
13. User Defined Structural Types
14. Functions vs. data
15. Data as 0-ary Functions
16. Data vs. Functions
17. Data vs. Functions: Recursion
18. Lists
19. Constructors
20. Shapes
21. Shapes: Triangle Inequality
22. Shapes: Area
23. Shapes: Area
24. ML: Try out
25. ML: Try out (contd.)
26. Enumeration Types
27. Recursive Data Types
28. Resistors: Datatype
29. Resistors: Equivalent
30. Resistors
31. Resistors: Example
32. Resistors: ML session
User Defined Structured Data Types  (452-472)

1. User Defined Types
2. Resistors: Grouping
3. Resistors: In Pairs
4. Resistor: Values
5. Resistance Expressions
6. Resistance Expressions
7. Arithmetic Expressions
8. Arithmetic Expressions: 0
9. Arithmetic Expressions: 1
10. Arithmetic Expressions: 2
11. Arithmetic Expressions: 3
12. Arithmetic Expressions: 4
13. Arithmetic Expressions: 5
15. Arithmetic Expressions: 7
16. Arithmetic Expressions: 8
17. Binary Trees
18. Arithmetic Expressions: 0
19. Trees: Traversals
20. Recursive Data Types: Correctness
21. Data Types: Correctness

Introducing a Memory Model (473-499)

1. Summary: Functional Model
2. CPU & Memory: Simplified
3. Resource Management
4. Shell: User Interface
5. GUI: User Interface
6. Memory Model: Simplified
7. Memory
8. The Imperative Model
9. State Changes: $\sigma$
10. State
11. State Changes
12. State Changes: $\sigma$
13. State Changes: $\sigma_1$
14. State Changes: $\sigma_2$
15. Languages
16. User Programs
17. Imperative Languages
18. Imperative vs Functional Variables
19. Assignment Commands
20. Assignment Commands
21. Assignment Commands
22. Assignment Commands
23. Assignment Commands
24. Assignment Commands: Swap
25. Swap
26. Swap
27. Swap

Imperative Programming: (500-524)

1. Imperative vs Functional
2. Features of the Store
3. References: Experiments
4. References: Experiments
5. References: Experiments
6. Aliases
7. References: Experiments
8. References: Aliases
9. References: Experiments
10. After Garbage Collection
11. Side Effects
12. Imperative ML
13. Imperative ML
14. Imperative ML
15. Imperative ML
16. Nasty Surprises
17. Imperative ML
18. Imperative ML
19. Common Errors
20. Aliasing & References
21. Dangling References
22. New Reference
23. Imperative Commands: Basic
24. Imperative Commands: Compound
25. Predefined Compound Commands

Arrays (525-537)

1. Why Imperative
2. Arrays
3. Indexing Arrays
4. Indexing Arrays
5. Indexing Arrays
6. Physical Addressing
7. Arrays
8. 2D Arrays
9. 2D Arrays: Indexing
10. Ordering of indices
11. Arrays vs. Lists
12. Arrays: Physical
13. Lists: Physical

Large Example: Tautology Checking

1. Logical Arguments
2. Saintly and Rich
3. About Cats
4. About God
5. Russell’s Argument
6. Russell’s Argument
7. Russell’s Argument
8. Russell’s Argument
9. Propositions
10. Compound Propositions
11. Valuations
12. Valuations
13. Tautology
14. Properties
15. Negation Normal Form
16. Literals & Clauses
17. Conjunctive Normal Form
18. Validity
19. Logical Validity
20. Validity & Tautologies
21. Problem
22. Tautology Checking
23. Falsifying

Tautology Checking Contd.  (561-574)

1. Tautology Checking
2. Normal Forms
3. Top-down Development
4. The Signature
5. The Core subproblem
6. The datatype
7. Convert to CNF
8. Rewrite into NNF
9. $\iff$ and $\Rightarrow$ Elimination
10. Push $\neg$ inward
11. Push $\neg$ inward
12. conj_of_disj
13. Push \( \lor \) inward
14. Tautology & Falsification