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

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.

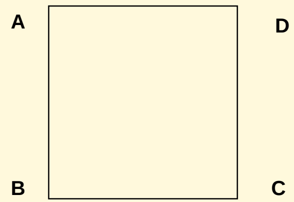
Algorithm

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.



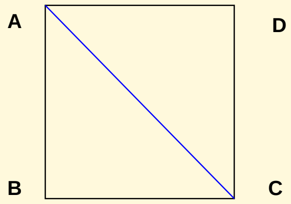
Solution: Doubling a Square

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

1. Draw the diagonal $\overline{AC}$.
2. Complete the square $\Box ACEF$ on side $\overline{AC}$.

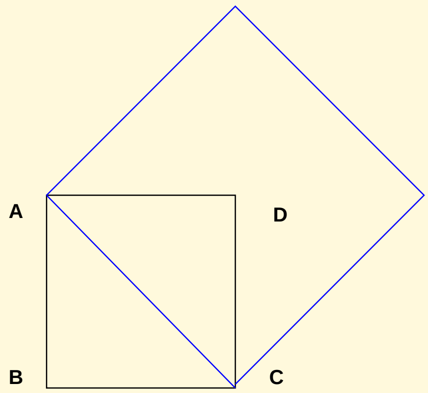
Execution: Step 1

Draw the diagonal $\overline{AC}$.



Execution: Step 2

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



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

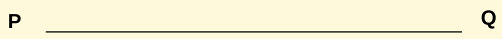
Given a line segment of length $b > 0$ construct a square of side b.

Assume given a line segment $\overline{PQ}$ of length b.

1. Construct two lines l_1 and l_2 perpendicular to $\overline{PQ}$ passing through P and Q respectively
2. On the same side of $\overline{PQ}$ mark points R on l_1 and S on l_2 such that $PR = PQ = QS$.
3. Draw $\overline{RS}$. $\square PQSR$ is a square of side b

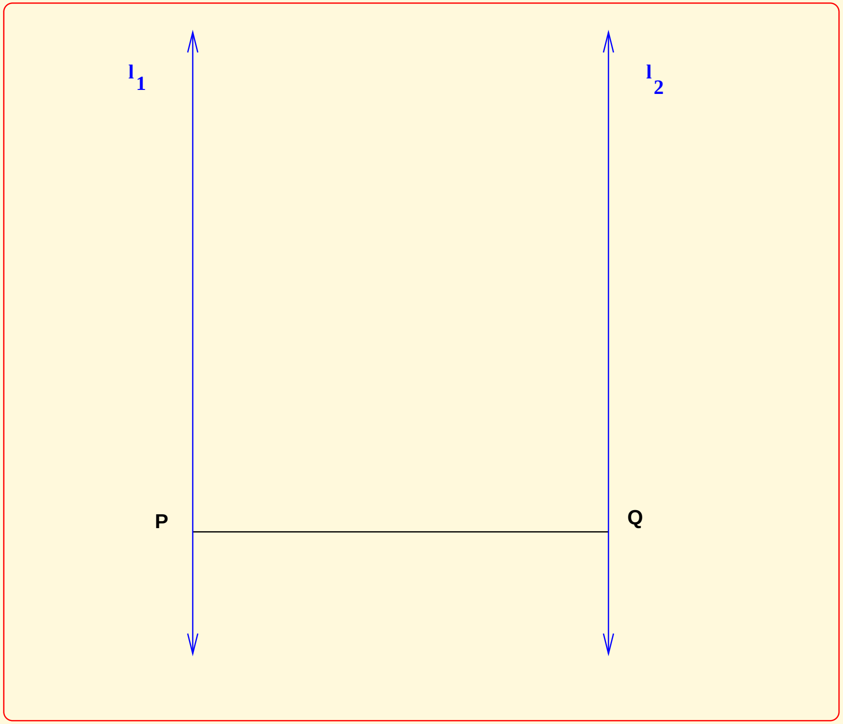
Square on Segment: 0

Assume given a line segment $\overline{PQ}$ of length b.



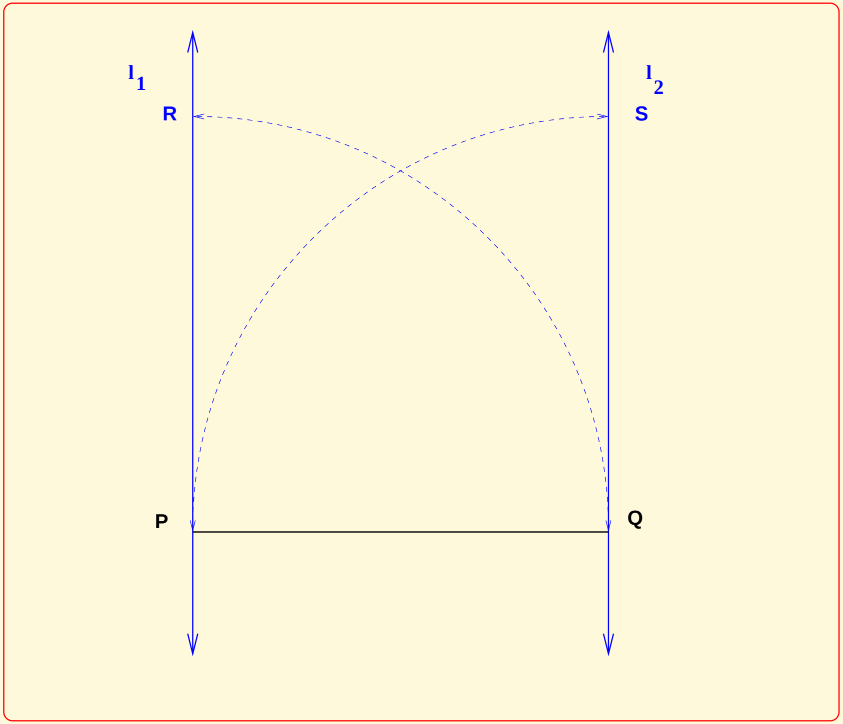
Square on Segment: 1

Construct two lines l_1 and l_2 perpendicular to $\overline{PQ}$ passing through P and Q respectively



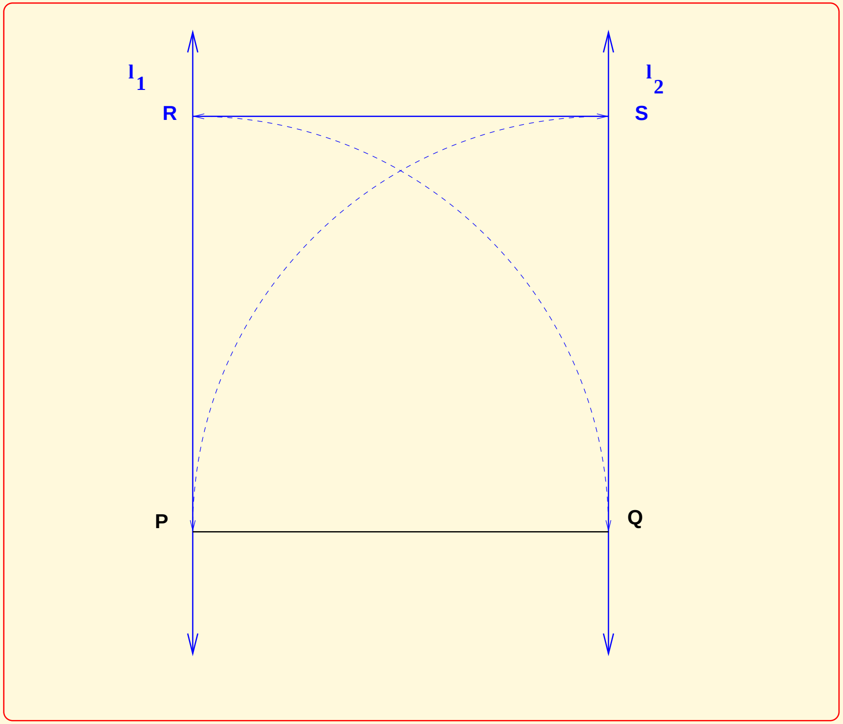
Square on Segment: 2

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



Square on Segment: 3

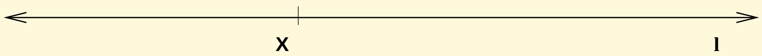
Draw $\overline{RS}$. $\square PQSR$ is a square of side b



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.

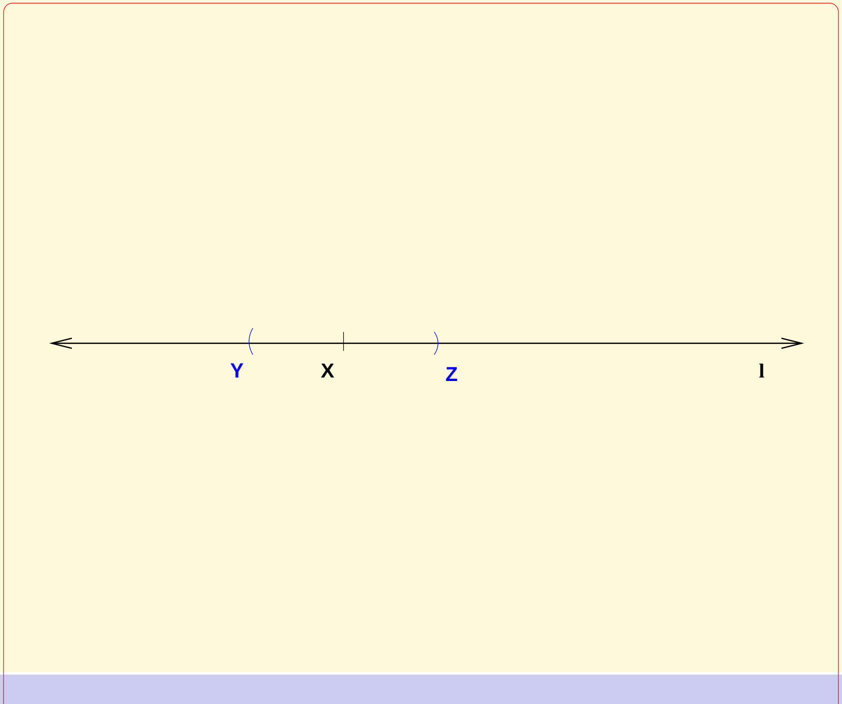


Solution: Perpendicular at a point

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.

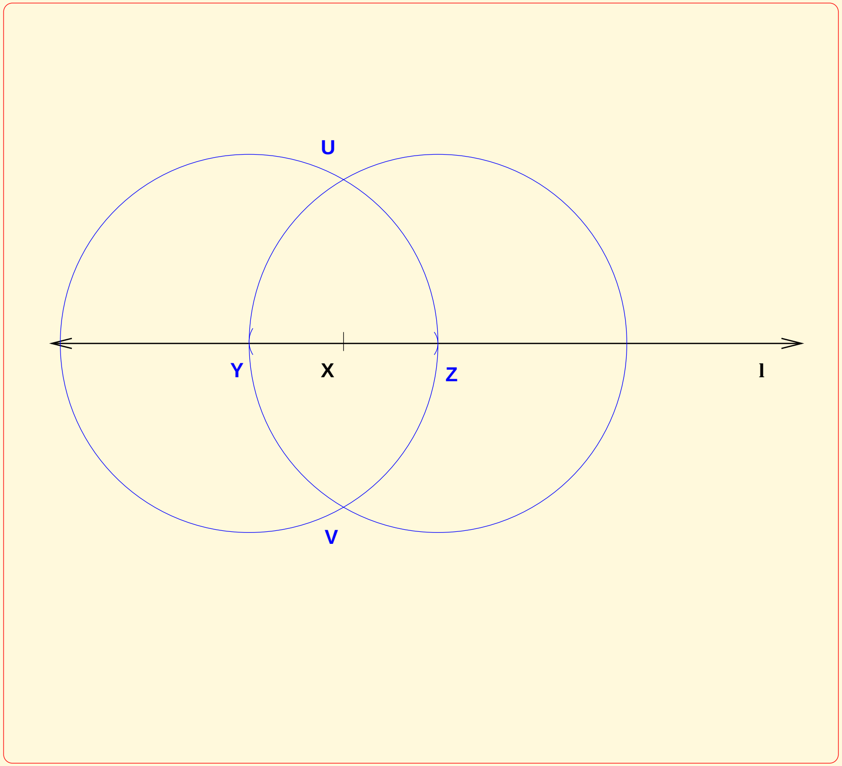
Perpendicular at a Point: 1

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

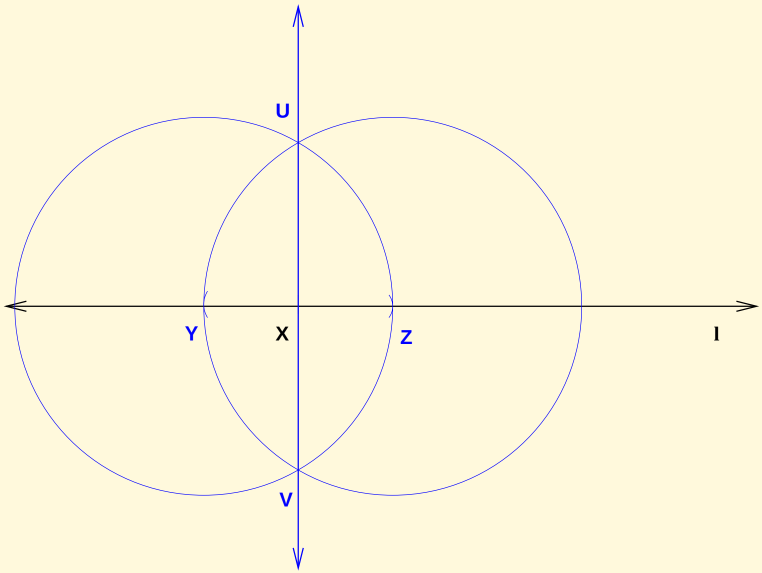


Perpendicular at a Point: 2

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



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: 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 $\overline{YZ}$.
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

Previous: Introduction to Computing

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 facilties they offer even though they are all equally powerful.

Computing Models

We consider mainly two models.

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

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

Primitives

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

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

The functional model

Primitive expressions

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

- basic data elements: numbers, characters, truth values etc.
- basic operations on the data elements: addition, substraction, multiplication, division, boolean operations, string operations etc.
- a naming mechanism for various quantitities 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

Mathematical Notation 1: Factorial

$$\mathbf{n}! = \begin{cases} \mathbf{1} & \text{if } \mathbf{n} < \mathbf{1} \\ \mathbf{n} \times (\mathbf{n} - \mathbf{1})! & \text{otherwise} \end{cases}$$

Mathematical Notation 2: Factorial

Or more informally,

$$\mathbf{n}! = \begin{cases} \mathbf{1} & \text{if } \mathbf{n} < \mathbf{1} \\ \mathbf{1} \times \mathbf{2} \times \ldots \times \mathbf{n} & \text{otherwise} \end{cases}$$

Mathematical Notation 3: Factorial

How about this?

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

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
-
```

More SML

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

Previous: Primitives: Integer & Real

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
-
```

SML: Primitive Integer Operations 2

```
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
-
```

SML: Primitive Integer Operations 2

```
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 * y;
val it = 30 : int
- val a = 25;
val a = 25 : int
- val b = 7;
val b = 7 : int
- val q = a div b;
val q = 3 : int
- val r = a mod b;
GC #0.0.0.0.2.45:    (0 ms)
val r = 4 : int
-
```

SML: Primitive Integer Operations 3

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

SML: Primitive Integer Operations 3

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

SML: Primitive Integer Operations 3

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

SML: Primitive Integer Operations 3

```
- val r = a mod b;
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 q1 = a div c;
val q1 = ~4 : int
- val r1 = a mod c;
val r1 = ~3 : int
-
```

SML: Primitive Integer Operations 3

```
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 $\mathbf{a}$ and $\mathbf{b}$, the quotient $\mathbf{q}$ and remainder $\mathbf{r}$ are uniquely determined to satisfy

- $$\mathbf{a} = \mathbf{b} \times \mathbf{q} + \mathbf{r}$$

- $$\begin{cases} 0 \leq \mathbf{r} < \mathbf{b} & \text{when } \mathbf{b} > 0 \\ \mathbf{b} < \mathbf{r} \leq 0 & \text{when } \mathbf{b} < 0 \end{cases}$$

So $0 \leq |\mathbf{r}| < |\mathbf{b}|$ always.

SML: Primitive Real Operations 1

```
sml
Standard ML of New Jersey,
- val real_a = real a;
val real_a = 25.0 : real
-
```

SML: Primitive Real Operations 1

```
sml
Standard ML of New Jersey,
- val real_a = real a;
val real_a = 25.0 : real
- real_a + b;
stdIn:40.1-40.11 Error: operator and
  operator domain: real * real
  operand:         real * int
  in expression:
    real_a + b
-
```

SML: Primitive Real Operations 1

```
stdIn:40.1-40.11 Error: operator and
  operator domain: real * real
  operand:         real * int
  in expression:
    real_a + b
- b + real_a;
stdIn:1.1-2.6 Error: operator and ope
  operator domain: int * int
  operand:         int * real
  in expression:
    b + real_a
-
```

SML: Primitive Real Operations 2

```
- val a = 25.0;
val a = 25.0 : real
- val b = 7.0;
val b = 7.0 : real
- a/b;
val it = 3.57142857143 : real
- a div b;
stdIn:49.3-49.6 Error: overloaded va
  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
-
```

SML: Primitive Real Operations 4

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

SML: Precision

```
- pi + d*10.0;
val it = 30000000003.0 : real
- pi + d*100.0;
val it = 3E12 : real
- d*100.0 + pi;
val it = 3E12 : real
- d*100.0 -pi;
val it = 3E12 : real
- d*10.0 - pi;
val it = 29999999997.0 : real
-
```

1.4. Example: Fibonacci

Fibonacci Numbers: 1

$$\begin{cases} \mathbf{F}(0) = 1 \\ \mathbf{F}(1) = 1 \\ \mathbf{F}(n) = \mathbf{F}(n-1) + \mathbf{F}(n-2) & \text{if } n > 1 \end{cases}$$

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

Fibonacci Numbers: 2

$$\begin{cases} \mathbf{F}(0) & = 1 \\ \mathbf{F}(1) & = 1 \\ \mathbf{F}(n) & = \mathbf{F}(n-1) + \mathbf{F}(n-2) \quad \text{if } n > 1 \end{cases}$$

Alternatively,

$$\mathbf{F}(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathbf{F}(n-1) + \mathbf{F}(n-2) & \text{if } n > 1 \end{cases}$$

Fibonacci Numbers: 3

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

Alternatively,

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

Fibonacci Numbers: 4

$$\begin{cases} \mathbf{F}(0) &= 1 \\ \mathbf{F}(1) &= 1 \\ \mathbf{F}(n) &= \mathbf{F}(n-1) + \mathbf{F}(n-2) \quad \text{if } n > 1 \end{cases}$$

Alternatively,

$$\mathbf{F}(n) = \mathbf{Fa}(n, 1, 1)$$

where

$$\mathbf{Fa}(n, a, b) = \begin{cases} a & \text{if } n = 0 \\ b & \text{if } n = 1 \\ \mathbf{Fa}(n-1, b, a+b) & \text{if } n > 1 \end{cases}$$

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 $\mathbf{Fa}(n, 1, 1) = \mathbf{F}(n)$?

Intuition $\mathbf{Fa}$ is a generalization of $\mathbf{F}$.

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

$$\begin{aligned}
\mathbf{Fa}(2, a, b) &= a + b \\
\mathbf{Fa}(3, a, b) &= \mathbf{Fa}(2, b, a + b) \\
&= a + 2b \\
\mathbf{Fa}(4, a, b) &= \mathbf{Fa}(3, b, a + b) \\
&= \mathbf{Fa}(2, a + b, a + 2b) \\
&= 2a + 3b \\
\mathbf{Fa}(5, a, b) &= \mathbf{Fa}(2, a + 2b, 2a + 3b) \\
&= 3a + 5b
\end{aligned}$$

Generalization

- $\mathbf{Fa}(0, a, b) = a$
- $\mathbf{Fa}(1, a, b) = b$
- $\mathbf{Fa}(n, a, b) = a\mathbf{F}(n-2) + b\mathbf{F}(n-1)$
- When $a = 1$ and $b = 1$, for all $n \geq 0$, $\mathbf{Fa}(n, a, b) = \mathbf{F}(n)$

Theorem 1 *For all integers* a, b *and* $n > 1$,

$$\boxed{\mathbf{Fa}(n, a, b) = a\mathbf{F}(n-2) + b\mathbf{F}(n-1)}$$

Proof:

Basis For $n = 2$, $\mathbf{Fa}(2, a, b) = a + b = a\mathbf{F}(0) + b\mathbf{F}(1)$

Induction hypothesis (IH) Assume $\mathbf{Fa}(k, a, b) = a\mathbf{F}(k-2) + b\mathbf{F}(k-1)$, for some $k > 1$ and all integers a, b

Induction Step

$$
\begin{aligned}
& \mathbf{Fa}(k+1, a, b) \\
= \; & \mathbf{Fa}(k, b, a+b) \quad \text{Definition of } \mathbf{Fa} \\
= \; & b\mathbf{F}(k-2) + (a+b)\mathbf{F}(k-1) \quad IH \\
= \; & a\mathbf{F}(k-1) + b(\mathbf{F}(k-2) + \mathbf{F}(k-1)) \\
= \; & a\mathbf{F}(k-1) + b\mathbf{F}(k) \quad \text{Definition of } \mathbf{F}
\end{aligned}
$$

Another Generalization

Try to prove a different and more direct theorem.

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

$$\boxed{\mathbf{Fa}(n, 1, 1) = \mathbf{F}(n)}$$

Try Proving it!

Proof: By induction on $n > 1$.

Basis For $n = 0$ and $n = 1$, $\mathbf{Fa}(0, 1, 1) = 1 = \mathbf{Fa}(1)$

Induction hypothesis (IH) Assume $\mathbf{Fa}(k, 1, 1) = \mathbf{F}(k)$, for some $k > 1$

Induction Step

$$\begin{aligned} & \mathbf{Fa}(k+1, 1, 1) \\ = & \ \mathbf{Fa}(k, 1, 2) \quad \text{Definition of } \mathbf{Fa} \\ = & \ ??? \quad IH \end{aligned}$$

STUCK!

Another Generalization

Try to prove a different and more direct theorem.

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

$$\boxed{\mathbf{Fa}(n, \mathbf{F}(j-1), \mathbf{F}(j)) = \mathbf{F}(n+j-1)}$$

Proof: By induction on $n \geq 1$, for all values of $j \geq 1$. $\square$

Corollary 4 *For all integers* $n \geq 1$,

$$\boxed{\mathbf{Fa}(n, \mathbf{F}(0), \mathbf{F}(1)) = \mathbf{F}(n)}$$

Try Proving it!

Basis For $n = 1$, $\mathbf{Fa}(1, \mathbf{F}(j-1), \mathbf{F}(j)) = \mathbf{F}(j)$

Induction hypothesis (IH) For some $k > 1$ and all $j \geq 1$, $\mathbf{Fa}(k, \mathbf{F}(j-1), \mathbf{F}(j)) = \mathbf{F}(k+j-1)$

Induction Step We need to prove $\mathbf{Fa}(k+1, \mathbf{F}(j-1), \mathbf{F}(j)) = \mathbf{F}(k+j)$.

$$\begin{aligned}
& \mathbf{Fa}(k+1, \mathbf{F}(j-1), \mathbf{F}(j)) \\
= \ & \mathbf{Fa}(k, \mathbf{F}(j), \mathbf{F}(j-1) + \mathbf{F}(j)) \\
= \ & \mathbf{Fa}(k, \mathbf{F}(j), \mathbf{F}(j+1)) \\
= \ & \mathbf{Fa}(k + (j+1) - 1) \quad \text{IH} \\
= \ & \mathbf{Fa}(k+j)
\end{aligned}$$

Complexity

- Time complexity:
  - No of additions: $\mathcal{A}_{\mathbf{F}}(n)$
  - No of comparisons: $\mathcal{C}_{\mathbf{F}}(n)$
  - No of recursive calls to $\mathbf{F}$: $\mathcal{R}_{\mathbf{F}}(n)$
- Space complexity:

Complexity

- Time complexity:
- Space complexity:
  - left-to-right evaluation: $\mathcal{LR}_{\mathbf{F}}(n)$
  - arbitrary evaluation: $\mathcal{U}_{\mathbf{F}}(n)$

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 $\mathcal{R}$.

Time Complexity: $\mathcal{R}$

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

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

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

Proof: By induction on $n > 2$. $\square$

Bound on $\mathcal{R}$

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

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

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

$$
\begin{aligned}
& \mathcal{R}_{\mathbf{F}}(n) \\
= \ & 2 + \mathcal{R}_{\mathbf{F}}(n-2) + \mathcal{R}_{\mathbf{F}}(n-1) \\
\leq \ & 2 + 2^{n-3} + 2^{n-2} \quad \text{(IH)} \\
\leq \ & 2.2^{n-3} + 2^{n-2} \quad \text{for } n > 3,\ 2^{n-3} \geq 2 \\
= \ & 2^{n-2} + 2^{n-2} \\
= \ & 2^{n-1}
\end{aligned}
$$

Other Bounds: $\mathcal{C}_{\mathbf{F}}$

One comparison for each call.

- $\mathcal{C}_{\mathbf{F}}(0) = \mathcal{C}_{\mathbf{F}}(1) = 1$
- $\mathcal{C}_{\mathbf{F}}(n) = 1 + \mathcal{C}_{\mathbf{F}}(n-1) + \mathcal{C}_{\mathbf{F}}(n-2)$ for $n > 1$

Theorem 6 $\mathcal{C}_{\mathbf{F}}(n) \leq 2^n$ *for all* $n \geq 0$.

Other Bounds: $\mathcal{A}_{\mathbf{F}}$

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

- $\mathcal{A}_{\mathbf{F}}(0) = \mathcal{A}_{\mathbf{F}}(1) = 0$
- $\mathcal{A}_{\mathbf{F}}(n) = 1 + \mathcal{A}_{\mathbf{F}}(n-1) + \mathcal{A}_{\mathbf{F}}(n-2)$ for $n > 1$

Theorem 7 $\mathcal{A}_{\mathbf{F}}(n) \leq 2^{n-1}$ *for all* $n > 0$.

1.5. Primitives: Booleans

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 >= 10) andalso (n=10);
val it = true : bool
- n < 0 orelse n >= 10;
val it = true : bool
- not ((n >= 10)
= andalso (n=10))
= orelse n < 0 orelse n >= 10;
val it = true : bool
-
```

$\wedge$ vs. `andalso`

p	q	$p \wedge q$	p `andalso` q
true	true	true	`true`
true	false	false	`false`
false	true	false	`false`
false	false	false	`false`

$\vee$ vs. `orelse`

p	q	$p \vee q$	p `orelse` q
true	true	true	`true`
true	false	true	`true`
false	true	true	`true`
false	false	false	`false`

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`, $\perp$

p	q	$p \wedge q$	p `andalso` q
true	$\perp$	$\perp$	$\perp$
$\perp$	true	$\perp$	$\perp$
false	$\perp$	false	`false`
$\perp$	false	false	$\perp$

$\wedge$ is commutative whereas `andalso` is not.

or, `orelse`, $\perp$

p	q	$p \vee q$	p `orelse` q
true	$\perp$	true	`true`
$\perp$	true	true	$\perp$
false	$\perp$	$\perp$	$\perp$
$\perp$	false	$\perp$	$\perp$

$\vee$ is commutative whereas `orelse` is not.

Complex Boolean Conditions

Assume p and q are boolean conditions

$$\boxed{p \ \mathtt{orelse} \ q \equiv \mathtt{if} \ p \ \mathtt{then} \ \mathtt{true} \ \mathtt{else} \ q}$$

$$\boxed{p \ \mathtt{andalso} \ q \equiv \mathtt{if} \ p \ \mathtt{then} \ q \ \mathtt{else} \ \mathtt{false}}$$

2. Algorithms: Design & Refinement

2.1. Technical Completeness & Algorithms

Recapitulation: Integers & Real

- Primitive Integer Operations
- Primitive Real Operations
- Some algorithms

Forward

Recap: Integer Operations

- Primitive Integer Operations
  - Naming, $+$, $-$, $\sim$
  - Multiplication, division
  - Quotient & remainder
- Primitive Real Operations
- Some algorithms

Back

Recapitulation: Real Operations

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

Back

Recapitulation: Simple Algorithms

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

Back

More Algorithms

- Powering
- Integer square root
- Combinations ${}^{n}C_{k}$

Powering: Math

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

$$x^n = \underbrace{x \times x \times \cdots \times x}_{n \text{ times}}$$

Noting that $x^0 = 1$ we give an inductive definition:

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x^{n-1} \times x & \text{otherwise} \end{cases}$$

Powering: SML

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

Is it technically complete?

Technical completeness

Can it be always guaranteed that

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

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

What SML says

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

Can it be always guaranteed that

- x will be real? YES
- n will be integer? YES

Technical completeness

Can it be always guaranteed that

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

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

```
- power(0.0, 0);
val it = 1.0 : real
```

What SML says ... *contd*

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

Goes on forever!

Powering: Math 1

For any real number x and integer n

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

Powering: SML 1

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

Is this definition technically complete?

Technical Completeness

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

$$
\begin{aligned}
& 0.0^n \\
= \; & 1.0/(0.0^m) \\
= \; & 1.0/0.0
\end{aligned}
$$

Division by zero!

What SML says

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

SML is somewhat more understanding 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: $\perp$ denotes the $undefined$

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
```

Integer Power: SML

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

uncaught exception negExponent
  raised at: intpower.sml:4.16-4.32
- intpower (0, 0);

uncaught exception zeroPowerZero
  raised at: stdIn:24.26-24.39
```

Back to More Algos

Integer Square Root 1

$$isqrt(n) = \lfloor \sqrt{n} \rfloor$$

```
- fun isqrt n =
      trunc (Real.Math.sqrt
              (real (n)));
val isqrt = fn : int -> int
- isqrt (38);
val it = 6 : int
- isqrt (~38);
uncaught exception domain error
  raised at: boot/real64.sml:89.32-8
-
```

Integer Square Root 2

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

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

That is,

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

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

This value of k is unique!

An analysis

$$
\begin{aligned}
& 0 \leq k^2 \leq n < (k+1)^2 \\
\Rightarrow\ & 0 \leq k \leq \sqrt{n} < k+1 \\
\Rightarrow\ & 0 \leq k \leq n
\end{aligned}
$$

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

$$\boxed{l^2 \leq n < u^2}$$

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

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

where

Algorithm: shrink

$$shrink(n, l, u) =$$

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

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

SML: 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

```
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
```

shrink: Another algorithm

Shrink

$$shrink2(n, l, u) =$$

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

where $m = (l + u)\ \mathtt{div}\ 2$

Shrink2: SML

```
fun shrink2 (n, l, u) =
    if l>u orelse
       l*l > n orelse
       u*u < n
    then raise intervalError
    else if l = u
    then l
```

Shrink2: SML ... *contd*

```
else
let val m = (l+u) div 2;
    val msqr = m*m
in  if msqr <= n
    then shrink (n, m, u)
    else shrink (n, l, m)
end;
```

Back to More Algos

2.2. Algorithm Refinement

Recap: More Algorithms

- x^n for real and integer x
- Integer square root

Forward

Recap: Power

- x^n for real and integer x
  - Technical Completness
    - Undefinedness
    - Termination
  - More complete definition for real x
  - Power of an integer
  - $\perp$ and exceptions
- Integer square root

Recap: Technical completeness

Can it be always guaranteed that

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

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

INFINITE COMPUTATION

Recap: More Algorithms

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

Intsqrt: Reorganized

```
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 if n<0
     then raise negError
     else if n=0
     then 0
     else shrink (n, 0, n)
end
```

Back

Some More Algorithms

- Combinations
- Perfect Numbers

Combinations: Math

$$
\begin{aligned}
{}^{n}C_k &= \frac{n!}{(n-k)!k!} \\
&= \frac{n(n-1)\cdots(n-k+1)}{k!} \\
&= \frac{n(n-1)\cdots(k+1)}{(n-k)!} \\
&= {}^{n-1}C_{k-1} + {}^{n-1}C_k
\end{aligned}
$$

Since we already have the function `fact`, we may program ${}^{n}C_k$ 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 $0 \leq k \leq n$

$$
{}^{n}C_k = \begin{cases} \perp & \text{if } n < 0 \text{ or} \\ & k < 0 \text{ or} \\ & k > n \\ 1 & \text{if } n = 0 \text{ or} \\ & k = 0 \text{ or} \\ & k = n \\ {}^{n-1}C_{k-1} + {}^{n-1}C_k & \text{otherwise} \end{cases}
$$

Combinations: SML

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

Back to Some More Algorithms

Perfect Numbers

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

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

$$perfect(n)$$

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

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

where

Refinement

1. $ifdivisor(k)$ needs to be defined
2. $\sum_{k=1}^{n-1} ifdivisor(k)$ 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

$$\sum_{l}^{u} ifdivisor(k)$$

$\sum_{k=l}^{u} ifdivisor(k) =$

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

where $ifdivisor(k)$ needs to be defined

SML: `sum_divisors`

From the algorithmic definition of $\sum_{k=l}^{u} ifdivisor(k)$

```
fun sum_divisors (l, u) =
    if l > u
    then 0
    else ifdivisor (l) +
         sum_divisors (l+1, u)
```

where $ifdivisor(k)$ still needs to be defined

$ifdivisor$ 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 ...

SML: Assembly 1

```
fun sum_divisors (l, u) =
  if l > u then 0
  else
    let fun ifdivisor (k) =
          if n mod k = 0
          then k
          else 0
    in ifdivisor (l) +
       sum_divisors (l+1, u)
    end
```

Clearly $k \in [l, u]$

SML: Assembly 2

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

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

Technically complete!

Perfect Numbers ...*contd.*

Clearly for all k, $n/2 < k < n$, $ifdivisor(k) = 0$.

$$\lfloor n/2 \rfloor = n \;\mathtt{div}\; 2 < n/2$$

Hence

$$\sum_{k=1}^{n-1} ifdivisor(k) = \sum_{k=1}^{n \;\mathtt{div}\; 2} ifdivisor(k)$$

Perfect Numbers ...*contd.*

Hence

$$perfect(n)$$

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

$$\Longleftrightarrow n = \sum_{k-1}^{n \texttt{ div } 2} ifdivisor(k)$$

where

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

SML: Assembly 3

```
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 $k \in [l, u] = [1, n \texttt{ div} 2]$ whenever $n > 0$.

Technically complete!

Perfect Numbers: Run

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

Perfect Numbers: Run

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

SML: Code variations

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

Technically complete though `ifdivisor`, by itself is not!

SML: Code variations

What about this?

```
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?

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

Technically incomplete!

Summation: Generalizations

Need a method to compute summations in general.

For any function $f : \mathbb{Z} \rightarrow \mathbb{Z}$ and integers l and u,

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

Algorithmic Improvements:

1. $perfect2$
2. $shrink2$

Algorithmic Variations

1. For any $k|n$, $m = n$ div k is also a divisor of n
2. 1 is a divisor of every positive number
3. For $n > 2$, $\lfloor\sqrt{n}\rfloor < n \,\texttt{div}\, 2$
4. Hence $\sum_{k=1}^{n \,\texttt{div}\, 2} ifdivisor(k) =$

$$1 + \sum_{k=2}^{\lfloor\sqrt{n}\rfloor} ifdivisor2(k)$$

IIT Delhi
S. Arun-Kumar, CSE

Algorithmic Variations

$$perfect(n)$$

$$\Longleftrightarrow n = 1 + \sum_{k=2}^{\lfloor\sqrt{n}\rfloor} ifdivisor2(k)$$

where

$$ifdivisor2(k) = \begin{cases} k + \\ (n \; \mathtt{div} \; k) & \text{if } k|n \\ 0 & \text{otherwise} \end{cases}$$

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

2.3. Variations: Algorithms & Code

Recap

- Combinations
- Perfect Numbers
- Code Variations
- Algorithmic Variations

forward

Recap: Combinations

$$
\begin{aligned}
{}^{n}C_{k} &= \frac{n!}{(n-k)!k!} \\
&= \frac{n(n-1)\cdots(n-k+1)}{k!} \\
&= \frac{n(n-1)\cdots(k+1)}{(n-k)!} \\
&= {}^{n-1}C_{k-1} + {}^{n-1}C_{k}
\end{aligned}
$$

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));
```

Combinations 2

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

Combinations 3

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

Perfect 2

$$perfect(n)$$

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

where

$$ifdivisor2(k) =$$

$$\begin{cases} k+m & \text{if } k|n \text{ and } k \neq m \\ k & \text{if } k|n \text{ and } k = m \\ 0 & \text{otherwise} \end{cases}$$

where $m = (n \;\mathtt{div}\; k)$

Power 2

power

The previous inductive definition used

$$x^n = \underbrace{(x \times x \times \cdots \times x)}_{n-1 \text{ times}} \times x$$

We could associate the product differ-ently

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

$power2(x, n) =$

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

where $even(n) \iff n \bmod 2 = 0$.

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?

incorrectness

Recall

- A program is an
  - explicit,
  - unambiguous and
  - technically complete

  translation of an algorithm written in mathematical notation.
- Moreover, mathematical notation is more concise than a program.
- Hence mathematical notation is easier to analyse and diagnose.

Features: Definition before Use

incorrect version

Definition of a name before use:

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

Run ifdivisor

```
Standard ML of New Jersey,
- fun ifdivisor(k) =
= if n mod k = 0
= then k
= else 0
;
stdIn:18.8 Error:
unbound variable
or constructor: n
```

Diagnosis: Features of programs

incorrect version

- So both $sum\_divisors(l, u)$ and $perfect(n)$ may use $ifdivisor(k)$.
- $sum\_divisors(l, u)$ is defined before $perfect(n)$.
- So $perfect(n)$ may use both $ifdivisor(k)$ and $sum\_divisors(l, u)$

Back to Math

incorrect version

Let

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

and $sum\_divisors(l, u) =$

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

and $perfect(n) \iff$

$$n = sum\_divisors(1, \lfloor n/2 \rfloor)$$

Incorrectness

incorrect version

- $ifdivisor(k)$ 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 $sum\_divisors(l, u) =$

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

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

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

and $sum\_divisors2(\underline{n},l,u) =$

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

and $perfect(\underline{n}) \iff$

$$n = sum\_divisors2(n,1,\lfloor n/2 \rfloor)$$

Scope

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

Scope Rules

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

forward

2.4. Names, Scopes & Recursion

Disjoint Scopes

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

Nested Scopes

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

Overlapping Scopes

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

Spannning

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

Scope & Names

- A name may occur either as being defined or as a use of a previously defined name
- The same name may be used to refer to different objects.
- The use of a name refers to the textually most recent definition in the innermost enclosing scope

diagram

Names & References

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

[Back to scope names]

Names & References

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

Back to scope names

Names & References

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

Back to scope names

Names & References

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

Back to scope names

Names & References

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

Back to scope names

Names & References

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

Back to scope names

Names & References

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

Back to scope names

Names & References

```
let
    val x = 10; val x = x - 5;
    fun fun1 y =
        let
            ...
        in
            ...
        end
    fun fun2 z =
        let
            ...
        in
            ...
        end
in fun1 (fun2 x)
end
```

Back to scope names

Names & References

```
let
  val x = 10; val x = x - 5;
  fun fun1 y =
      let
          ...
      in
          ...
      end
  fun fun2 z =
      let
          ...
      in
          ...
      end
in fun1 (fun2 x)
end
```

Back to scope names

Names & References

```
let
    val x = 10; val x = x - 5;
    fun fun1 y =
        let
            ...
        in
            ...
        end
    fun fun2 z =
        let
            ...
        in
            ...
        end
in fun1 (fun2 x)
end
```

Back to scope names

Definition of Names

Definitions are of the form

$\mathit{qualifier}\ \underline{name} \ldots = \mathit{body}$

- `val` $\underline{name}$ =
- `fun` $\underline{name}$ ($\underline{argnames}$) =
- `local` $definitions$
  `in` $definition$
  `end`

Use of Names

Names are used in expressions.
Expressions may occur

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

```
let definitions
in  expression
end
```

use of local

local...in...end

perfect

```
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;
```

local...in...end

perfect

```
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 ifdivisor3 (n, l)
      + sum_div2 (n, l+1, u)
```

local...in...end

perfect

```
in
   fun perfect n =
      if n <= 0
      then raise invalidArg
      else
         let
            val nby2 = n div 2
         in
         n = sum_div2 (n, 1, nby2)
         end
end
```

local...in...end

perfect

```
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
```

IIT Delhi

S. Arun-Kumar, CSE

Title Page

Contents

Go Back

Full Screen

Close

Quit

Scope & `local`

```
local
        fun fun1 y =
                          ...

        fun fun2 z = ...
                        fun1

  in
        fun fun3 x =   ...
                      fun2 ...
                          fun1 ...
  end
```

Computations: Simple

For most simple expressions it is

- left to right, and
- top to bottom

except when

- presence of brackets
- precedence of operators

determine otherwise.

Hence

Simple computations

$$
\begin{aligned}
& 4+6-(4+6)\ \texttt{div}\ 2 \\
= \; & 10-(4+6)\ \texttt{div}\ 2 \\
= \; & 10-10\ \texttt{div}\ 2 \\
= \; & 10-5 \\
= \; & 5
\end{aligned}
$$

Computations: Composition

$$f(x) = x^2 + 1$$

$$g(x) = 3 * x + 2$$

Then for any value $a = 4$

$$\begin{aligned}
& f(g(a)) \\
= \; & f(3 * 4 + 2) \\
= \; & f(14) \\
= \; & 14^2 + 1 \\
= \; & 196 + 1 \\
= \; & 197
\end{aligned}$$

Composition: Alternative

$$f(x) = x^2 + 1$$

$$g(x) = 3 * x + 2$$

Why not

$$\begin{aligned}
& f(g(a)) \\
= \; & g(4)^2 + 1 \\
= \; & (3 * 4 + 2)^2 + 1 \\
= \; & (12 + 2)^2 + 1 \\
= \; & 14^2 + 1 \\
= \; & 196 + 1 \\
= \; & 197
\end{aligned}$$

Compositions: Compare

$$
\begin{array}{rlrl}
 & f(g(a)) & & f(g(a)) \\
= & f(3*4+2) & = & g(4)^2+1 \\
= & f(14) & = & (3*4+2)^2+1 \\
= & & = & (12+2)^2+1 \\
= & 14^2+1 & = & 14^2+1 \\
= & 196+1 & = & 196+1 \\
= & 197 & = & 197
\end{array}
$$

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

$$factL(n) = \begin{cases} 1 & \text{if } n = 0 \\ factL(n-1) * n & \text{otherwise} \end{cases}$$

$$factR(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * factR(n-1) & \text{otherwise} \end{cases}$$

Recursion: Left

$$
\begin{aligned}
& factL(4) \\
= & (factL(4-1)*4) \\
= & (factL(3)*4) \\
= & ((factL(3-1)*3)*4) \\
= & ((factL(2)*3)*4) \\
= & (((factL(2-1)*2)*3)*4) \\
= & \cdots
\end{aligned}
$$

Recursion: Right

$$
\begin{aligned}
& factR(4) \\
= & (4 * factR(4-1)) \\
= & (4 * factR(3)) \\
= & (4 * (3 * factR(3-1))) \\
= & (4 * (3 * factR(2))) \\
= & (4 * (3 * (2 * factR(2-1)))) \\
= & \cdots
\end{aligned}
$$

3. Introducing Reals

3.1. Floating Point

So Far-1: Computing

- The general nature of computation
- The notion of primitives, composition & induction
- The notion of an algorithm
- The digital computer & programming language

So Far-2: Algorithms & Programs

- Algorithms: Finite mathematical processes
- Programs: Precise, unambiguous explications of algorithms
- Standard ML: Its primitives
- Writing technically complete specifications

So far-3: Top-down Design

integer Square Root

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

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.

back

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 $\mathtt{a}^*$ is the computed value,

absolute error = $\mathtt{a}^* - a$

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

Errors

Errors in floating point computations are mainly due

finite precision Round-off errors

fnite process 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 ∞

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

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

$$\sin x = \sum_{m=0}^{\infty} \frac{(-1)^m x^{2m+1}}{(2m+1)!}$$

Truncation

Truncation Errors

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

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

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

$$\sin x = \sum_{m=0}^{n} \frac{(-1)^m x^{2m+1}}{(2m+1)!}$$

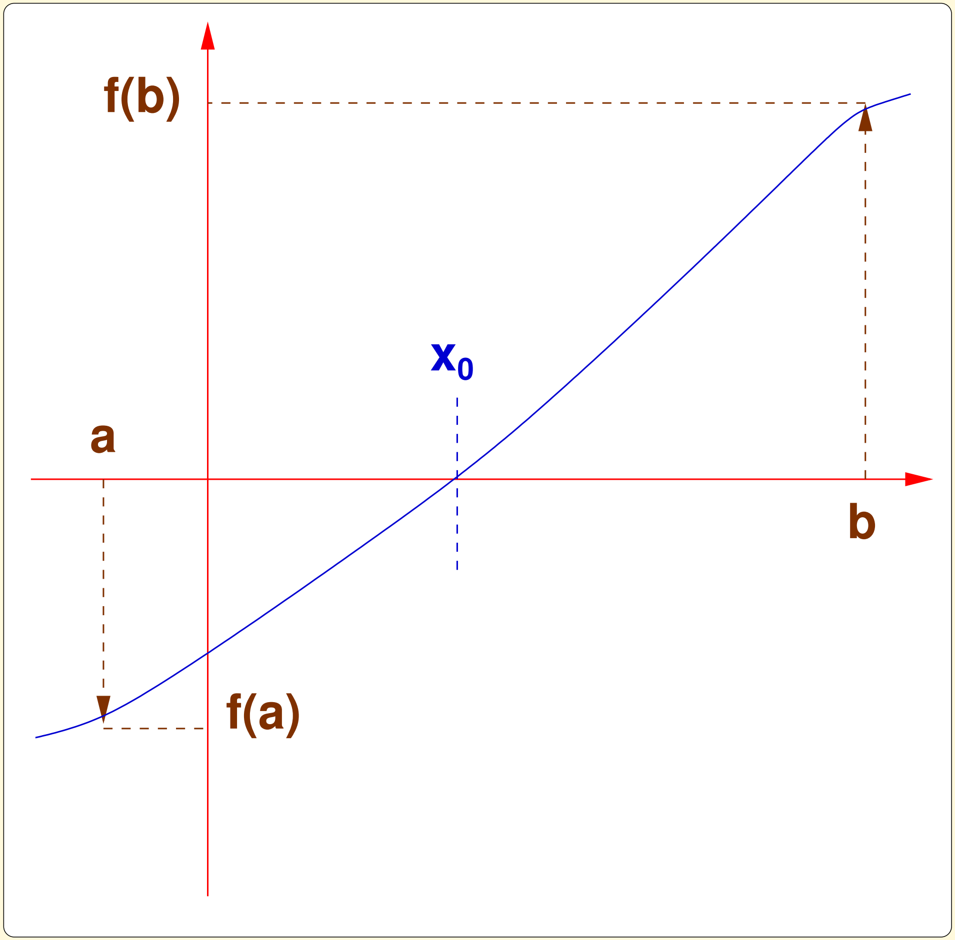
back to Errors

Equation Solving

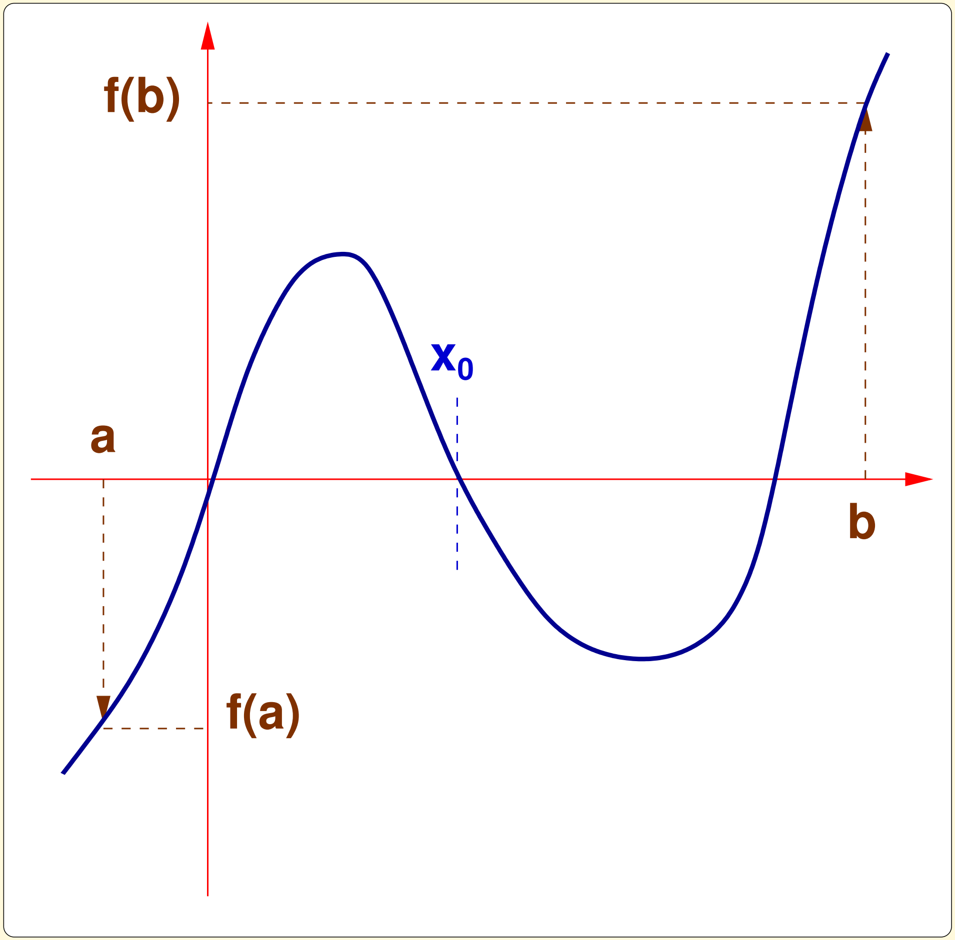
- The fifth most basic operation
- Root finding: A particular form of equation solving

$$f(x) = 0$$

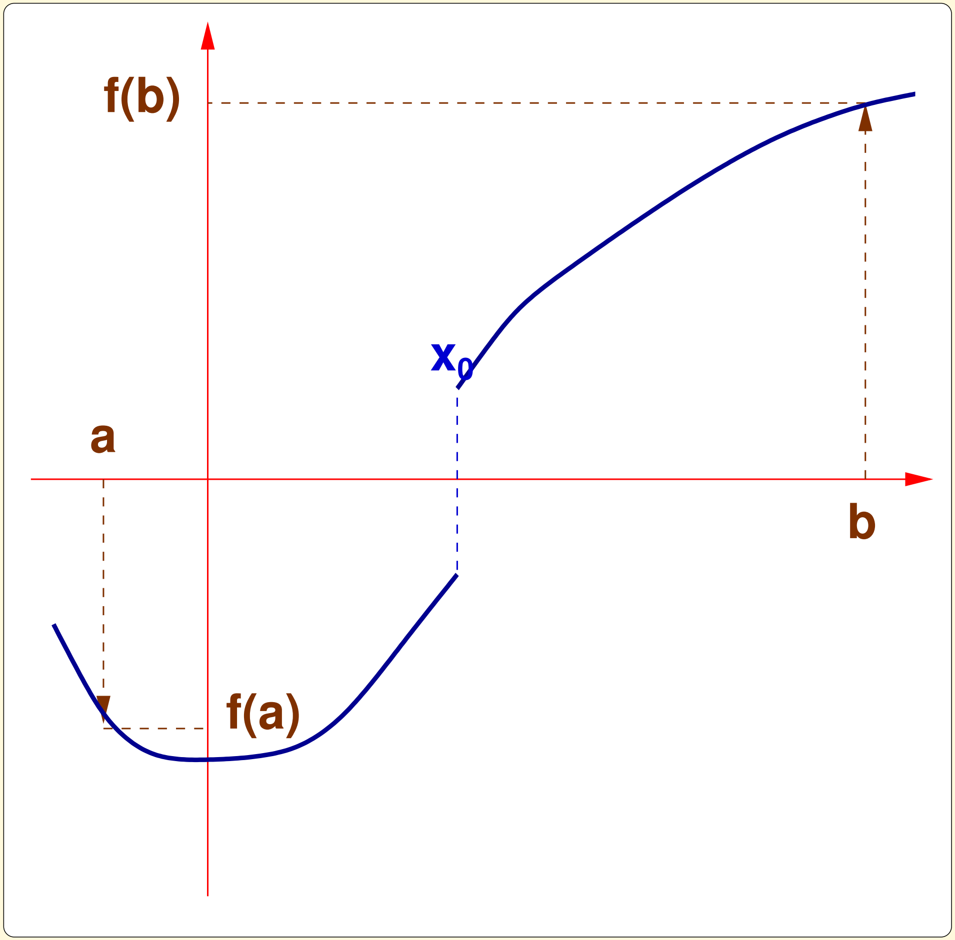
Root Finding-1



Root Finding-2

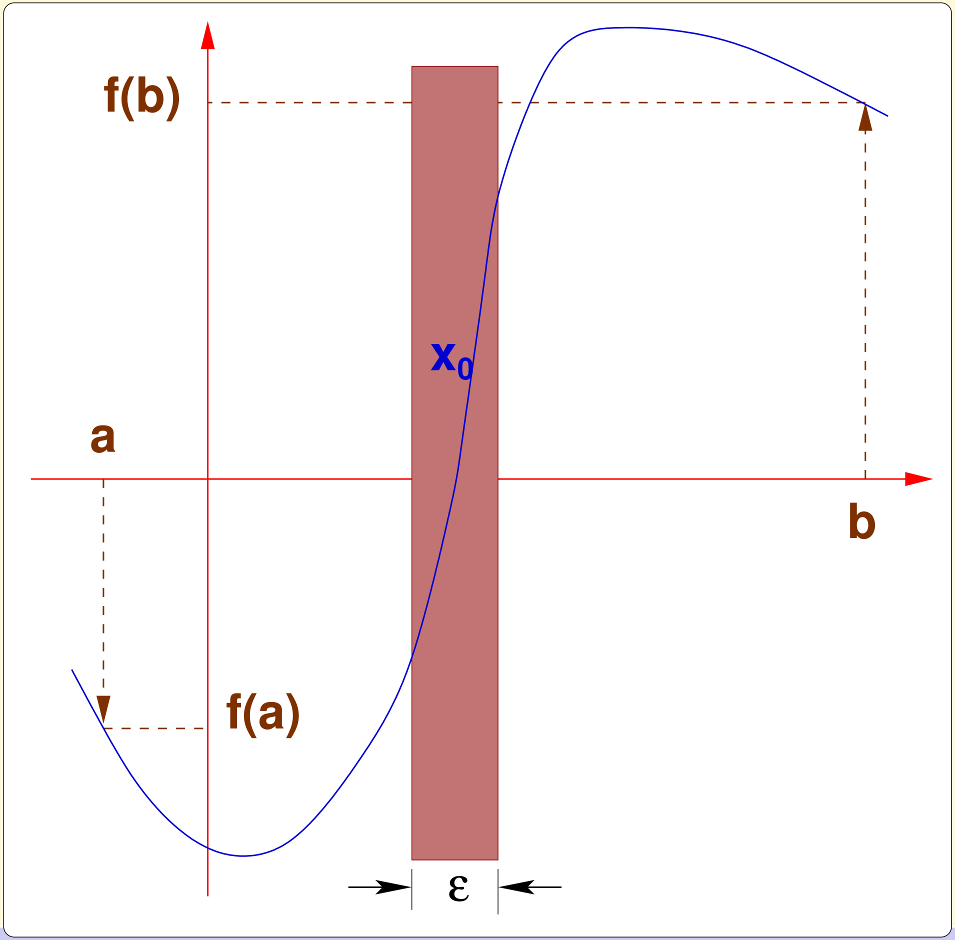


Root Finding-3



Root Finding-4

Rather steep isn't it?



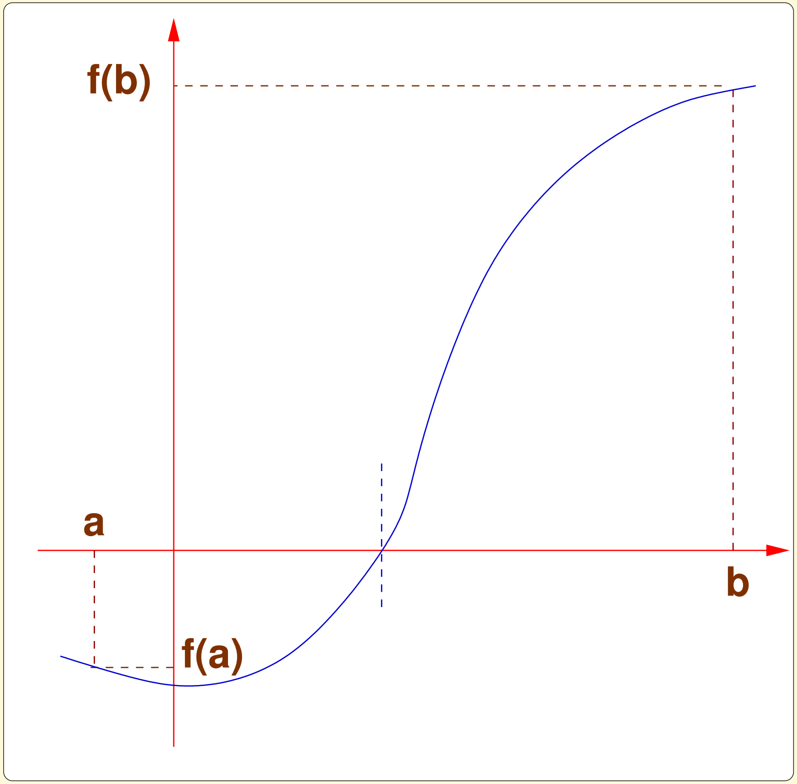
3.2. Root Finding, Composition and Recursion

Root Finding: Newton's Method

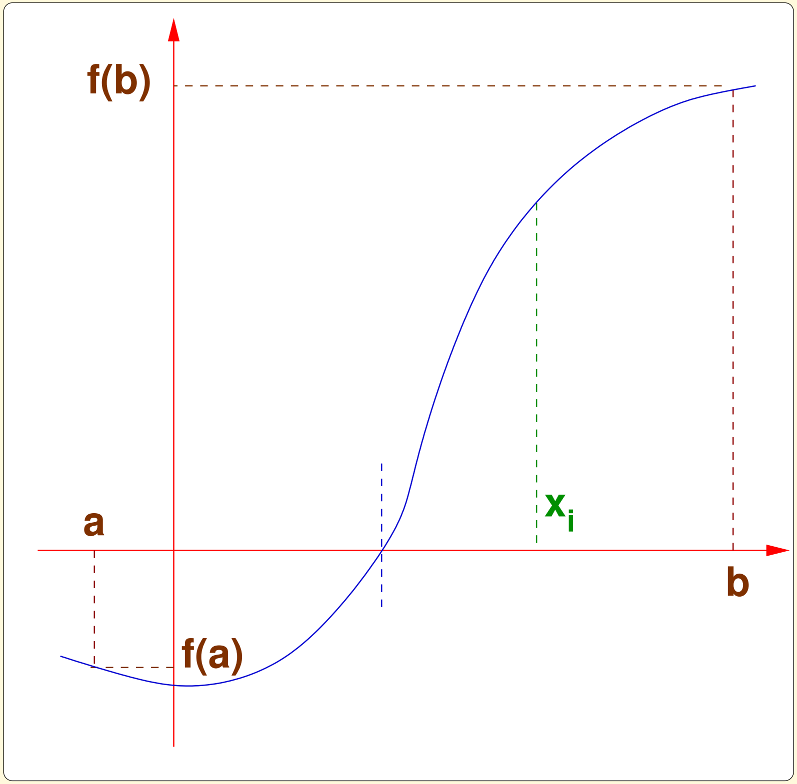
Consider a function $f(x)$

- 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

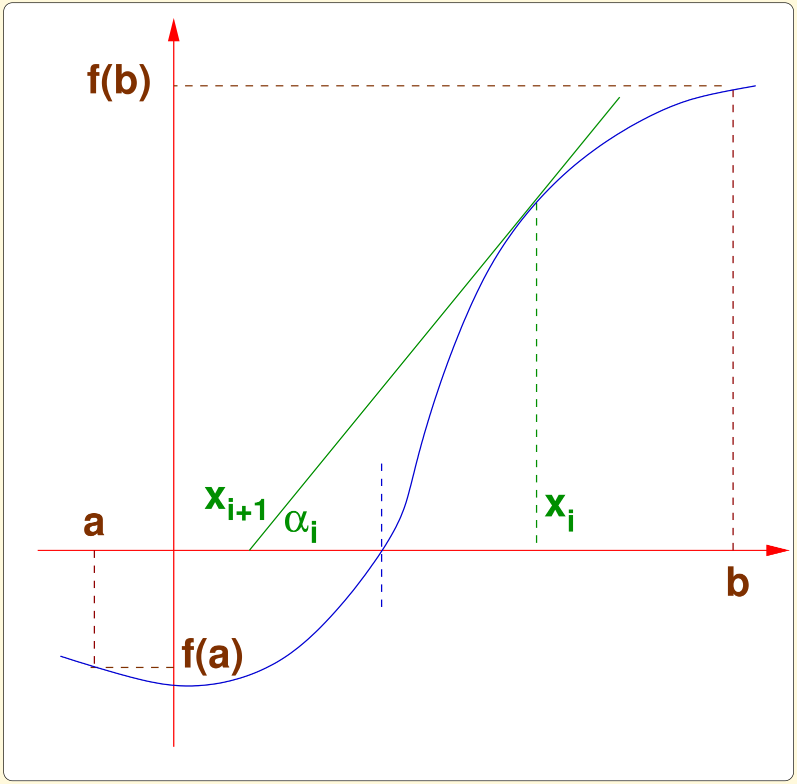
Root Finding: Newton's Method



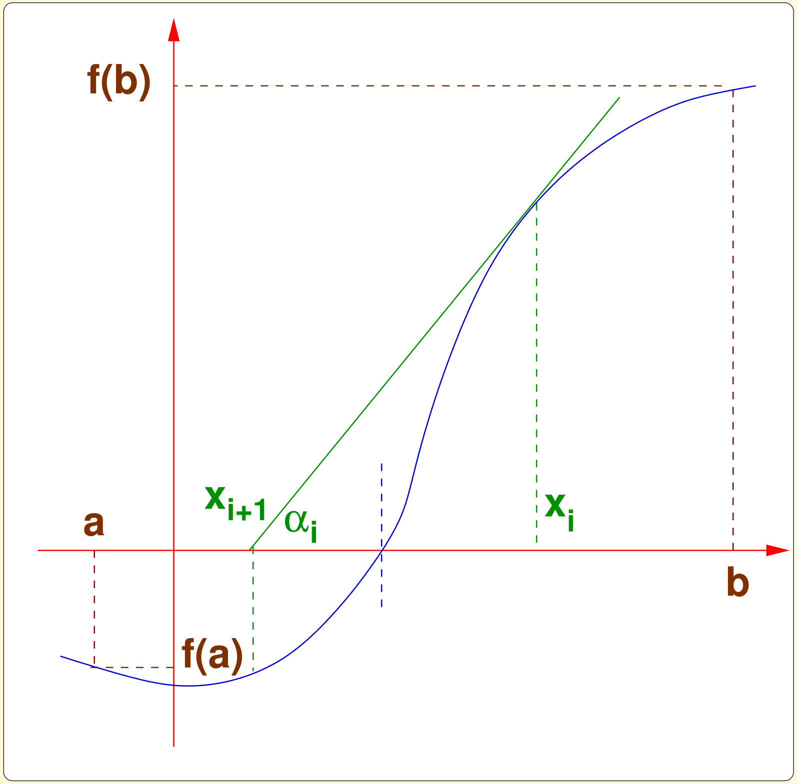
Root Finding: Newton's Method



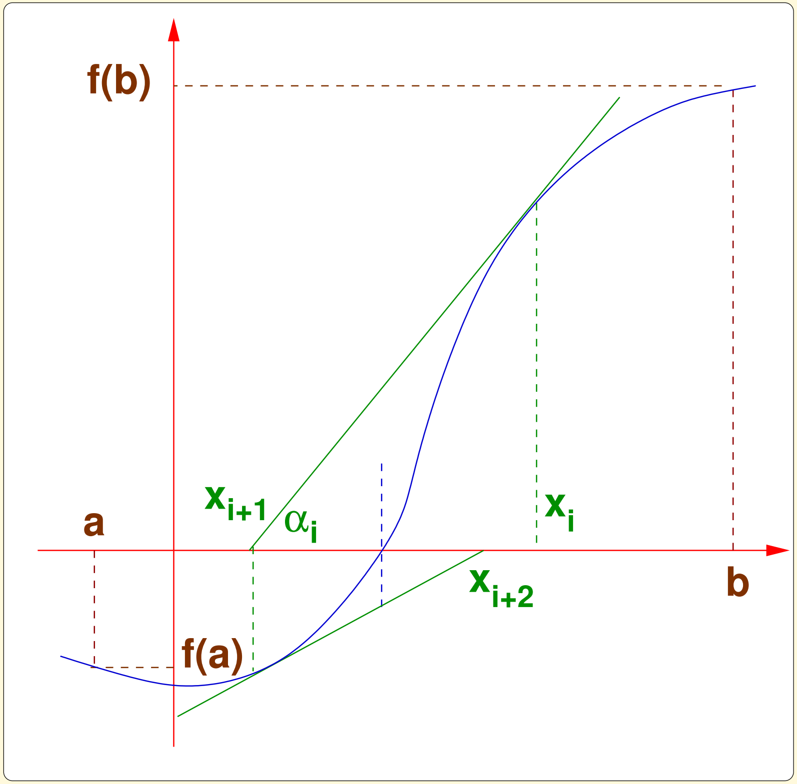
Root Finding: Newton's Method



Root Finding: Newton's Method



Root Finding: Newton's Method



Newton's Method: Basis

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

whence

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

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

$$f(x_0), f(x_1), f(x_2), \cdots \longrightarrow 0$$

Newton's Method: Basis

$$i.e. \lim_{n\to\infty} |f(x_n)| = 0$$

$$i.e. \forall \varepsilon > 0 : \exists N \geq 0 : \forall n > N :$$

$$|f(x_n)| < \varepsilon$$

then the sequence

$$x_0, x_1, x_2, \cdots$$

converges to a root of f.

Newton' Method: Algorithm

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

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

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

where

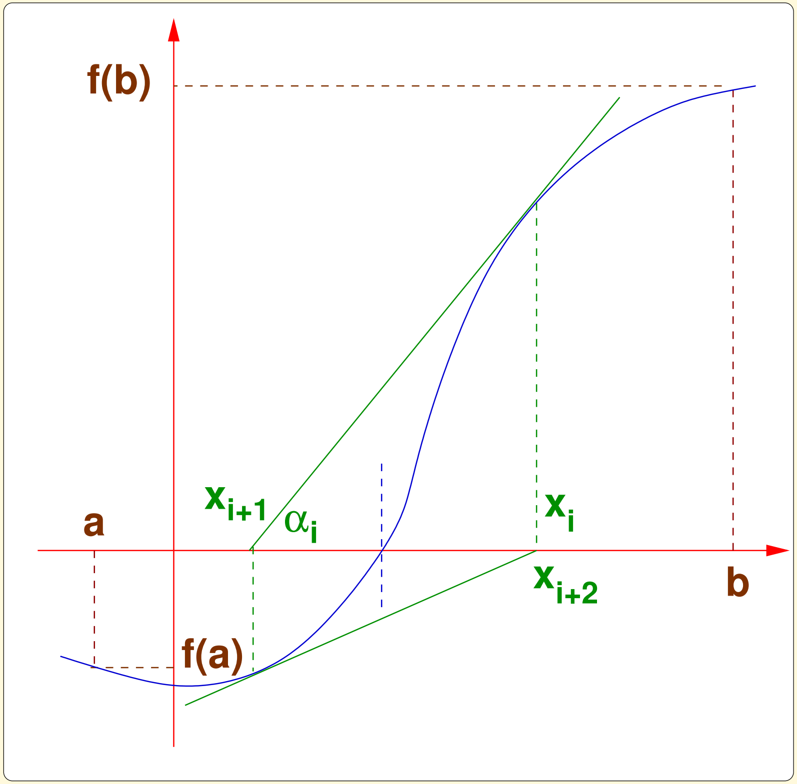
$$x_0 \in [a, b]$$

and

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

What can go wrong!-1

Oscillations!

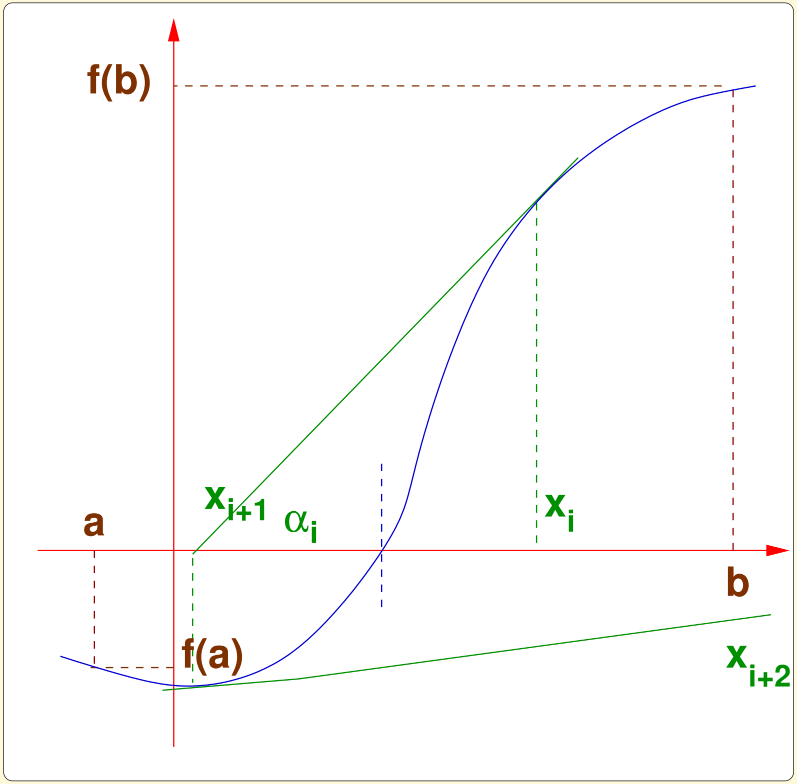


What can go wrong!-2

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.

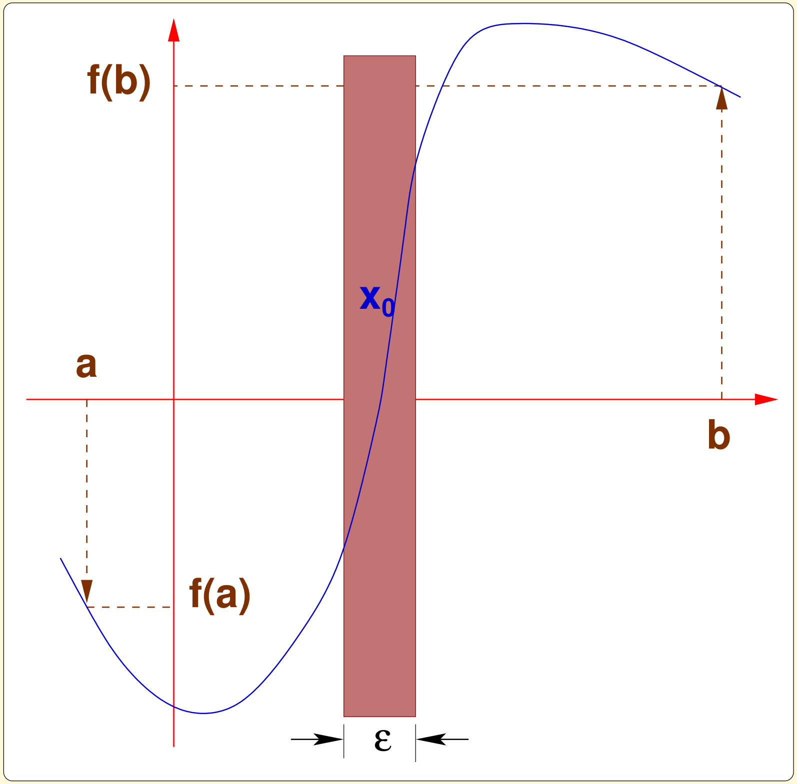
What can go wrong!-2

Interval bounds error!



What can go wrong!-3

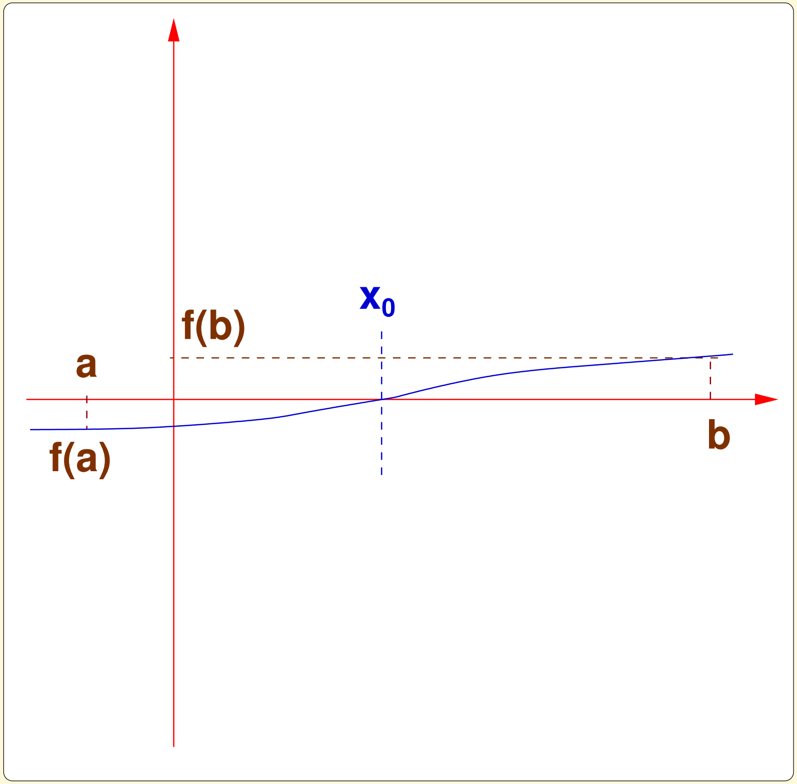
The function may be too steep



for the available precision.

What can go wrong!-4

Or too shallow!



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 ε

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

$$\boxed{x^n = c}$$

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

$$\begin{aligned} f(x) &= x^n - c \\ f'(x) &= nx^{n-1} \end{aligned}$$

with $[a, b] = [0, c]$ or $[0, \sqrt{c}]$ and an appropriately chosen ε.

What's it good for? 2

Finding roots of polynomials.

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

$$f'(x) = \sum_{i=1}^{n} i c_i x^{i-1}$$

and

- an appropriately chosen ε,
- an appropriately chosen $[a, b]$ with one of the limits possibly being c_0.

$newton$: Computation

$$
\begin{array}{ll}
 & newton(f, f', a, b, \varepsilon, x_0) \\
\rightsquigarrow & newton(f, f', a, b, \varepsilon, x_1) \\
\rightsquigarrow & newton(f, f', a, b, \varepsilon, x_2) \\
\rightsquigarrow & newton(f, f', a, b, \varepsilon, x_3) \\
\vdots & \vdots \qquad\qquad\qquad\qquad \vdots \\
\rightsquigarrow & newton(f, f', a, b, \varepsilon, x_n) \\
\rightsquigarrow & x_n
\end{array}
$$

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 $h(1)$

$$
\begin{array}{rl|rl}
 & h(1) & & h(1) \\
\rightsquigarrow & f(1, g(1)) & \rightsquigarrow & f(1, g(1)) \\
\rightsquigarrow & g(1) & \rightsquigarrow & f(1, (g(0)+1)) \\
\rightsquigarrow & g(0)+1 & \rightsquigarrow & f(1, (0+1)) \\
\rightsquigarrow & 0+1 & \rightsquigarrow & f(1, 1) \\
\rightsquigarrow & 1 & \rightsquigarrow & 1
\end{array}
$$

Two Computations of $h(-1)$

$$
\begin{array}{ll|ll}
 & h(-1) & & h(-1) \\
\leadsto & f(-1, g(-1)) & \leadsto & f(-1, g(-1)) \\
\leadsto & 0 & \leadsto & f(-1, (g(-2) + 1)) \\
 & \text{DONE!} & \leadsto & f(-1, ((g(-3) + 1) + 1 \\
 & & \leadsto & \ldots \\
 & & \leadsto & \text{FOREVER!}
\end{array}
$$

Recursive Computations

- Newton's method
- Factorial
  - $factL$
  - $factR$

skip to nonlinear recursion
skip to Recursion Revisited

Recursion: Left

$$
\begin{aligned}
& factL(4) \\
\rightsquigarrow\ & (factL(4-1) * 4) \\
\rightsquigarrow\ & (factL(3) * 4) \\
\rightsquigarrow\ & ((factL(3-1) * 3) * 4) \\
\rightsquigarrow\ & ((factL(2) * 3) * 4) \\
\rightsquigarrow\ & (((factL(2-1) * 2) * 3) * 4) \\
\rightsquigarrow\ & \cdots
\end{aligned}
$$

Recursion: Right

$$
\begin{aligned}
& factR(4) \\
\rightsquigarrow\ & (4 * factR(4-1)) \\
\rightsquigarrow\ & (4 * factR(3)) \\
\rightsquigarrow\ & (4 * (3 * factR(3-1))) \\
\rightsquigarrow\ & (4 * (3 * factR(2))) \\
\rightsquigarrow\ & (4 * (3 * (2 * factR(2-1)))) \\
\rightsquigarrow\ & \cdots
\end{aligned}
$$

Recursion: Nonlinear

Fibonacci

$$
\begin{array}{ll}
 & fib(5) \\
\rightsquigarrow & fib(4) + fib(3) \\
\rightsquigarrow & (fib(3) + fib(2)) + fib(3) \\
\rightsquigarrow & ((fib(2) + fib(1)) + fib(2)) + fib(3) \\
\rightsquigarrow & (((fib(1) + fib(0)) + fib(1)) + fib(2)) + fib(3) \\
\rightsquigarrow & (((1 + fib(0)) + fib(1)) + fib(2)) + fib(3) \\
\rightsquigarrow & \cdots
\end{array}
$$

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

Recursion Revisited

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

Linear Recursion: Waxing

$$
\begin{array}{ll}
 & factL(4) \\
\rightsquigarrow & (factL(3) * 4) \\
\rightsquigarrow & ((factL(2) * 3) * 4) \\
\rightsquigarrow & (((factL(1) * 2) * 3) * 4) \\
\rightsquigarrow & ((((factL(0) * 1) * 2) * 3) * 4)
\end{array}
$$

contrast with newton

Recursion: Waning

$$\rightsquigarrow ((((1 * 1) * 2) * 3) * 4)$$
$$\rightsquigarrow (((1 * 2) * 3) * 4)$$
$$\rightsquigarrow ((2 * 3) * 4)$$
$$\rightsquigarrow (6 * 4)$$
$$\rightsquigarrow 24$$

contrast with newton

Nonlinear Recursions

Fibonacci

- Each computation of fib has its own waxing and waning
- There is still an "envelope" which shows waxing and waning.

Fibonacci: *contd*

$$\rightsquigarrow (((1+1)+fib(1))+fib(2))+fib(3)$$

$$\rightsquigarrow (2+fib(1))+fib(2))+fib(3)$$

$$\rightsquigarrow ((2+1)+fib(2))+fib(3)$$

$$\rightsquigarrow \cdots$$

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

Algorithm

- 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 $n > 0$, $\forall k : 0 \leq k \leq n : factL(k)$ terminates in $\propto k$ steps.

Induction step . Then $factL(n+1) = factL(n) * (n+1)$ is guaranteed to terminate in $\propto (n+1)$ steps, since $factL(n)$ does so in $\propto n$ steps.

back

Fibonacci: Termination

The proof is similar to that of $factL$.

Basis For $n = 0$ or $n = 1$ $fib(n) = 1$.

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

Induction Step Then since each of $fib(n)$ and $fib(n-1)$ is guaranteed to terminate in $\propto f(n)$ and $\propto f(n-1)$ steps $fib(n+1) = fib(n) + fib(n-1)$ is also guaranteed to terminate in $f(n+1) \propto f(n) + f(n-1)$ steps.

GCD computations

Euclidean GCD

$$
\begin{array}{rl}
 & gcd(12, 64) \\
\leadsto & gcd(64, 12) \\
\leadsto & gcd(12, 4) \\
\leadsto & gcd(4, 0) \\
\leadsto & 4
\end{array}
$$

Well-foundedness: GCD

Euclidean GCD

For $x, y > 0$, $0 \leq x \bmod y < y$. 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

- $fact$ arbitrarily extended to include negative integers. (since $w(n)$ no longer strictly non-negative)
- $fact(n) = fact(n{+}1)$ div $(n{+}1)$, even if n is non-negative (since $w(n)$ is no longer decreasing)

since the function is no longer well-founded.

Well-foundedness is inductive

But the induction variable is

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

Well-foundedness is inductive

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

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

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

$$ind(w(\vec{x})) = n - i$$

GCD: Well-foundedness

GCD

Well-founded function for gcd

$$w(a, b) = b$$

Newton: Well-foundedness

Newton's Method

Convergence condition

$$f(x_0), f(x_1), f(x_2), \cdots \longrightarrow 0$$

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 ε Hence

$$w(x) = \lfloor\frac{|f(x)|}{\varepsilon}\rfloor$$

Example: Zero

A peculiar way to define the $zero$ function

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

$w(x) = \lceil |x| \rceil$ is the well-founded function

Questions

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

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

The Collatz Problem

The Collatz Problem

Does the following algorithm terminate?

$$collatz(m) =$$

$$\begin{cases} 1 & \text{if } m \leq 1 \\ collatz(m \text{ div } 2) & \text{if } m \text{ is even} \\ collatz(3 * m + 1) & \text{otherwise} \end{cases}$$

Unproven Claim. $collatz(m) \leadsto 1$ for all m.

Questions

Q: what other uses can well-founded functions be put to?

A: They can be used to estimate the complexity of your algorithm in *order of magnitude* terms.

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

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

Space Complexity

What is the space complexity of

- Newton's method
- Euclidean GCD
- Factorial
- Fibonacci

Newton & 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

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

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

$$\boxed{S_{gcd(a,b)} \propto 1}$$

A similar analysis and result holds for $newton$

Factorial: Space

$factL$

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

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

$$\boxed{S_{factL(n)} \propto n}.$$

A similar analysis and result holds for $factR$.

Fibonacci: Space

Fibonacci

$$S_{fib(n)} = \begin{cases} 1 & \text{if } n \leq 1 \\ S_{fib(n-1)} + S_{fib(n-2)} & \text{if } n > 1 \end{cases}$$

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

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

4.2. Efficiency Measures and Speed-ups

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

$isqrt$: Space

Integer Sqrt shrink

$S_{isqrt(n)} = S_{shrink(n,0,n)}$ for large n.

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

Assuming 1 unit of space for output.
By induction on $||[l, u]||$

$$\boxed{S_{isqrt(n)} = S_{shrink(n,0,n)} \propto 1}$$

Time Complexity

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

- Integer sqrt
- Factorial
- Fibonacci

forward

isqrt: Time Complexity

Integer Sqrt
shrink

Assume condition-checking (along with $+1$ or -1) takes a unit of time. Then $T_{shrink(n,l,u)} =$

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

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

$$\boxed{T_{isqrt(n)} = T_{shrink(n,0,n)} \propto n}$$

shrink

Assume condition-checking (along with $(l+u)$ $\mathtt{div}$ 2) takes a unit of time. Then $T_{shrink2(n,l,u)} =$

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

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

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

$$\boxed{T_{isqrt2(n)} \propto \lceil \log_2 n \rceil}$$

$shrink$ vs. $shrink2$: Times

$shrink$ $shrink2$

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

$factL$

Here we assume multiplication takes unit time.

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

Then

$$\boxed{T_{factL(n)} \propto n}$$

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

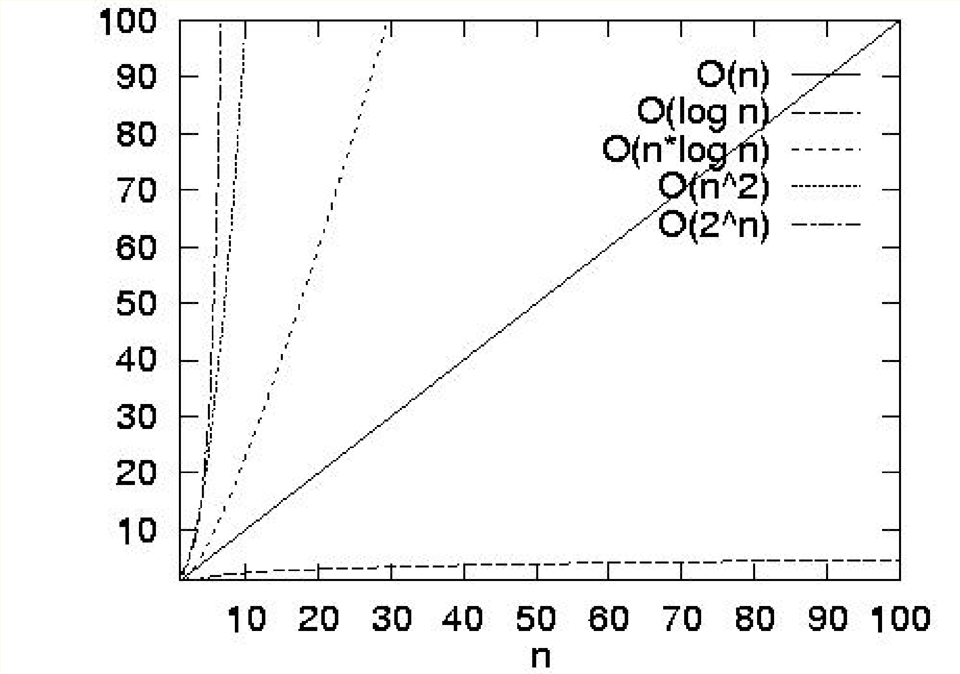
$$\boxed{2^{n-2} < T_{fib(n)} \leq 2^{n-1}}$$

Comparative Complexity

Algorithm	Space	Time
$isqrt(n)$	$O(1)$	$O(n)$
$isqrt2(n)$	$O(1)$	$O(\log_2 n)$
$factL(n)$	$O(n)$	$O(n)$
$fib(n)$	$O(2^n)$	$O(2^n)$

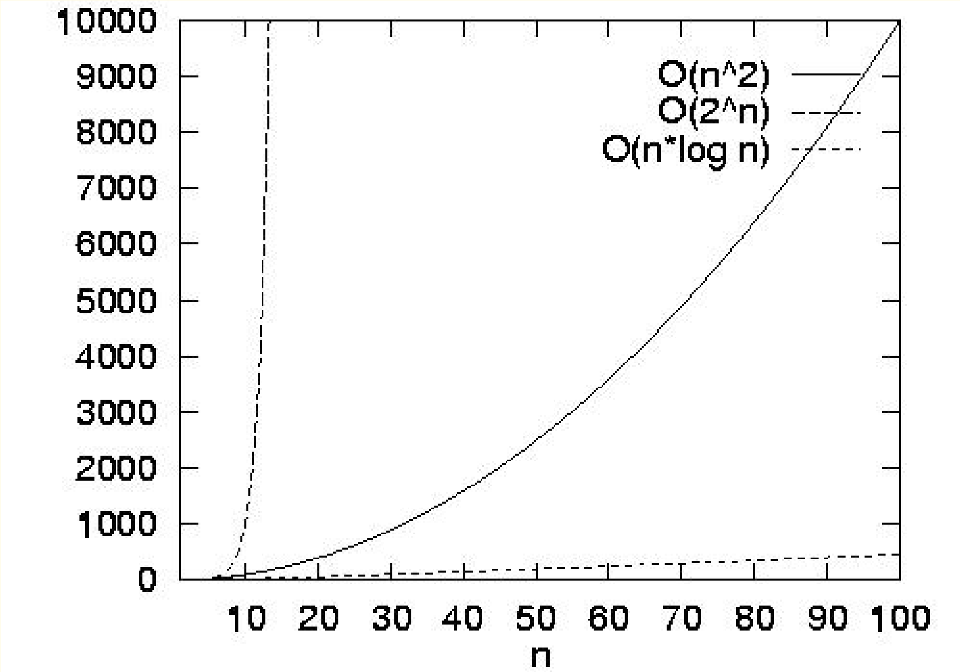
Comparisons

For smaller values



Comparisons

For large values



Efficiency Measures: Time

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

Efficiency Measures: Space

Similarly an algorithm is asymptotically more space efficient than another if its space complexity is bounded by a function that has a *slower* growth rate.

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. $isqrt2$ vs. $isqrt$ 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

$$
\begin{array}{ll}
 & gcd(a_0, b_0) \\
 & \text{compute } a_1, b_1 \\
\rightsquigarrow & gcd(a_1, b_1) \\
 & \text{compute } a_2, b_2 \\
\rightsquigarrow & gcd(a_2, b_2) \\
\rightsquigarrow & \cdots \\
\rightsquigarrow & gcd(a_n, b_n) \\
\rightsquigarrow & a_n
\end{array}
$$

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

$factL$ Waxing $factL$ 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

Factorial

$$factL2(n) = \begin{cases} \perp & \text{if } n < 0 \\ 1 & \text{if } n = 0 \\ factL\_tr(n, 1) & \text{otherwise} \end{cases}$$

where

$$factL\_tr(n, p) =$$

$$\begin{cases} p & \text{if } n = 0 \\ factL\_tr(n-1, np) & \text{otherwise} \end{cases}$$

A Computation

$$
\begin{array}{ll}
 & factL2(4) \\
\leadsto & factL\_tr(4,1) \\
\leadsto & factL\_tr(3,4) \\
\leadsto & factL\_tr(2,12) \\
\leadsto & factL\_tr(1,24) \\
\leadsto & factL\_tr(0,24) \\
\leadsto & 24
\end{array}
$$

Reminiscent of gcd and newton!

Factorial: Issues

- Correctness: Prove (by induction on n) that for all $n \geq 0$, $factL2(n) = n!$.
- termination: Prove by induction on n that every computation of $factL2$ terminates.
- Space complexity: Prove that $S_{factL2(n)} = O(1)$ (as against $S_{factL(n)} \propto n$).
- Time complexity: Prove that $T_{factL2(n)} = O(n)$

Complexity table

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

$$fibTR(n) =$$

$$\begin{cases} \perp & \text{if } n < 0 \\ 1 & \text{if } 0 \leq n \leq 1 \\ fib\_iter(n, 1, 1, 1) & \text{otherwise} \end{cases}$$

where

$$fib\_iter(n, a, b, m) =$$

$$\begin{cases} b & \text{if } m \geq n \\ fib\_iter(n, b, a+b, m+1) & \text{otherwise} \end{cases}$$

fibTR: SML

```
local
   fun fib_iter (n, a, b, m) =
 (* fib (m) = b ,fib (m-1) = a *)
   if m >= n then b
   else fib_iter (n, b, a+b, m+1);
in
fun fibTR (n) =
   if n < 0 then raise negativeArgumen
   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

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: $factL$ vs. $factL2$ vs.
- Fibonacci vs. Fibonacci: fib vs. $fibTR$

Comparisons

Algorithm	Space	Time
$isqrt(n)$	$O(1)$	$O(n)$
$isqrt2(n)$	$O(1)$	$O(\log_2 n)$
$factL(n)$	$O(n)$	$O(n)$
$factL2(n)$	$O(1)$	$O(n)$
$fib(n)$	$O(2^n)$	$O(2^n)$
$fibTR(n)$	$O(1)$	$O(n)$

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

$$\boxed{f(\vec{x}) = A(\vec{x})}$$

  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

$$\boxed{A(\vec{x}) = B(\vec{x})}$$

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* $n \geq 0$,

$$\boxed{factL2(n) = n!}$$

Proof: For $n = 0$, it is clear that $factL2(0) = 1 = 0!$. For $n > 0$, $factL2(n) = factL\_tr(n, 1)$. The proof is *done* provided we can show that $factL\_tr(n, 1) = n!$. □

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 $factL2$

- 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 $factL2$

Lemma 9 *For all* $n \geq 0$ *and* p

$$\boxed{factL\_tr(n, p) = (n!)p}$$

Proof: By induction on n. $\square$

Back to theorem

Invariance: Example

$factL2$

$$
\begin{aligned}
& factL\_tr(4, 7) \\
\rightsquigarrow\ & factL\_tr(3, 28) \\
\rightsquigarrow\ & factL\_tr(2, 84) \\
\rightsquigarrow\ & factL\_tr(1, 168) \\
\rightsquigarrow\ & factL\_tr(0, 168) \\
\rightsquigarrow\ & 168
\end{aligned}
$$

Contrast with a $factL2(4)$ computation

Invariance: Example

$factL2$

So what exactly *is* invariant?

$$
\begin{array}{llcl}
 & factL\_tr(4,7) & 168 = & 4! \times 7 \\
\leadsto & factL\_tr(3,28) & 168 = & 3! \times 28 \\
\leadsto & factL\_tr(2,84) & 168 = & 2! \times 84 \\
\leadsto & factL\_tr(1,168) & 168 = & 1! \times 168 \\
\leadsto & factL\_tr(0,168) & 168 = & 0! \times 168 \\
\leadsto & 168 & &
\end{array}
$$

Proof

Basis For $n = 0$, $factL\_tr(0, p) = p = (0!)p$.

Induction hypothesis (IH) For all k, $0 < k \leq n$,

$$\boxed{factL\_tr(k, p) = p = (k!)p}$$

Induction Step

$$\begin{aligned} & factL\_tr(n+1, p) \\ = \; & factL\_tr(n, (n+1)p) \\ = \; & (n!)(n+1)p \qquad (IH) \\ = \; & (n+1)!p \end{aligned}$$

Back to lemma

Invariance Lemma: fib_iter

fib_iter $\mathbf{F}$

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

$$\boxed{\text{INV} : fib\_iter(n, a, b, m) = \mathbf{F}(n)}$$

.

Proof: By induction on $k = n - m$ $\square$

Proof

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

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 $n - m = k + 1$, $\mathbf{F}(m) = b$ and $\mathbf{F}(m-1) = a$. Then $\mathbf{F}(m+1) = a + b$ and

$$\begin{aligned} & fib\_iter(n, a, b, m) \\ = \; & fib\_iter(n, b, a+b, m+1) \\ = \; & \mathbf{F}(n) \qquad (IH) \end{aligned}$$

Correctness: Fibonacci

$fibTR$ Fibonacci

Theorem 11 *For all* $n \geq 0$, $fibTR(n) = \mathbf{F}(n)$.

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

Variants & Invariants

$factL2$

$$factL3(n) = \begin{cases} \perp & \text{if } n < 0 \\ 1 & \text{if } n = 0 \\ factL\_tr2(n, 1, 1) & \text{else} \end{cases}$$

where

Variants & Invariants

$factL2$

$$factL\_tr2(n, p, m) =$$

$$\begin{cases} p & \text{if } n = m \\ factL\_tr2(n, (m+1)p, m+1) & \text{else} \end{cases}$$

$$\boxed{factL\_tr2(n, p, m) = (m!)p}$$

for all $1 \leq m \leq n$.

More Invariants

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

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

- $shrink2$
For all $n > 0$, l, u, if $[l, u] \subseteq [0, n]$,

$$\boxed{m = \lfloor(l + u)/2\rfloor \text{ and } l \leq \lfloor\sqrt{n}\rfloor \leq u}$$

Fast Powering 1

power2

$$power3(x, n) =$$

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

where

Fast Powering 2

power2

$$powerTR(x, n, p) =$$

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

where $even(n) \iff n \bmod 2 = 0$.

$$\boxed{powerTR(x, n, p) = x^n p}$$

Root Finding: Bisection

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

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

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

Recap: Tail Recursion

- Asymptotic Complexity:

  Time Linear

  Space Constant

- Correctness: Capture the algorithm through

  Invariant Invariance Lemma

  Bound function Proof by induction

Examples: Invariants

$factL\_tr2$

$shrink$ **&** $shrink2$

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

$$\boxed{m = \lfloor (l+u)/2 \rfloor \text{ and } l \leq \lfloor \sqrt{n} \rfloor \leq u}$$

Fast Powering

$$\boxed{powerTR(x, n, p) = x^n p}$$

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.

```
- 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: divmod

```
Standard ML of New Jersey, ...
- fun divmod (a, b) =
      (a div b, a mod b);
val divmod =
fn : int * int -> int * int
- val dm = divmod (24,9);
val dm = (2,6) : int * int
- #1 dm;
val it = 2 : int
- #2 dm;
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 `(`, `,` 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

$$\boxed{\mathbf{a} = (\#1\ \mathbf{a}, \#2\ \mathbf{a}, \ldots, \#n\ \mathbf{a})}$$

Lists

An α $list$ represents a sequence of elements of a given type α.
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: nil or $[]$
- Nonempty lists: Given a nonempty list L

$$\boxed{L = [1, 2, 3, 4]}$$

head : $hd : \alpha List \rightarrow \alpha$

$$\boxed{hd(L) = 1}$$

tail : $tl : \alpha List \rightarrow \alpha List$

$$\boxed{tl(L) = [2, 3, 4]}$$

List Operations: $cons$

- $L = [1, 2, 3, 4]$

  cons : $cons : \alpha \times \alpha List \rightarrow \alpha List$

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

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

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

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

$$poly0(L, x) =$$

$$\begin{cases} 0 & \text{if } L = nil \\ hx^n + poly0(T, x) & \text{if } L = h :: T \end{cases}$$

where $n = |L| - 1$.

```
fun poly0 ([], x) = 0.0
  | poly0 ((h::T), x) =
    h*power(x, n)+poly0(T, x)
```

Complexity of $poly0$

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.

$$poly1(L, x) = poly\_TR(0, L, x)$$

where

$$poly\_TR(p, L, x) =$$

$$\begin{cases} p & \text{if } L = nil \\ poly\_TR(px + h, T, x) & \text{if } L = h :: T \end{cases}$$

$poly1$ 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

```
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;
```

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.

revpoly0

$$revpoly0(L, x) =$$

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

```
fun revpoly0 ([], x) = 0.0
  | revpoly0 ((h::T), x) =
      h + x * revpoly0 (T, x)
```

Tail Recursive

$$revpoly1(L, x) = revpoly1\_TR(L, x, 1, x)$$

where

$$revpoly1\_TR(L, x, p, s) =$$

$$\begin{cases} s & \text{if } L = nil \\ revpoly1(T, x, px, s + ph) & \text{if } L = h :: T \end{cases}$$

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 $revpoly1$

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 <u>uniquely (upto order)</u> 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 $\leq \lfloor \sqrt{n} \rfloor$.

$$primesUpto(n)$$

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

where

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

bound function $\boxed{n - m}$

Invariant

$$\boxed{2 < m \leq n \wedge odd(m)}$$

implies

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

and

$$\boxed{\forall q : p_{k-1} < q < m : composite(q)}$$

$$generateFrom$$

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

$$\begin{cases} P & \text{if } m > n \\ generateFrom\ (((k, m) :: P), m + 2, n, k + 1) & \text{elseif } pwrt \\ generateFrom\ (P, m + 2, n, k) & \text{else} \end{cases}$$

where $pwrt = primeWRT(m, P)$

$primeWRT(m, P)$

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 $\leq \sqrt{n}$

$$primeWRT(m, P)$$

bound function $length(P)$

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

$$primeWRT$$

$$primeWRT(m, P) =$$

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

where

$$(i, h) = hd(P)$$

for some $i > 0$

Density of Primes

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

n	$\pi(n)$	%
100	25	25.00%
1000	168	16.80%
10000	1229	12.29%
100000	9592	9.59%
1000000	78,498	7.85%
10000000	664579	6.65%
100000000	5761455	5.76%

The Prime Number Theorem

$$\boxed{\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1}$$

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

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

from David Burton: *Elementary Number Theory*.

Complexity

function	calls
$primesUpto$	1
$generateFrom$	$n/2$
$primeWRT$	$\sum_{m=3,odd(m)}^{n} \pi(m)$

Diagnosis

For each $m \leq n$,

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

ascending-order

5.2. Compound Data & Lists

Compound Data

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

Recap: Tuples

formation : Cartesian products of types

selection : Selection of individual components

equality : Equality checking

equality errors : Equality errors

forward to Lists

Tuple: Formation

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

back

Tuples: Selection

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

back

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
```

back

Tuples: Equality errors

```
- ("arun", (1, true))
= ("arun", 1, true);
stdIn:1.1-29.39 Error: operator and o
  operator domain: (string * (int * b
  operand:         (string * (int * b
  in expression:
    ("arun",(1,true))
  = ("arun",1,true)
- ("arun", (1, true))
= (("arun", 1), true);
stdIn:1.1-29.39 Error: operator and o
```

Lists: Recap

formation : Sequence $\alpha\ List$

selection : Selection of individual components

new lists : Making new lists from old

Lists: Append

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

$cons$ vs. $@$

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

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

Lists of Functions

```
- fun add1 x = x+1;
val add1 = fn : int -> int
- fun add2 x = x + 2;
val add2 = fn : int -> int
- fun add3 x = x + 3;
val add3 = fn : int -> int
```

Lists of Functions

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

IIT Delhi
S. Arun-Kumar, CSE
Title Page
Contents
Page 442 of 709
Go Back
Full Screen
Close
Quit

Arithmetic Sequences

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

function	calls	Order
$AS1$	n	$O(n)$
$@$	n	$O(n)$
$::$	$\sum_{i=0}^{n} i$	$O(n^2)$

Tail Recursion

$$AS2(a,d,n) =$$

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

where

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

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

Tail Recursion: Invariant

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

and bound function

$$\boxed{n - k}$$

$$AS2\_iter(a, d, n, k, L) =$$

$$\begin{cases} L & \text{if } k \geq n \\ AS2\_iter(a, d, n, k+1 \\ L@[a + k * d]) & \text{else} \end{cases}$$

Tail Recursion: Complexity

function	calls	Order
$AS2$	1	
$AS2\_iter$	n	$O(n)$
$@$	n	$O(n)$
$::$	$\sum_{i=0}^{n} i$	$O(n^2)$

So tail recursion simply doesn't help!

Another Tail Recursion: $AS3$

$$AS3(a,d,n) =$$

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

where

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

$$\boxed{INV3 : L = (a + (n-1) * d) :: \cdots ::}$$

$$\boxed{\cdots :: (a + (n_0 - 1) * d) :: L_0}$$

Another Tail Recursion: $AS3\_iter$

$$\boxed{INV3 : L = (a + (n-1) * d) :: \cdots}$$

$$\boxed{\cdots :: (a + (n_0 - 1) * d) :: L_0}$$

and bound function n,

$$AS3\_iter(a, d, n, L) =$$

$$\begin{cases} L & \text{if } n \leq 0 \\ AS3\_iter(a, d, n-1, & \text{else} \\ (a + (n-1) * d) :: L) & \end{cases}$$

AS3: Complexity

function	calls	Order
$AS3$	1	
$AS3\_iter$	n	$O(n)$
$@$	0	
$::$	$\sum_{i=0}^{n} 1$	$O(n)$

Generating Primes: 2

composites

- $primesUpto$
- invariant
- $generateFrom$

$$primes2Upto(n)$$

$$primes2Upto(n) =$$

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

where

$$generate2From(P, m, n, k)$$

bound function $\boxed{n - m}$

Invariant

$$\boxed{2 < m \leq n \wedge odd(m)}$$

implies

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

and

$$\boxed{\forall q : p_{k-1} < q < m : composite(q)}$$

$$generate2From$$

$$generate2From(P, m, n, k) =$$

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

where $pwrt = prime2WRT(m, P)$

$$prime2WRT(m, P)$$

bound function $length(P)$

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

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

$$prime2WRT$$

$$prime2WRT(m,P) =$$

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

where

$$(i,h) = hd(P)$$

for some $i > 0$

$primes2$: Complexity

function	calls
$primes2Upto$	1
$generate2From$	$n/2$
$prime2WRT$	$\sum_{m=3,odd(m)}^{n} \pi(\lfloor\sqrt{m}\rfloor)$

$primes2$: 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

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

Records: Destructors

```
#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:st
- 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

Back to Lists

- Every $L : \alpha\ List$ satisfies
$$\boxed{L = []}$$
XOR
$$\boxed{L = hd(L) :: tl(L)}$$
- Many functions on lists (L) are defined by induction on its length ($|L|$).

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

Lists: Correctness

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

Basis $|L| = 0$. Prove $\boxed{P([])}$

Induction hypothesis (IH) Assume for some $|T| = n > 0$, $\boxed{P(T)}$ holds.

Induction Step Prove $\boxed{P(h :: T)}$ 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

$$\boxed{P([])}$$

Induction hypothesis (IH) Assume

$$\boxed{P(T)}$$

Induction Step Prove

$$\boxed{P(h :: T)}$$

List-functions: $length$

$$\begin{cases} length\ [] & = 0 \\ length\ (h :: T) & = 1 + (length\ T) \end{cases}$$

List Functions: $search$

To determine whether x occurs in a list L

$$\begin{cases} search\ (x, []) & = false \\ search\ (x, h :: T) & = true \text{ if } x = h \\ search\ (x, h :: T) & = search(x, T) \text{ else} \end{cases}$$

List Functions: $search2$

Or even more conveniently

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

Time Complexity??

List Functions: $ordered$

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

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

Time Complexity??

List Functions: $insert$

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

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

Time Complexity??

List Functions: $reverse$

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

$$\begin{cases} reverse\ [] & = [] \\ reverse\ (h :: T) & = (reverse\ T)@[h] \end{cases}$$

Time Complexity?? $O(n^2)$

List Functions: $reverse2$

$$\begin{cases} reverse\ [] & = [] \\ reverse\ (h :: T) & = rev\ ((h :: T), []) \end{cases}$$

where

$$\begin{cases} rev\ ([], N) & = N \\ rev\ (h :: T, N) & = rev\ (T, h :: N) \end{cases}$$

Correctness ??

Time Complexity?? $O(n)$

List Functions: $merge$

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

$merge(L, M) = N$, where

$N = [0, 1, 3, 3, 4, 4, 5, 9, 10, 11]$

List Functions: $merge$

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

List Functions: $merge$ *contd.*

where

$$\begin{cases} L &= a :: S \\ M &= b :: T \end{cases}$$

ML: *merge*

```
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 $sort[10, 8, 3, 6, 9, 7, 4, 8, 1] = [1, 3, 4, 6, 7, 8, 8, 9, 10]$

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

Time Complexity??

Sorting by Merging

$$
\begin{cases}
msort\ [\,] & = [\,] \\
msort\ [a] & = [a] \\
msort\ L & = merge\ ((msort\ M), \\
 & \qquad (msort\ N))
\end{cases}
$$

where

$$(M, N) = split\ L$$

Sorting by Merging

where

$$\begin{cases} split\ [] & = ([], []) \\ split\ [a] & = ([a], []) \\ split\ (a :: b :: P) & = (a :: Left, b :: Right) \end{cases}$$

where

$$(Left, Right) = split\ P$$

Time Complexity??

Functions as Data

list of functions

- 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

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

$$\boxed{addc\ y\ x = x + y}$$

ML's response :

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

Contrast with ML's response

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

$addc$ is the curried version of the binary operation $+$.

Currying: Contd

$$f : (\alpha * \beta * \gamma) \rightarrow \delta$$

$$f_c : \alpha \rightarrow \beta \rightarrow \gamma \rightarrow \delta$$

$$f_c^1 : (\alpha * \beta) \rightarrow \gamma \rightarrow \delta$$

$$f_c^2 : \alpha \rightarrow (\beta * \gamma) \rightarrow \delta$$

Generalization

Then

- $addc1 \;\; = (addc\ 1)$: `int -> int`
- $addc2 \;\; = (addc\ 2)$: `int -> int`
- $addc3 \;\; = (addc\ 3)$: `int -> int`

and for any i,

$$\boxed{(addc\ i)\text{: }\texttt{int -> int}}$$

is the required function.

Generalization: 2

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

ML's response :

```
val list_adds = fn :
int -> (int -> int) list
```

Applying a list

addall

$$\begin{cases} applyl\ \ [\,]\ x & = [\,] \\ applyl\ \ (h :: T)\ x & = (h\ x) :: (applyl\ \ T\ x) \end{cases}$$

ML's response:

```
val applyl = fn :
('a -> 'b) list ->
'a -> 'b list
```

Trying it out

$$interval\ x\ n = applyl\ x\ (list\_adds\ n)$$

ML's response:

```
val interval = fn :
    int -> int -> int list
- interval 53 5;
val it = [54,55,56,57,58]
: int list
```

Associativity

- Application associates to the left.

$$\boxed{f\ x\ y = ((f\ x)\ y)}$$

- $\rightarrow$ associates to the right.

$$\boxed{\alpha \rightarrow \beta \rightarrow \gamma = \alpha \rightarrow (\beta \rightarrow \gamma)}$$

If $f : \alpha \rightarrow \beta \rightarrow \gamma \rightarrow \delta$
then $f\ \ a : \beta \rightarrow \gamma \rightarrow \delta$
and $f\ \ a\ \ b : \gamma \rightarrow \delta$
and $f\ \ a\ \ b\ \ c : \delta$

Apply to a list

$$\begin{cases} map\ f\ [] & = [] \\ map\ f\ (h :: T) & = (f\ h) :: (map\ f\ T) \end{cases}$$

```
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

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

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

IIT Delhi
S. Arun-Kumar, CSE
Title Page
Contents
Page 502 of 709
Go Back
Full Screen
Close
Quit

Further Generalization

Given

$$f : \alpha * \alpha \rightarrow \alpha$$

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

$$seq(f, a, d, n) =$$

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

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

Sequences

Arithmetic: $AS5(a, d, n) = seq(op+, a, d, n)$

Geometric: $GS1(a, r, n) = seq(op*, a, r, n)$

Harmonic: $HS1(a, d, n) = map\ reci\ (AS5(a, d, n))$

where
$reci\ x = 1.0/(real\ x)$ gives the reciprocal of a (non-zero) integer.

Efficient Generalization

Let's not use map repeatedly.

$seq2(f, g, a, d, n) =$

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

is the sequence of length n generated with a unary f, a binary g starting from $f(a)$.

Sequences: 2

- $AS6(a, d, n) = seq2(id, op+, a, d, n)$
- $GS2(a, r, n) = seq2(id, op*, a, r, n)$
- $HS2(a, d, n) = seq2(reci, op+, a, d, n)$

where $id\ x = x$ is the identity function.

More Generalizations

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

- arithmetic, geometric, harmonic sequences
- e^x, trigonometric functions 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 $succ$ function

$sum(f, succ, l, u) =$

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

Or Maybe . . . Products

Or may be interested in forming products of sequences.

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

$$prod(f, succ, l, u) =$$

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

Or Some Other $\otimes$

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

- $\otimes : (\alpha * \alpha) \longrightarrow \alpha$ is closed
- $\otimes$ is associative i.e.
  $$\boxed{a \otimes (b \otimes c) = (a \otimes b) \otimes c}$$
- $\otimes$ has an identity element e i.e
  $$\boxed{a \otimes e = a = e \otimes a}$$

$$\bigotimes_{i=l}^{u} f(l)$$

Other $\otimes$

Then if $f, succ : \alpha \rightarrow \alpha$

$ser(\otimes, f, succ, l, u) =$

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

Examples of $\otimes$, e

- $+$, 0 on integers and reals
- concatenation and the empty string on strings
- `andalso`, `true` on booleans
- `orelse`, `false` on booleans
- $+$, $\mathbf{0}$ on vectors and matrices
- $*$, $\mathbf{1}$ on vectors and matrices

6.2. Structured Data

Transpose of a Matrix

Map

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

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

where

Transpose: 0

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

```
[

]
```

Transpose: 10

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

[
]

Transpose: 01

[
[12 13]
[22 23]
[32 33]
[42 43]
]

[
[11 21 31 41]
]

Transpose: 20

[
[12 13]
[22 23]
[32 33]
[42 43]
]

[
[11 21 31 41]
]

Transpose: 02

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

IIT Delhi

S. Arun-Kumar, CSE

Title Page

Contents

Go Back

Full Screen

Close

Quit

Transpose: 30

[
[13]
[23]
[33]
[43]
]

[
[11 21 31 41]
[12 22 32 42]
]

Transpose: 03

[
[ ]
[ ]
[ ]
[ ]
]

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

$$trans$$

$$\begin{cases} trans\ [] & = [] \\ trans\ [] :: TL & = [] \\ trans\ LL & = (map\ hd\ LL) :: \\ & \quad (trans\ (map\ tl\ LL)) \end{cases}$$

and

$$is2DMatrix = \#1(dimensions\ L)$$

where

$$is2DMatrix$$

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

where $dimensions\ TL = (b, r, c)$
and $h = length\ H$

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 $\boxed{f(x) = 5, \text{ for all } x : \alpha}$ where $\boxed{f : \alpha \rightarrow \texttt{int}}$ is <u>not the same</u> as a value $\boxed{5 : int}$. Their types are different.

Data vs. Functions

Facilities	Functions	Data
primitive	operations	values
user-defined	functions	constructors
composition	application	alternative
recursion	recursion	recursion

Data vs. Functions: Recursion

Recursion
Basis
naming
composition
induction

Lists as Structured Data

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

Every α *list* is either

`nil` : (Basis, name)

`|` : or (alternative)

`cons` : constructed inductively from an element of type `'a` and another list of type `'a list` using the constructor `cons`

Constructors

- Inspired by the list constructors

  $nil : \alpha\ list$

  $cons : \alpha \times \alpha\ list \rightarrow \alpha\ list$

- combine heterogeneous types: α and $\alpha\ list$

- allows recursive definition by a form of induction

  Basis : $\boxed{nil}$

  Induction : $\boxed{cons}$

Shapes

A non-recursive data type

```
datatype shape =

    CIRCLE of real
  | RECTANGLE of real * real
  | TRIANGLE of
    real * real * real
```

Shapes: Triangle Inequality

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

Shapes: Area

```
exception notShape;

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

Shapes: Area

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

ML: Try out

```
- use "shapes.sml";
[opening shapes.sml]
datatype shape
  = CIRCLE of real
  | RECTANGLE of real * real
  | TRIANGLE of
    real * real * real
val isTriangle =
    fn : shape -> bool
exception notShape
val area = fn : shape -> real
```

ML: Try out (contd.)

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

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

Back to User defined types

Enumeration Types

- Enumeration types are non-recursive datatypes with
- 0-ary constructors

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

Back to User defined types

Recursive Data Types

- But the really interesting types are the recursive data types

Back to Lists

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

Correctness on lists

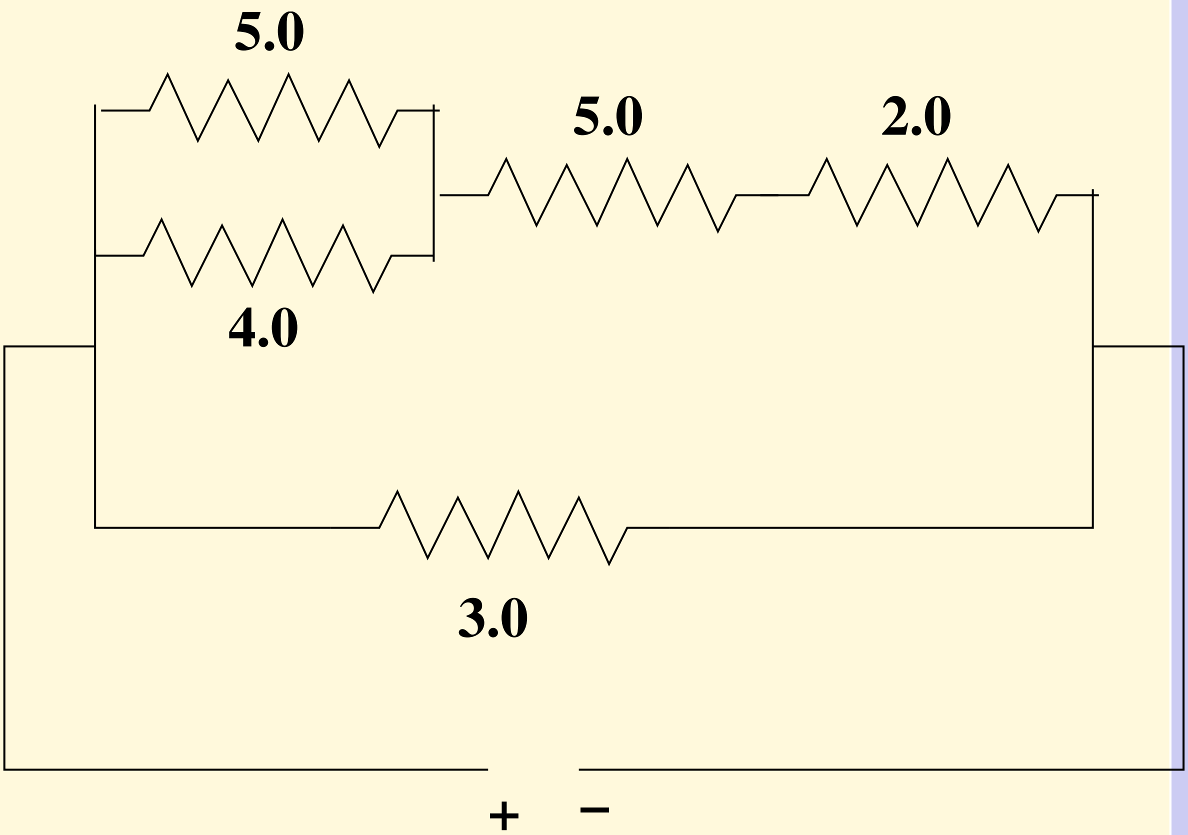
Resistors: Datatype

```
datatype resist =
    RES of real |
    SER of resist * resist |
    PAR of resist * resist
```

Resistors: Equivalent

```
fun value (RES (r)) = r
  | value (SER (R1, R2)) =
     value (R1) + value (R2)
  | value (PAR (R1, R2)) =
    let val r1 = value (R1);
        val r2 = value (R2)
    in  (r1*r2)/(r1+r2)
    end;
```

Resistors



Resistors: Example

```
val R = PAR(
            SER(
                PAR(
                    RES (5.0),
                    RES (4.0)
                   ),
                SER(
                    RES (5.0),
                    RES (2.0)
                   )
               ),
             RES(3.0)
            );
```

Resistors: ML session

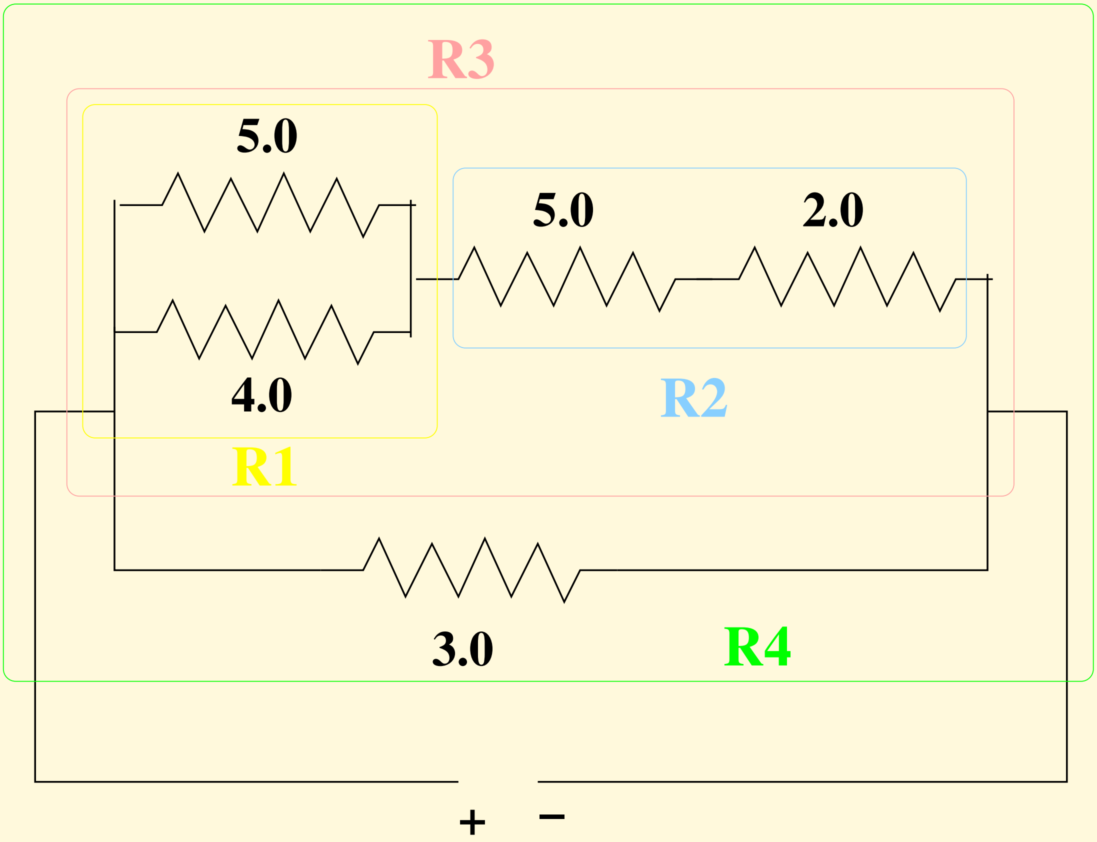
```
- use "resistors.sml";
[opening resistors.sml]
datatype resist = PAR of resist * res
                | RES of real
                | SER of resist * res
val value = fn : resist -> real
val R = PAR (SER (PAR #,SER #),RES 3
val it = () : unit
- value R;
val it = 2.26363636364 : real
-
```

6.3. User Defined Structured Data Types

User Defined Types

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

Resistors: Grouping



Resistors: In Pairs

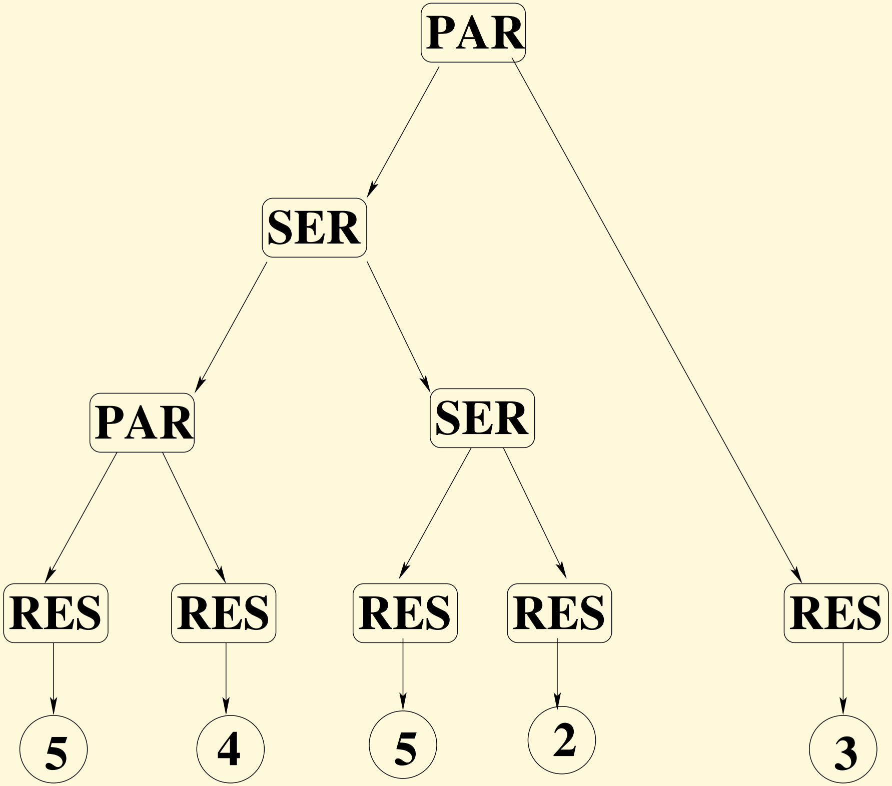
```
val R1 = PAR (RES 5.0,RES 4.0) : resi
val R2 = SER (RES 5.0,RES 2.0) : resi
val R3 = SER (R1, R2);
val R4 = PAR (R3, RES(3.0));
```

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

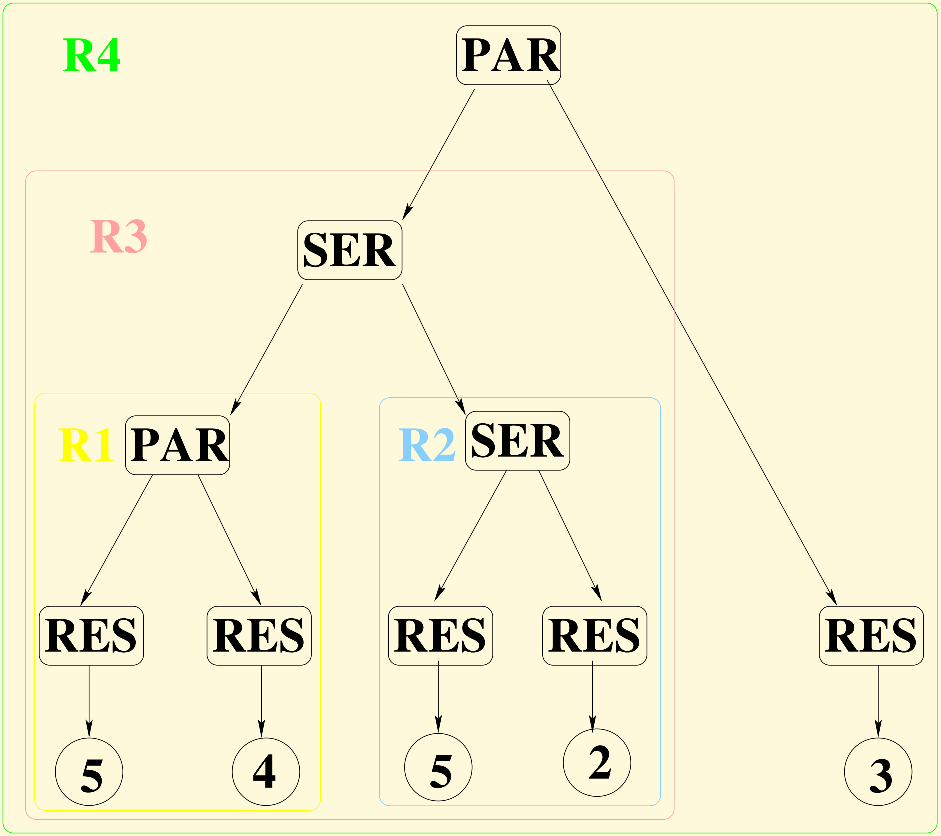
A resistance expression



Circuit Diagram

Resistance Expressions

A resistance expression



Circuit Diagram

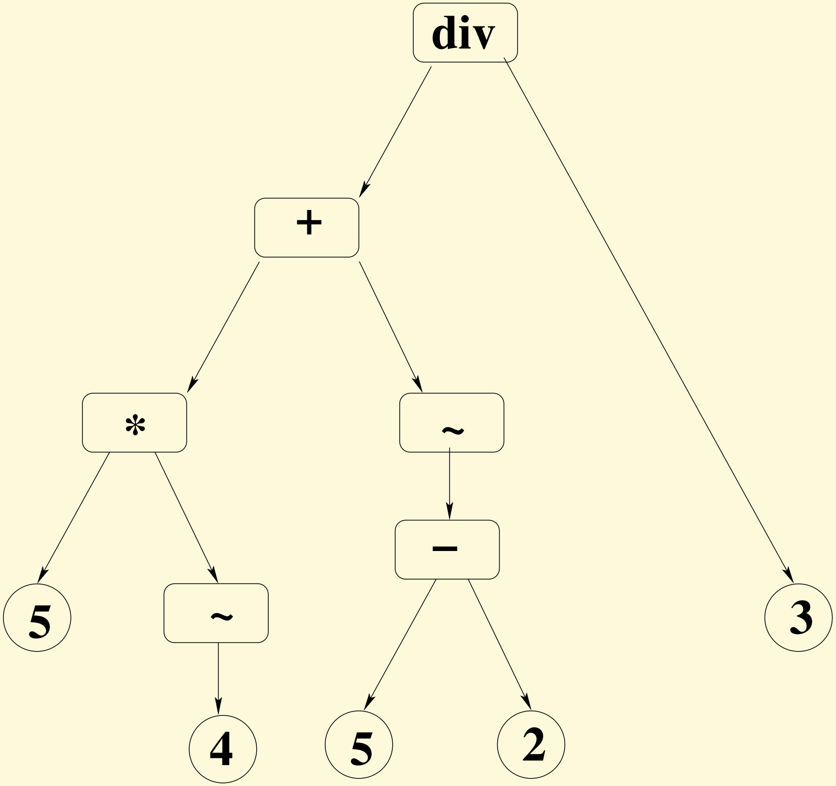
Arithmetic Expressions

ML arithmetic expressions:

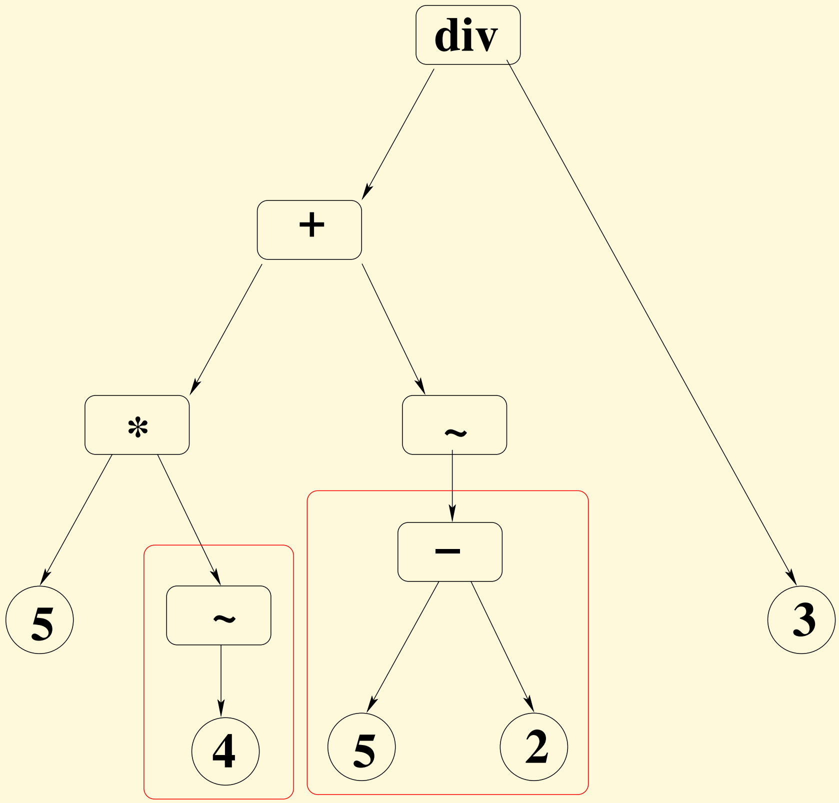
```
((5 * ~4) + ~(5 - 2)) div 3
```

are represented as trees

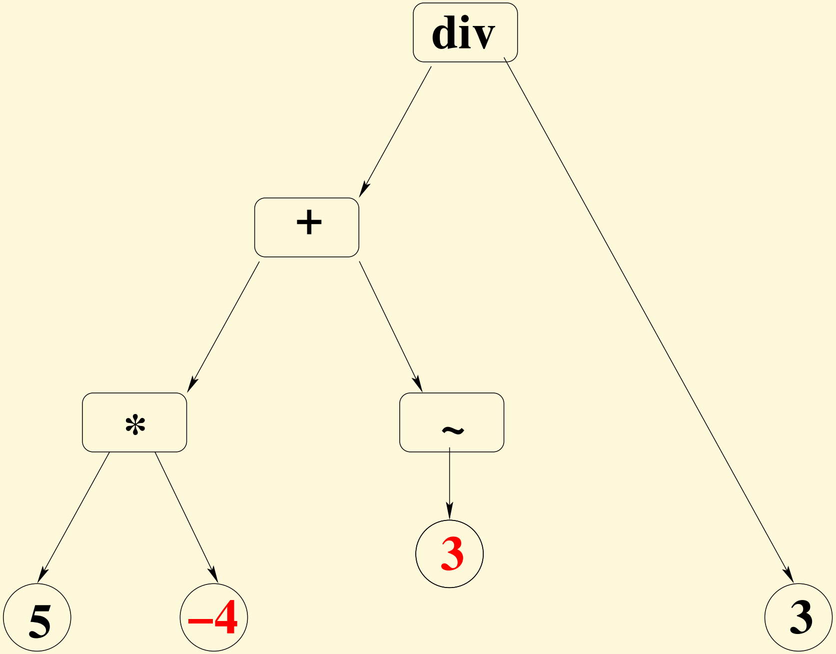
Arithmetic Expressions: 0



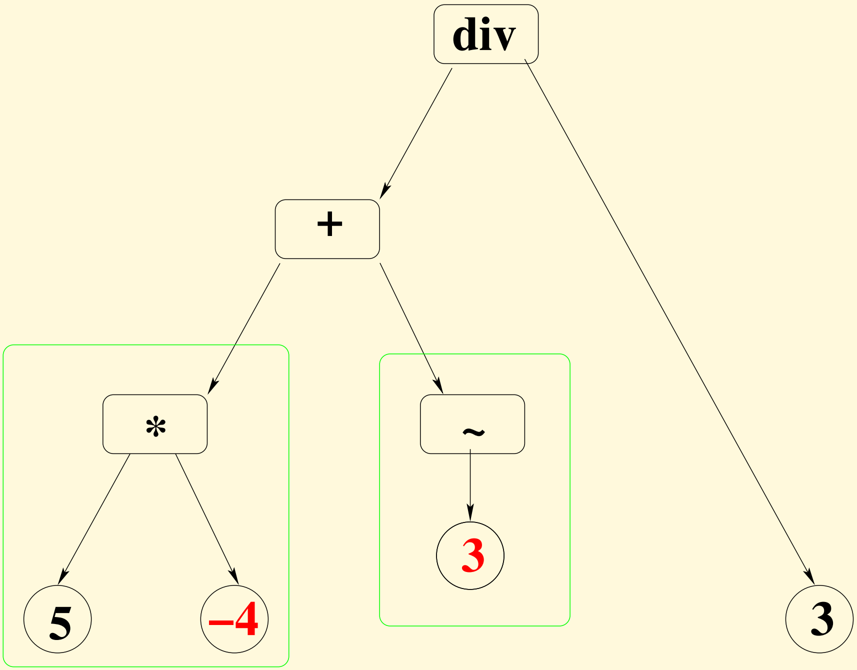
Arithmetic Expressions: 1



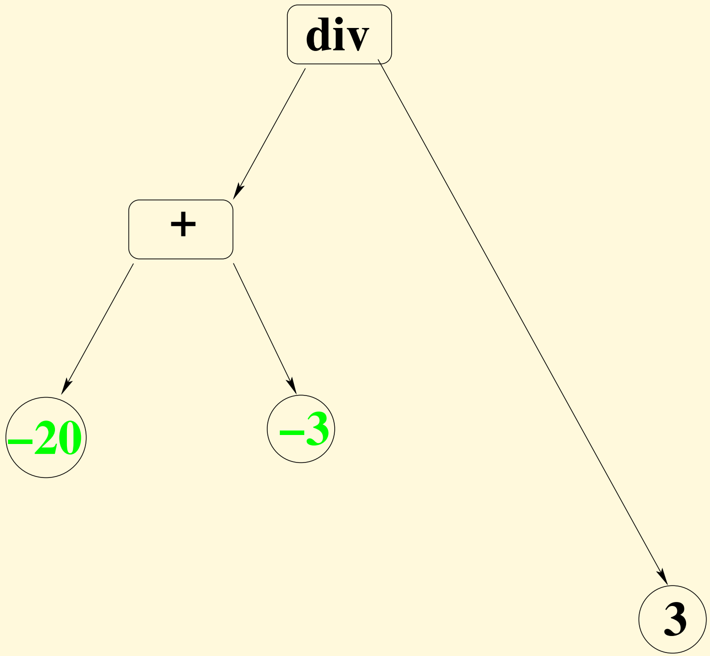
Arithmetic Expressions: 2



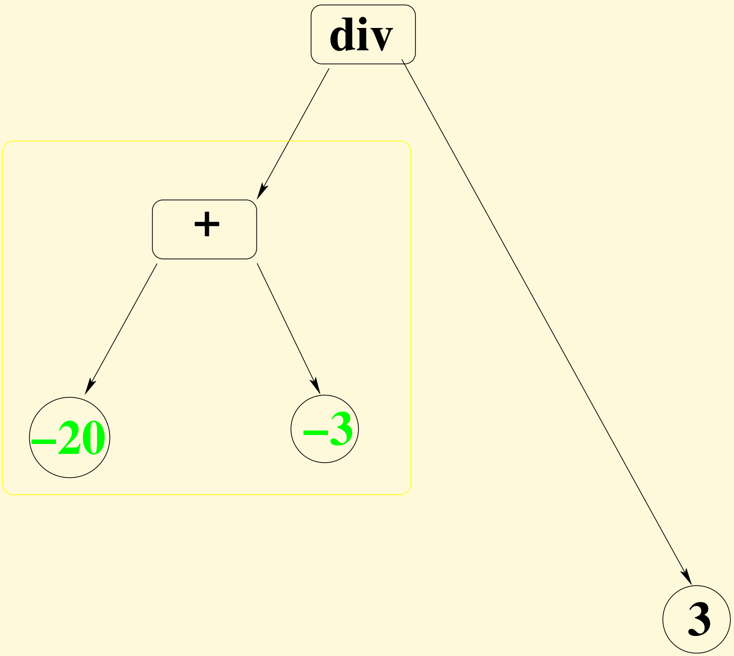
Arithmetic Expressions: 3



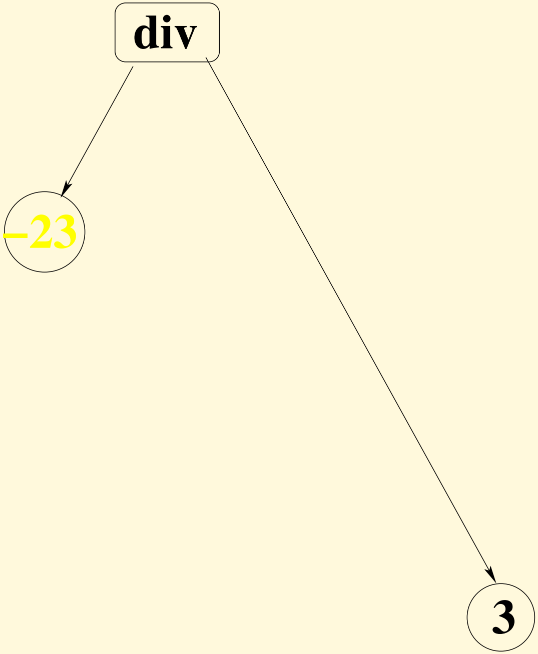
Arithmetic Expressions: 4



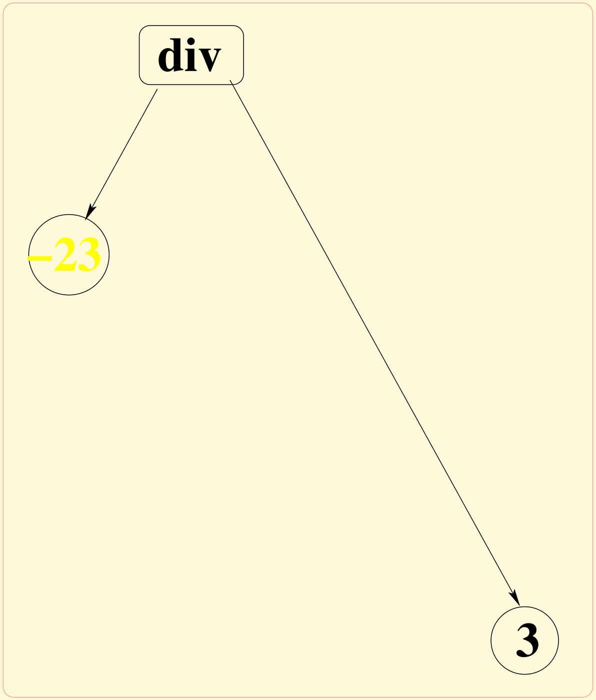
Arithmetic Expressions: 5



Arithmetic Expressions: 6



Arithmetic Expressions: 7



Arithmetic Expressions: 8

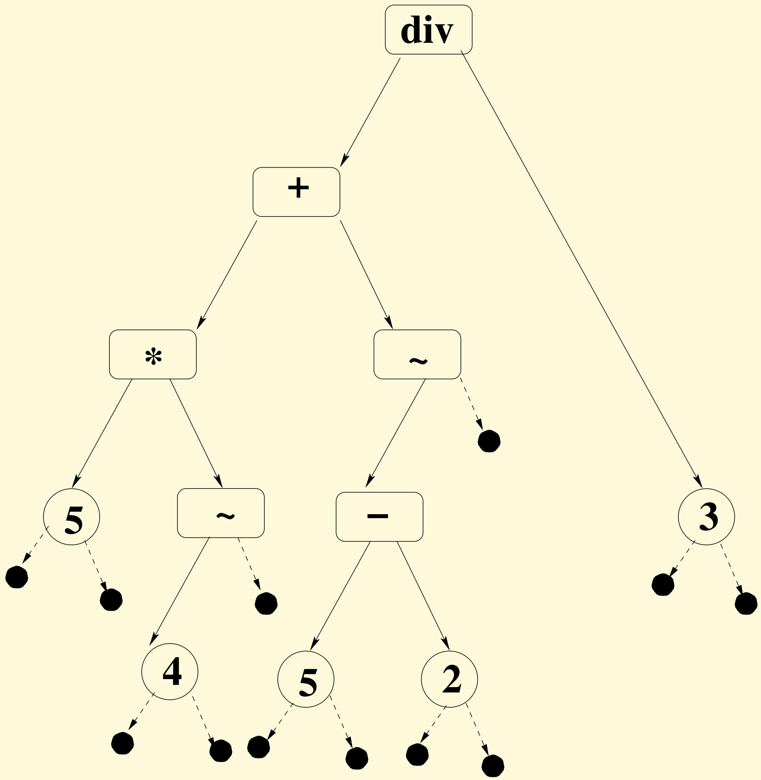
–8

Binary Trees

```
datatype 'a bintree =
         Empty |
         Node of 'a *
         'a bintree *
         'a bintree
```

Arithmetic Expressions: 0

Arithmetic Expressions



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 $\boxed{P(c)}$ for each non-recursive constructor c

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

Induction Step Prove $\boxed{P(r(T_1, \ldots, T_n))}$ for each recursive constructor r

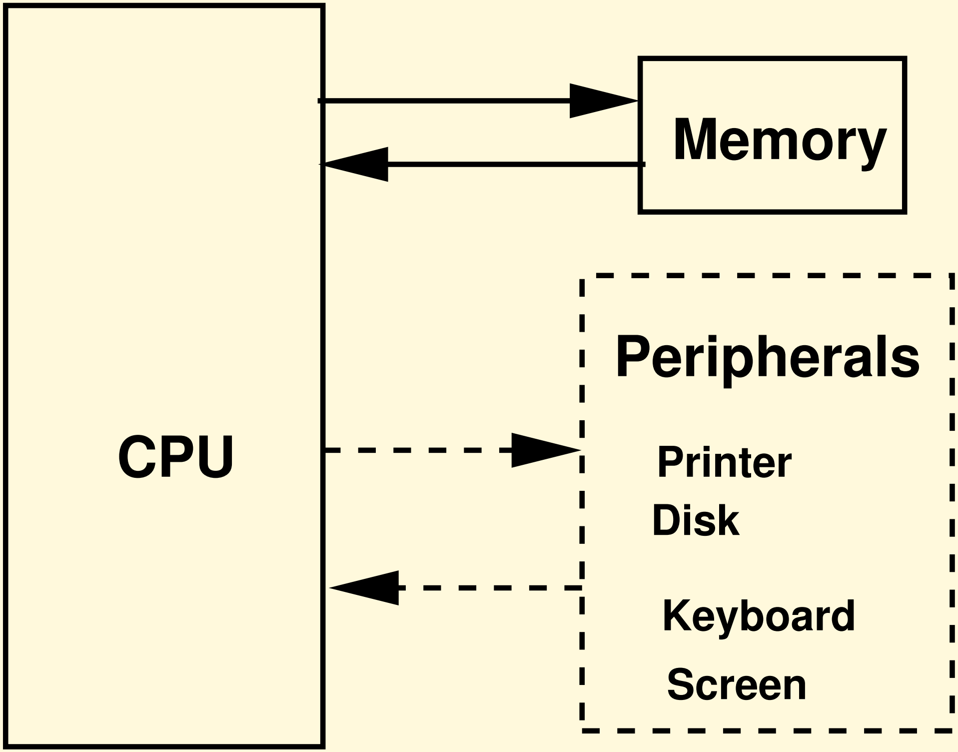
7. Imperative Programming: An Introduction

7.1. Introducing a Memory Model

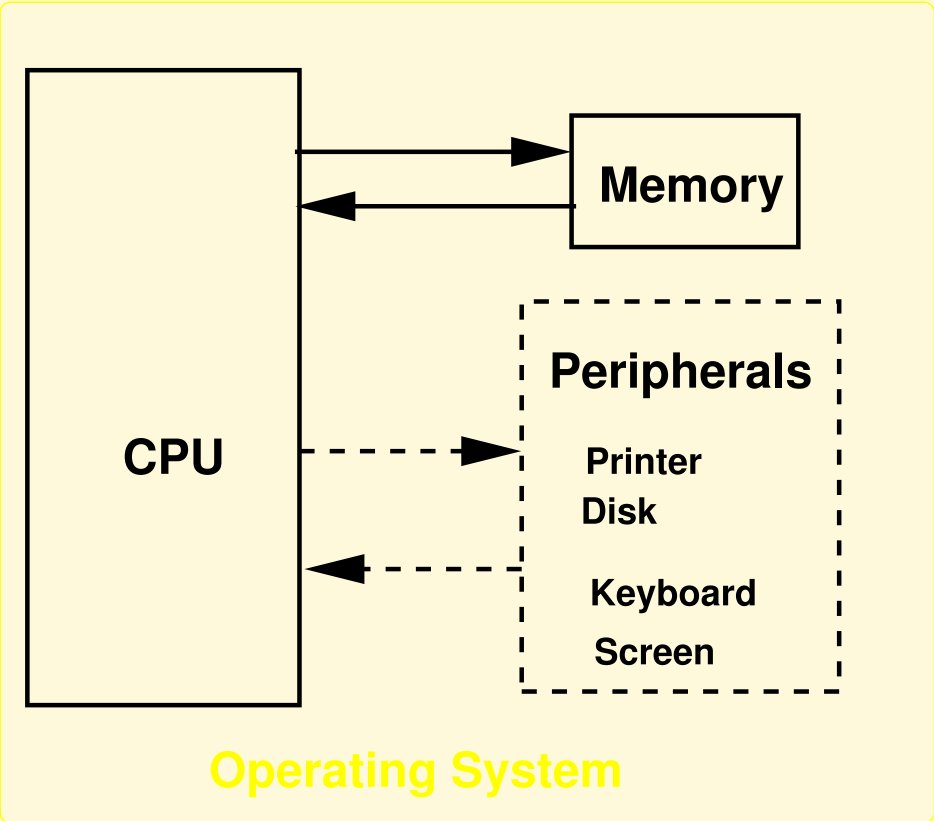
Summary: Functional Model

- 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)

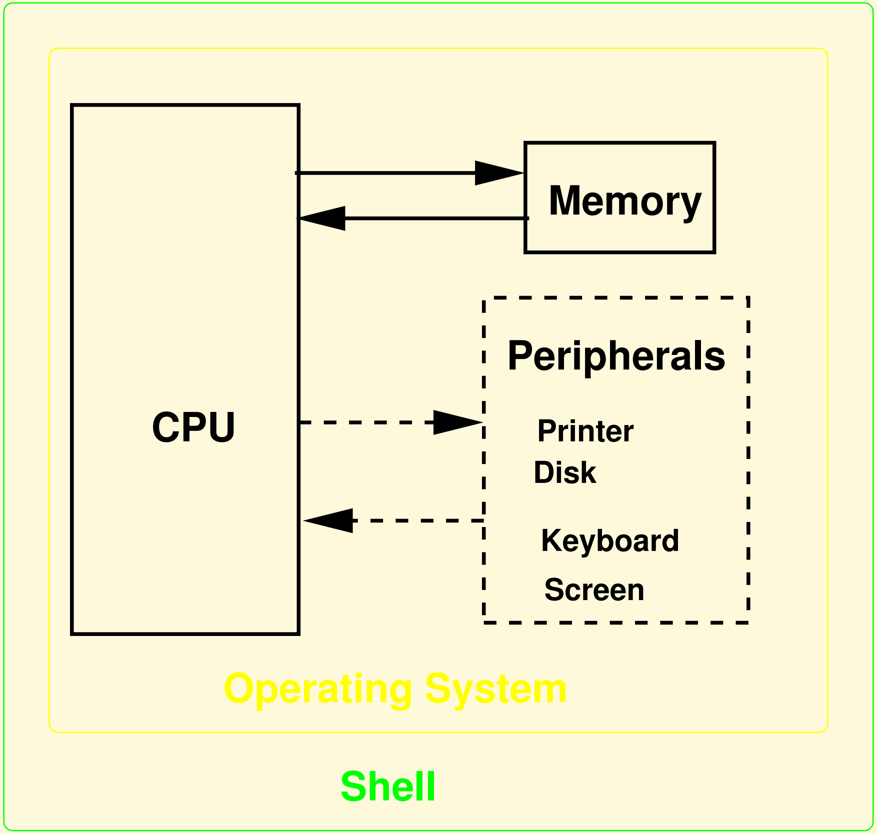
CPU & Memory: Simplified



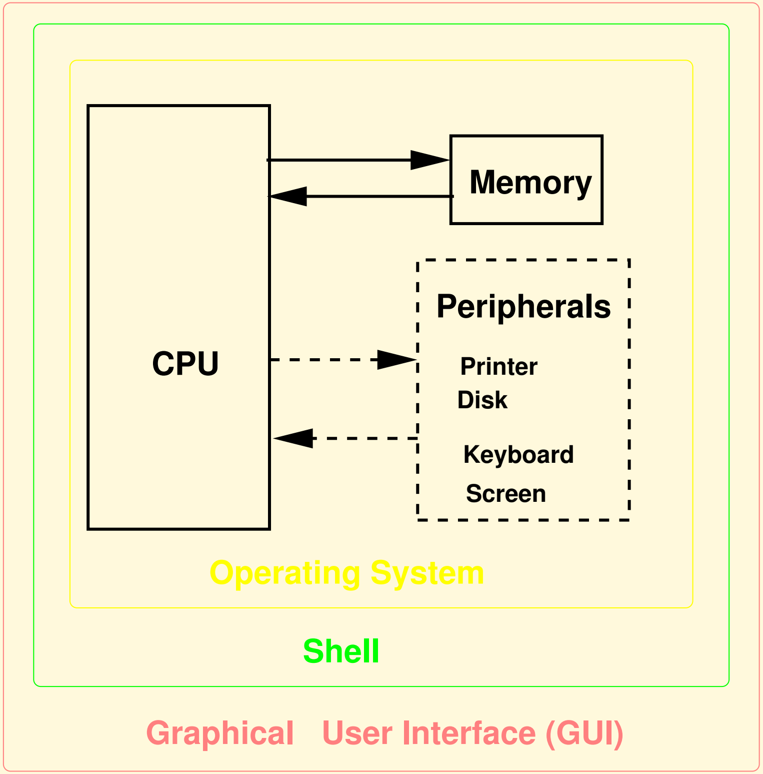
Resource Management



Shell: User Interface



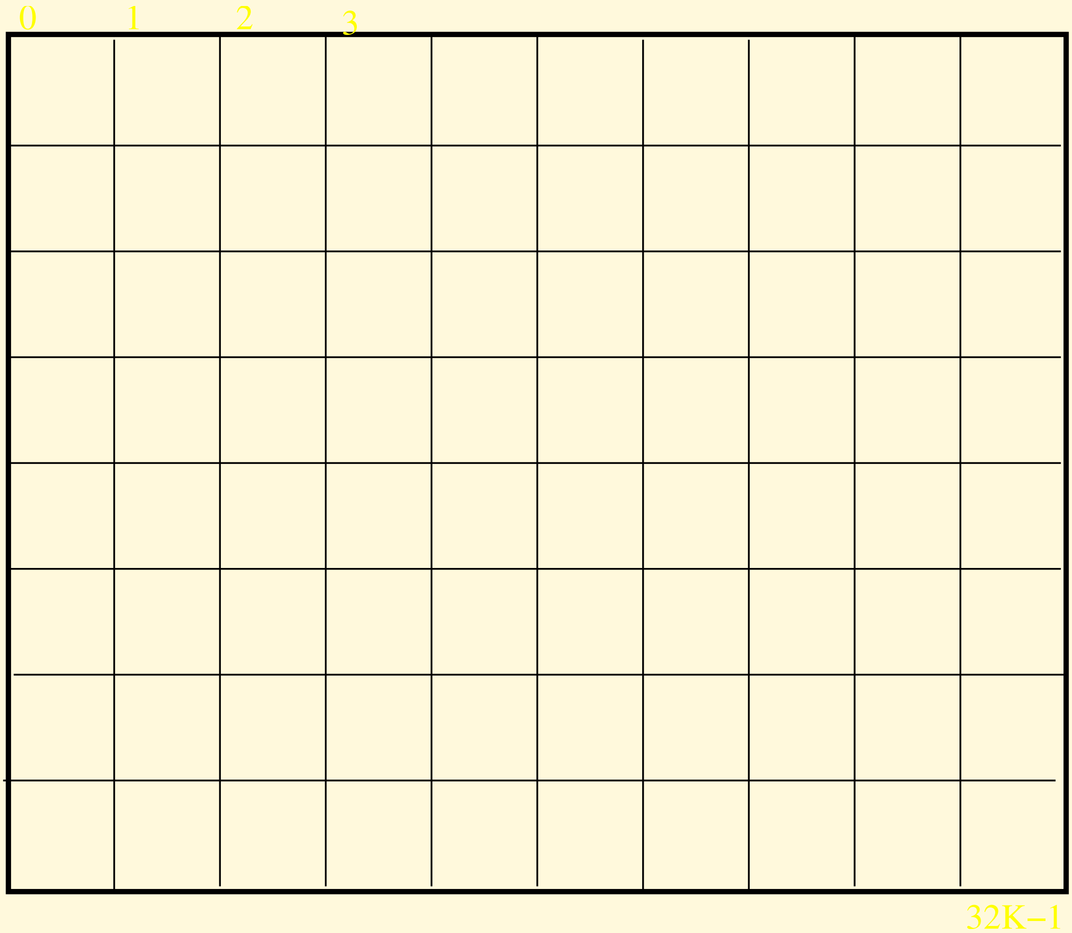
GUI: User Interface



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)

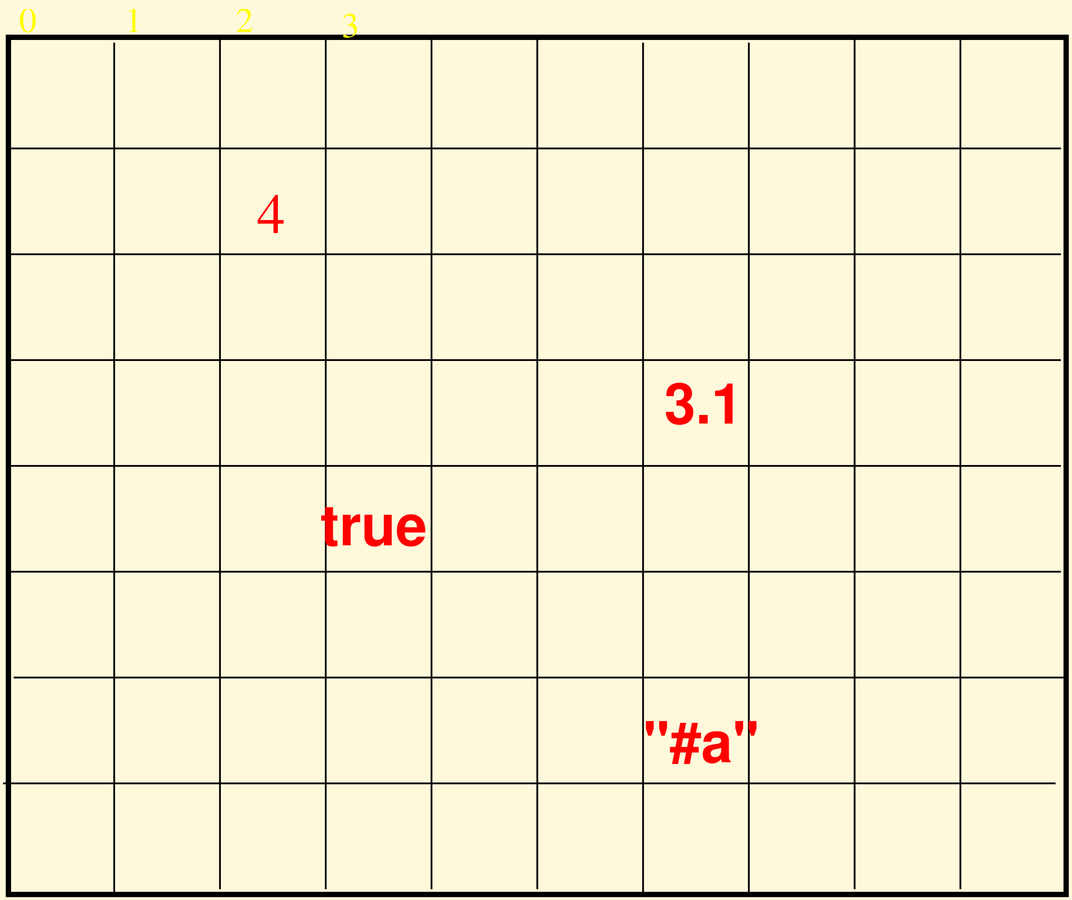
Memory



The Imperative Model

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

State Changes: σ



Assume all other cells are filled with null

State

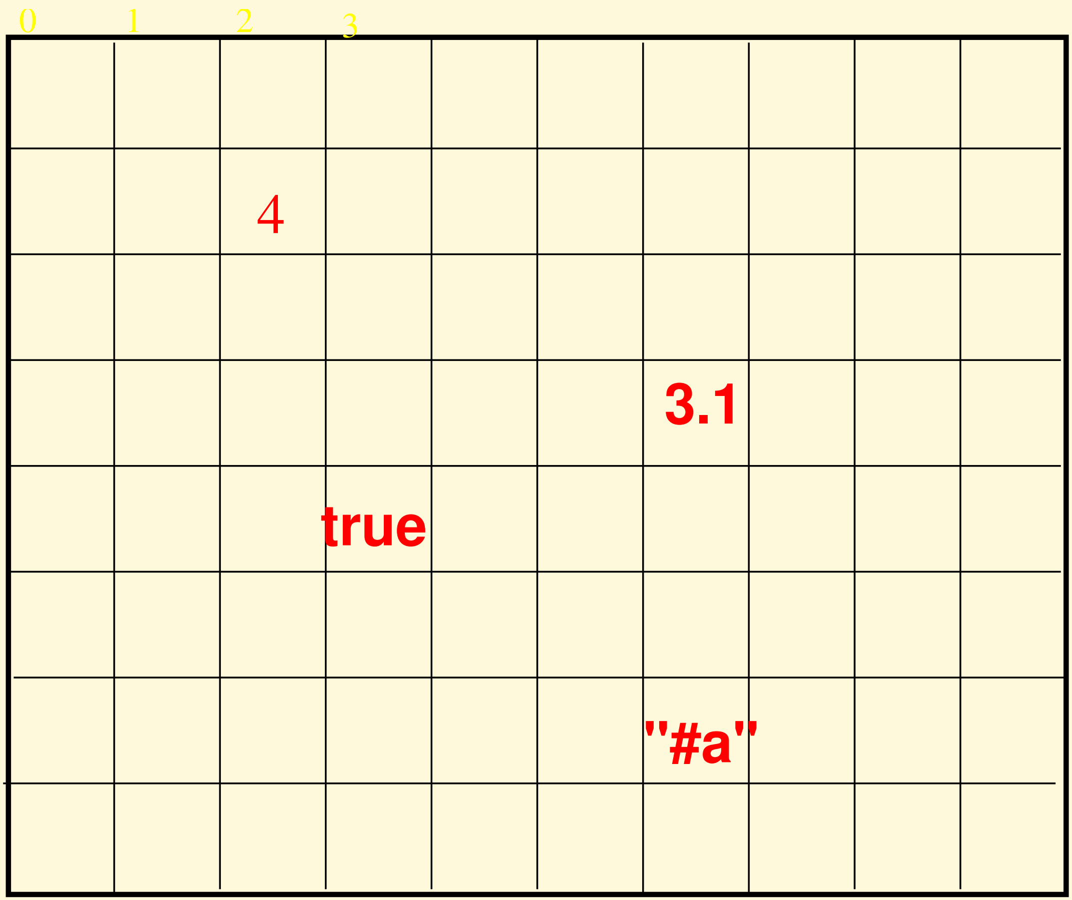
The state σ

- $\sigma(12) = 4 : int$
- $\sigma(20) = \mathsf{null}$
- $\sigma(43) = \mathsf{true} : bool$
- $\sigma(66) = "\#a" : char$

State Changes

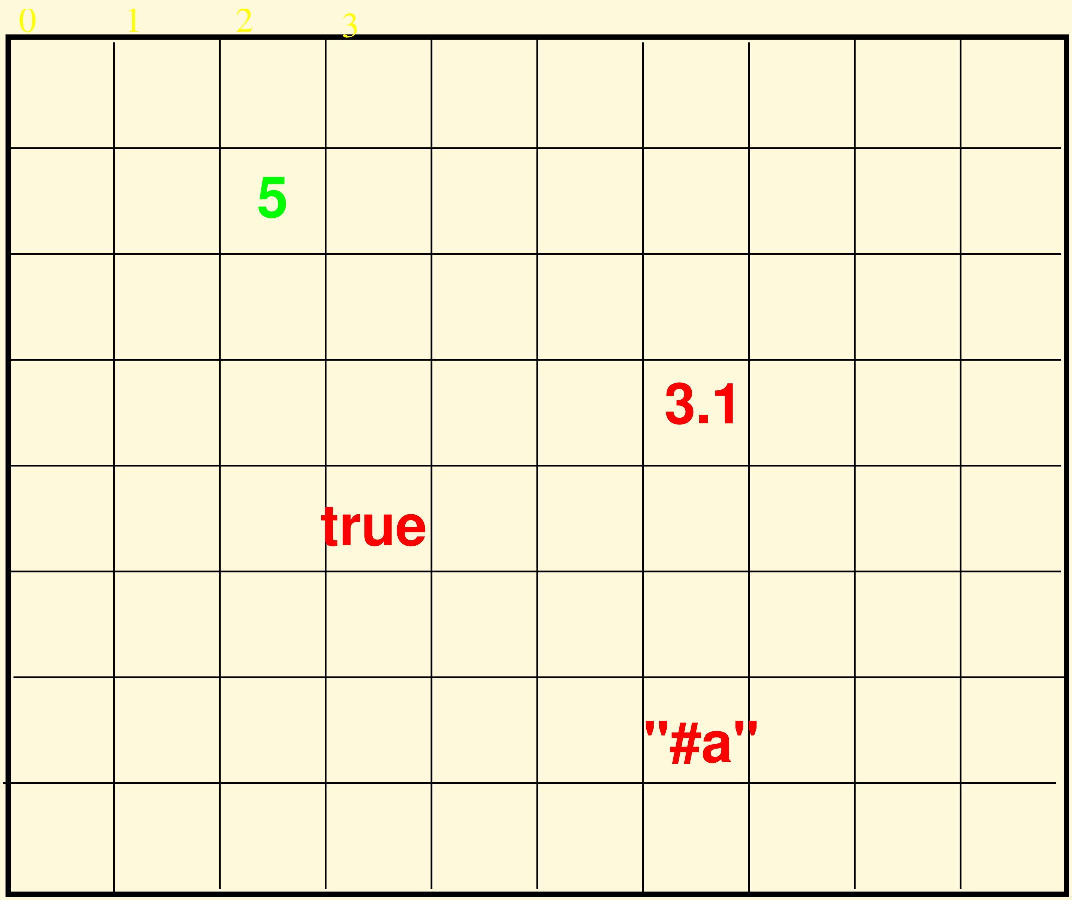
- A state change takes place when the value in some cell changes
- The contents of only one cell may be changed at a time.

State Changes: σ



Assume all other cells are filled with null

State Changes: σ_1



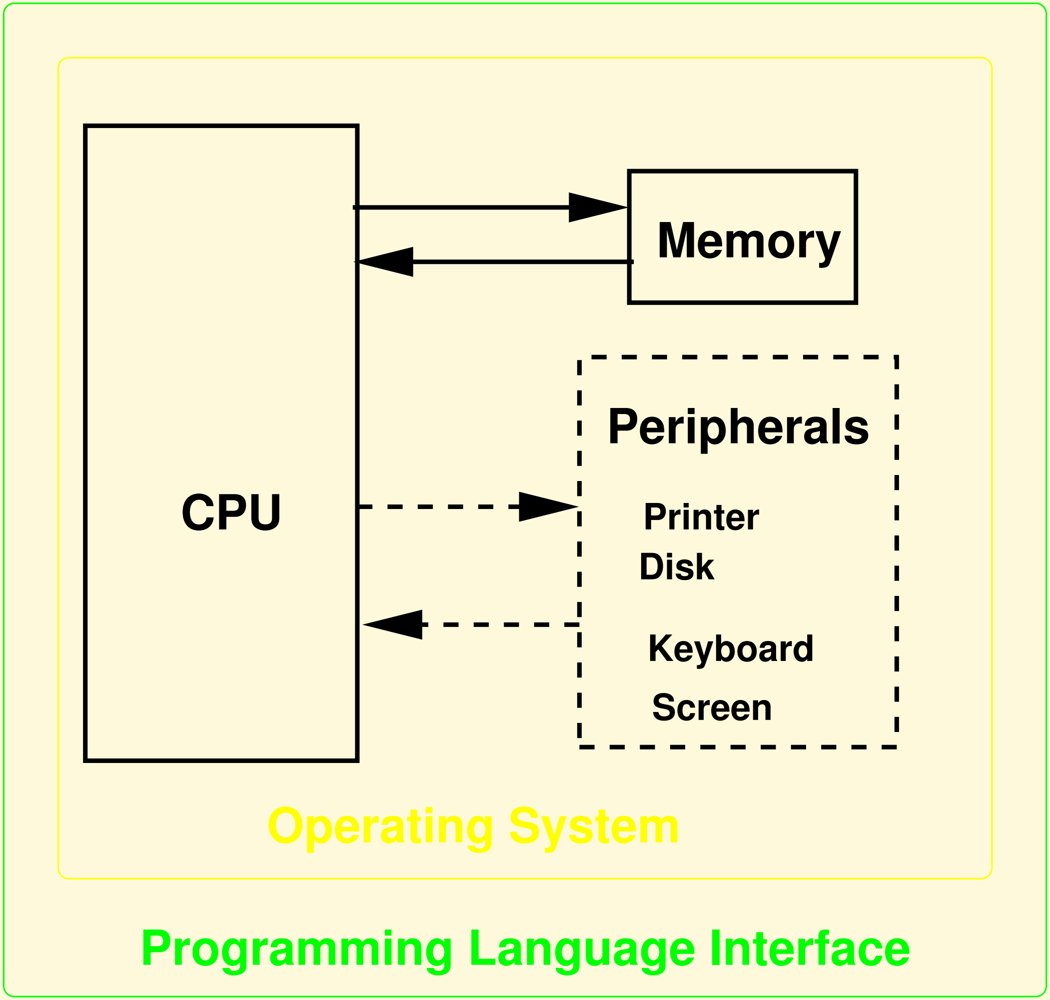
Assume all other cells are filled with null

State Changes: σ_2

0	1	2	3						
		5							
						3.1			
			true						
12						"#a"			

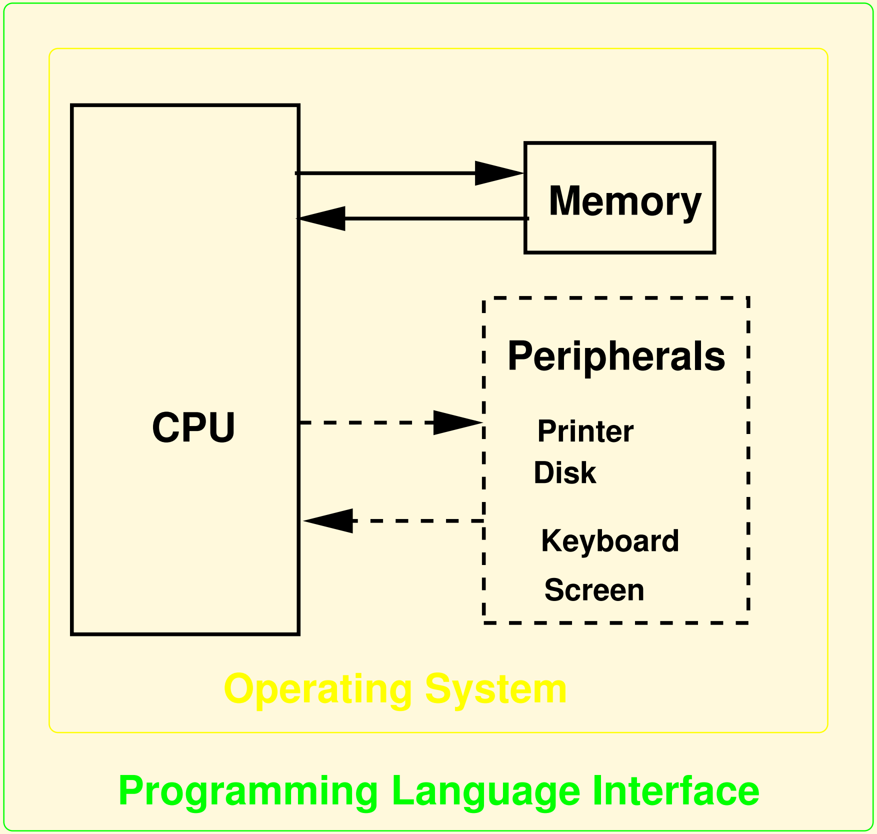
Assume all other cells are filled with `null`

Languages



Memory
CPU
Peripherals
Printer
Disk
Keyboard
Screen
Operating System
Programming Language Interface

User Programs



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

Functional	Imperative
name of a value	name of an address
constant	could change with time

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 $!y = 2$.

x	**1**	**y**	**2**

Assignment Commands

Store the value 5 *in* x.

$$\boxed{x := 5}$$

x	**5**	**y**	**2**

Assignment Commands

Copy the value contained in y *into* x.

$$\boxed{x := !y}$$

x	**2**	**y**	**2**

Assignment Commands

Increment the value contained in x *by* 1.

$$\boxed{x := !x + 1}$$

.

x	**3**	**y**	**2**

Assignment Commands

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

$$\boxed{y :=!x*!y}$$

x | **3**

y | **6**

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?

$$\boxed{val\ t\ =\ \mathtt{ref}\ 0}$$

Then the rest is easy

```
val t = ref 0;
t := !x;
x := !y;
y := !t;
```

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

```
val t = !x;
x := !y;
y := t;
```

7.2. Imperative Programming:

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 $ref : \alpha \rightarrow \alpha\ ref$

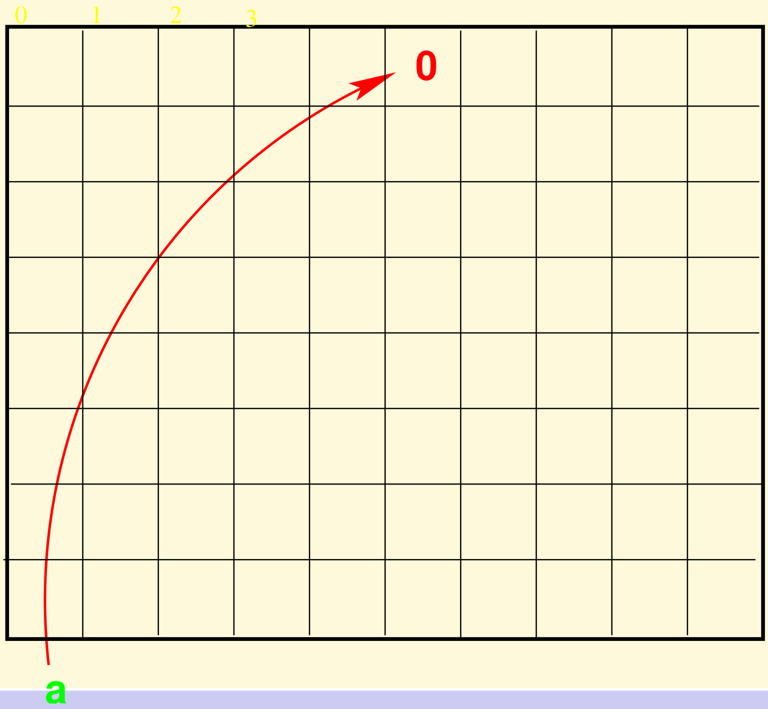
Dereferencing $! : \alpha\ ref \rightarrow \alpha$

Updation $:=: \alpha\ ref * \alpha \rightarrow unit$

Deallocation of memory is automatic!

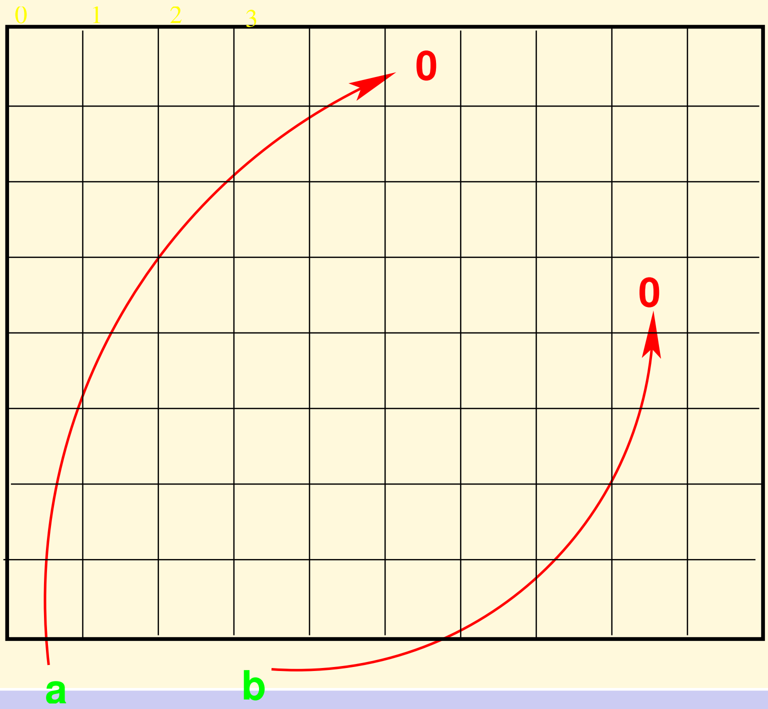
References: Experiments

```
- val a = ref 0;
val a = ref 0 : int ref
```



References: Experiments

```
- val b = ref 0;
val b = ref 0 : int ref
```

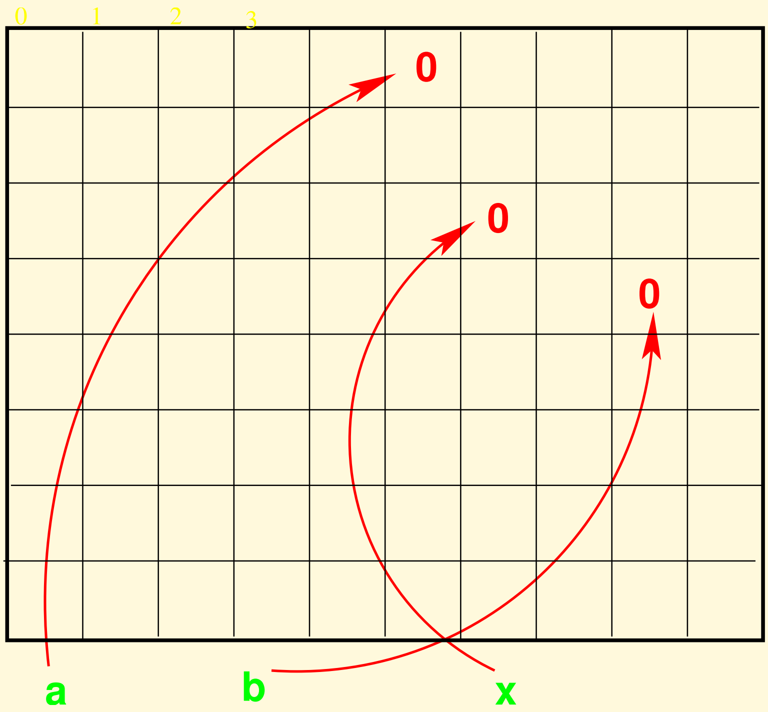


References: Experiments

```
- a = b;
val it = false : bool
- !a = !b;
val it = true : bool
```

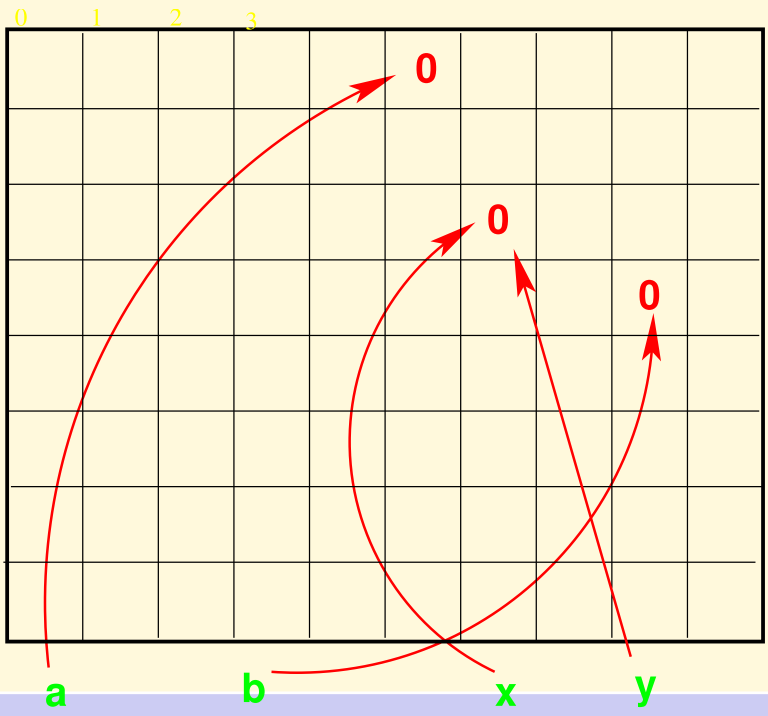
Aliases

```
- val x = ref 0;
val x = ref 0 : int ref
```



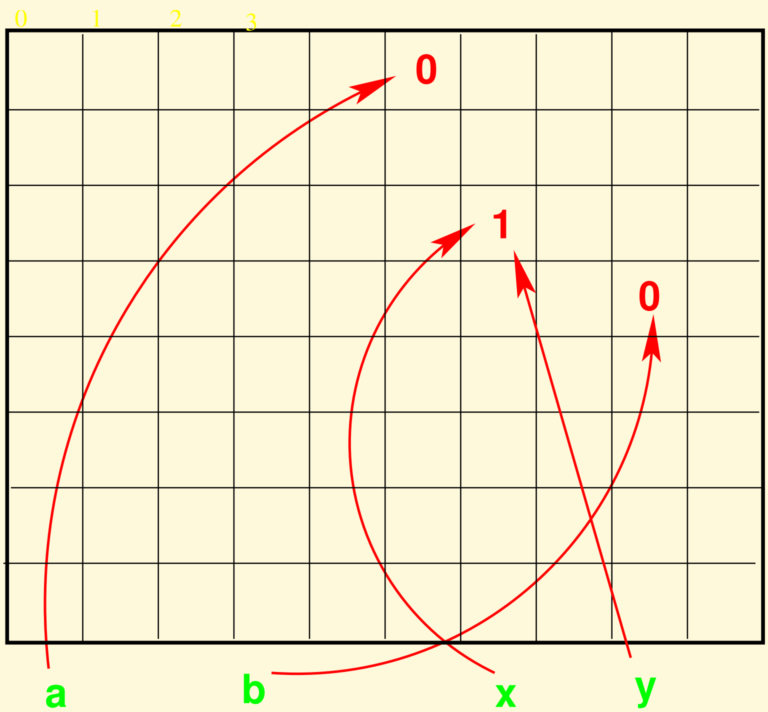
References: Experiments

```
- val y = x;
val y = ref 0 : int ref
```



References: Aliases

```
- x := !x + 1;
val it = () : unit
```

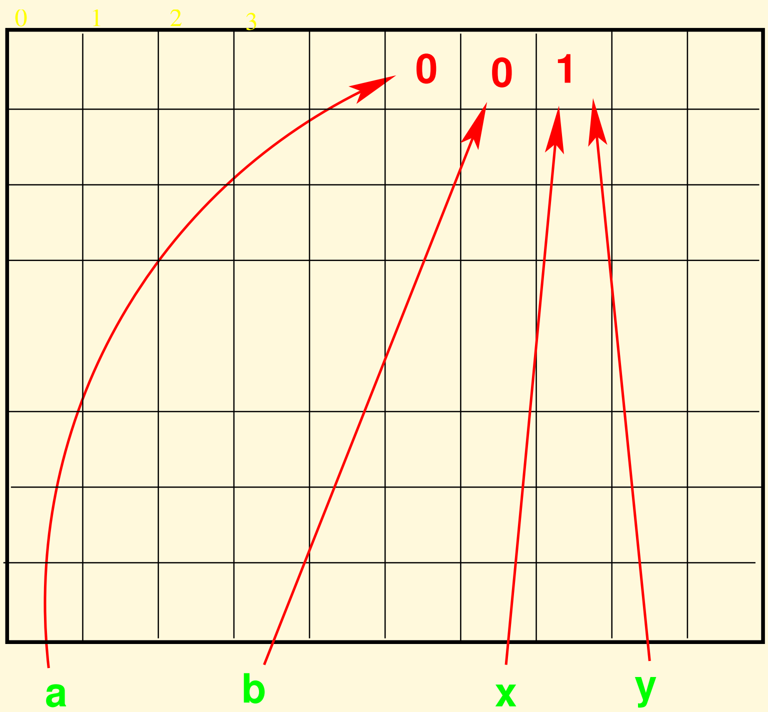


References: Experiments

```
- !y;
val it = 1 : int
- x = y;
val it = true : bool
```

After Garbage Collection

GC #0.0.0.0.2.45:   (0 ms)



Side Effects

- Assignment does not produce a value
- It produces only a state change (side effect)
- But side-effects are compatible with functional programming since it is provided as a new data type with constructors and destructors.

Imperative ML

- Does not provide direct access to memory addresses
- Does not allow for uninitialized imperative variables
- Provides a type with every memory location
- Manages the memory completely automatically

Imperative ML

- Does not provide direct access to memory addresses
  - Prevents the use of memory addresses as integers that can be manipulated by the user program
- Does not allow for uninitialized imperative variables
- Provides a type with every memory location
- Manages the memory completely automatically

Imperative ML

- Does not provide direct access to memory addresses
- Does not allow for uninitialized imperative variables
  - Most imperative languages keep declarations separate from initializations
- Provides a type with every memory cell
- Manages the memory completely automatically

Imperative ML

- Does not provide direct access to memory addresses
- Does not allow for uninitialized imperative variables
  - A frequent source of surprising results in most imperative language programs
- Provides a type with every memory cell
- Manages the memory completely automatically

Nasty Surprises

Separation of declaration from initialization

- Uninitialized variables
- Execution time errors if not detected by compiler, since every memory location contains some data
- Might use a value stored previously in that location by some imperative variable that no longer exists.
- Errors due to type violations.

Imperative ML

- Does not provide direct access to memory addresses
- Does not allow for uninitialized imperative variables
- Provides a type with every memory cell
- Manages the memory completely automatically and securely.

Imperative ML

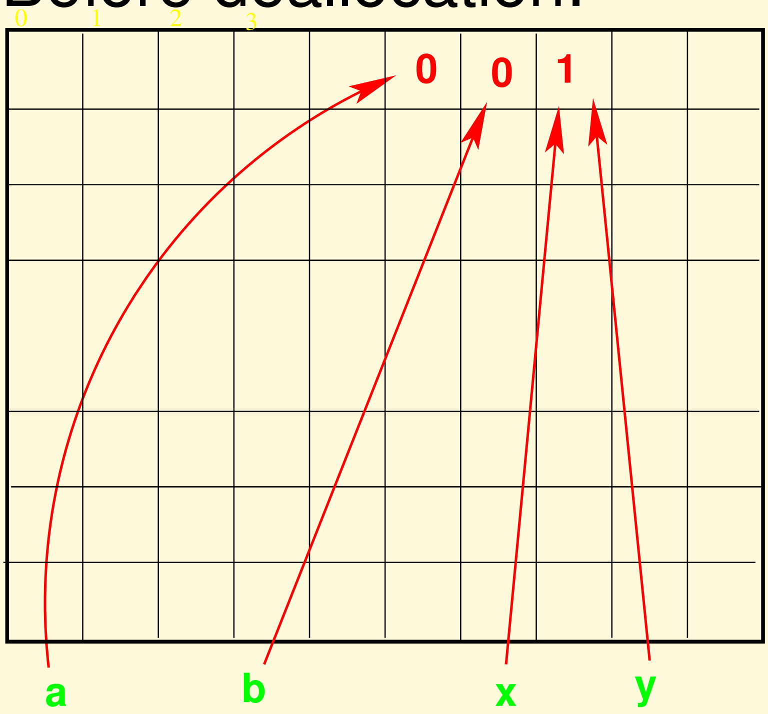
- Does not provide direct access to memory addresses
- Does not allow for uninitialized imperative variables
- Provides a type with every memory cell
- Manages the memory completely automatically and securely
  - Memory has to be managed by the user program in most languages
  - Prone to various errors

Common Errors

- Memory access errors due to integer arithmetic, especially in large structures (arrays)
- Dangling references on deallocation of aliased memory

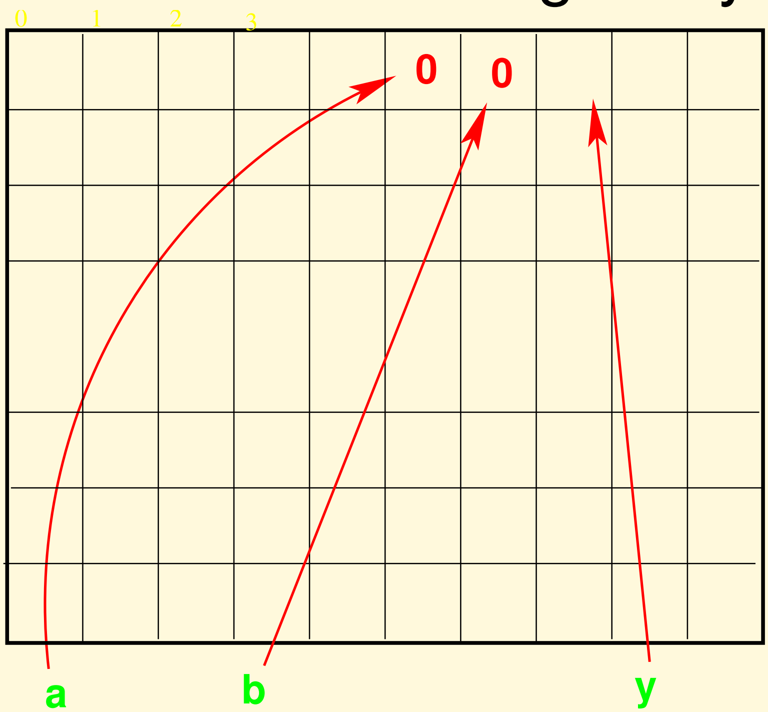
Aliasing & References

Before deallocation:



Dangling References

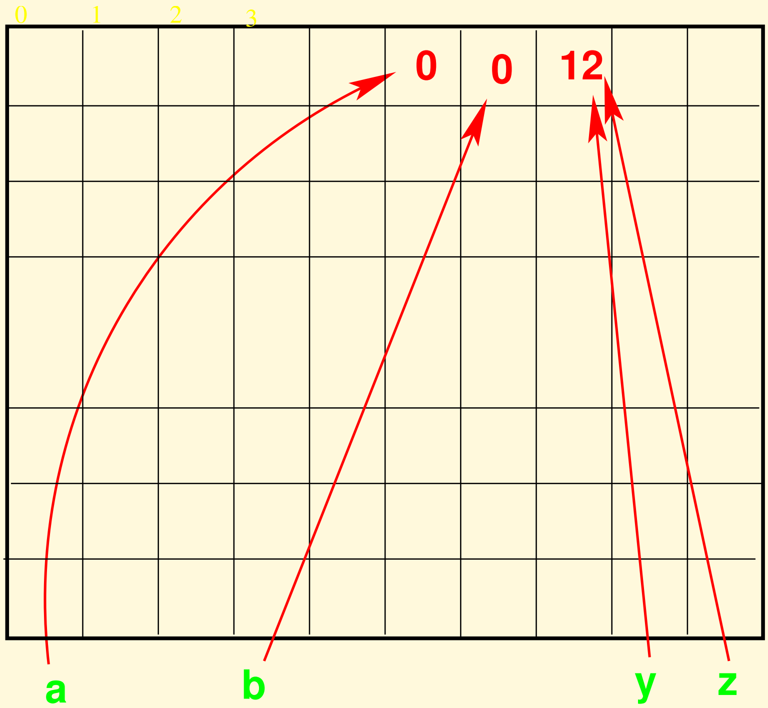
Deallocate x through a system call



y is left dangling!

New Reference

```
val z = ref 12;
```



By sheer coincidence $!y = 12$

Imperative Commands: Basic

A Command is an ML expression that creates a side effect and returns an empty tuple ($() : unit$).

Assignment

print

Imperative Commands: Compound

Any complex ML expression or function definition whose type is of the form $\alpha \rightarrow unit$ is a compound command.

- Predefined ML compound commands
- Could be user-defined. After all, *everything* is a *value*!

Predefined Compound Commands

branching $if\ e\ then\ c_1\ else\ c_0$.

cases $case\ e\ of p_1 \Rightarrow c_1 | \cdots | p_n \Rightarrow c_n$

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

looping $\underline{while\ e\ do\ c_1}$ is defined recursively as

$$if\ e\ then\ (c_1; \underline{while\ e\ do\ c_1})\ else\ ()$$

7.3. Arrays

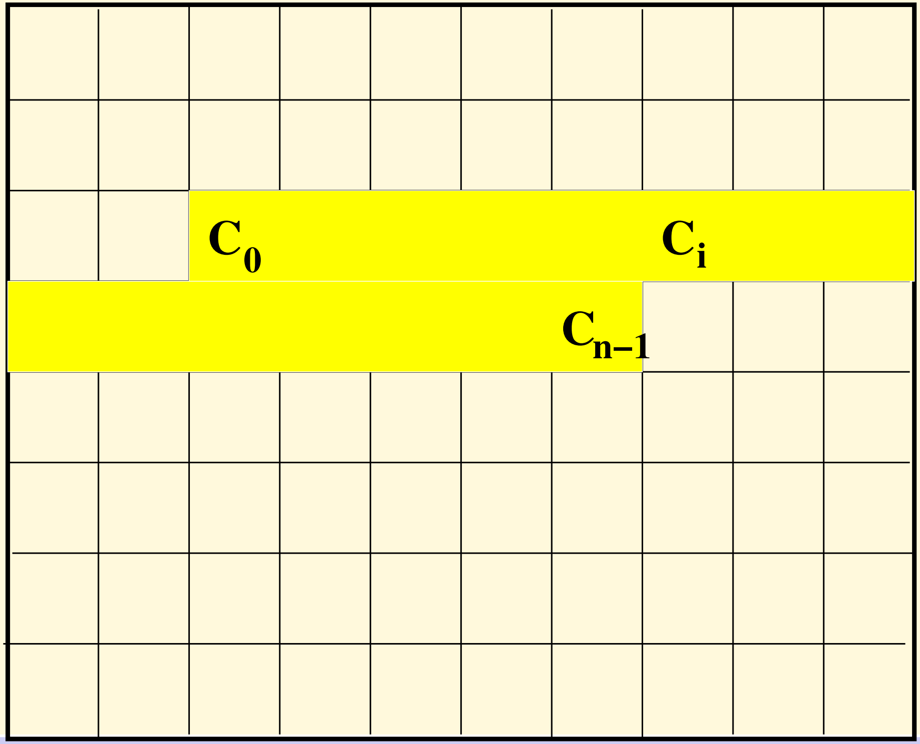
Why Imperative

- 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

$$C_0, C_1, \ldots, C_{n-1}$$

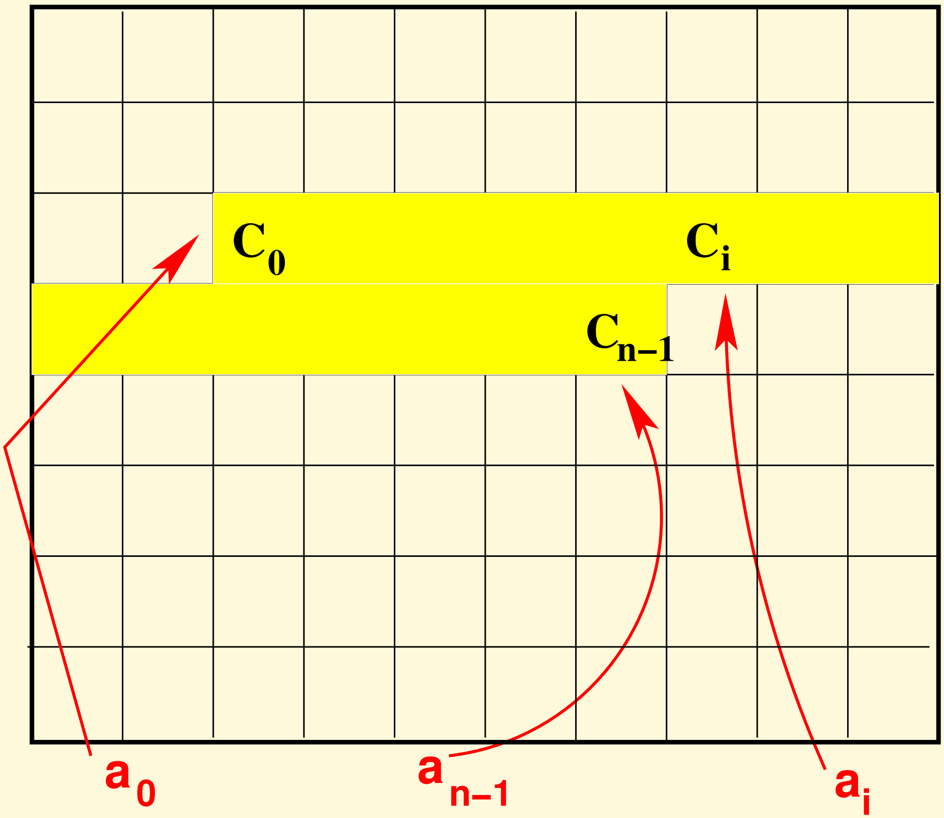


Indexing Arrays

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



Indexing Arrays

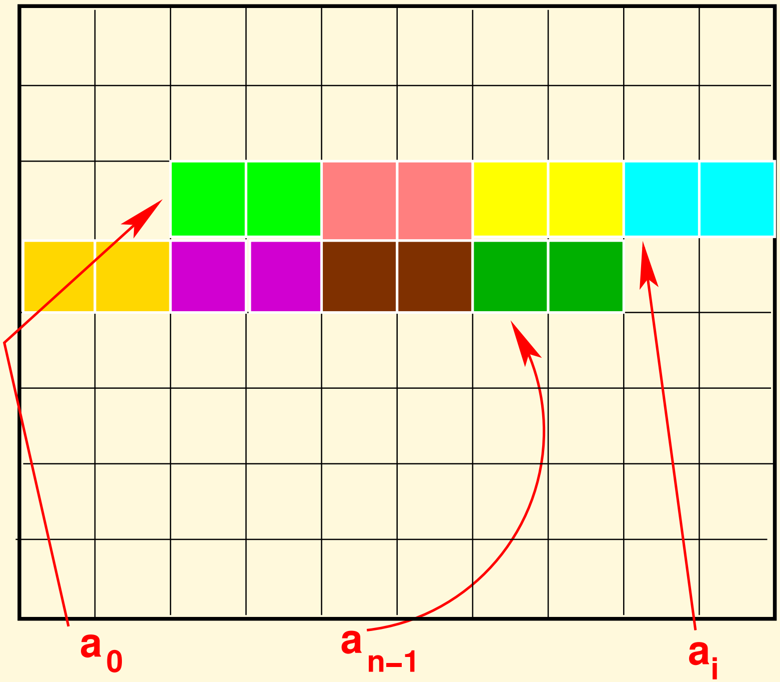
- The *start* address of the array and the *address* of C_0 are the same (say a_0)
- The address a_i of cell C_i is

$$\boxed{a_i = a_0 + i}$$

Physical Addressing

If each element occupies s *physical* memory locations, then

$$\boxed{a_i = a_0 + i \times s}$$



Arrays

A 2-dimensional array of

- r rows numbered 0 to $r-1$
- each row containing c elements numbered 0 to $c-1$

is also a contiguous sequence of rc memory cells

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

2D Arrays

A 2 dimensional-array is represented as an array of length $r \times c$, where

- a_{00} is the start address of the array, and
- the address of the (i, j)-th cell is given by

$$\boxed{a_{ij} = a_{00} + (c \times i + j)}$$

- the physical address of the (i, j)-th cell is given by

$$\boxed{a_{ij} = a_{00} + (c \times i + j) \times s}$$

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.

$$succ(i, j) = \begin{cases} (i+1, 0) & \text{if } j = n-1 \\ (i, j+1) & \text{else} \end{cases}$$

Ordering of indices

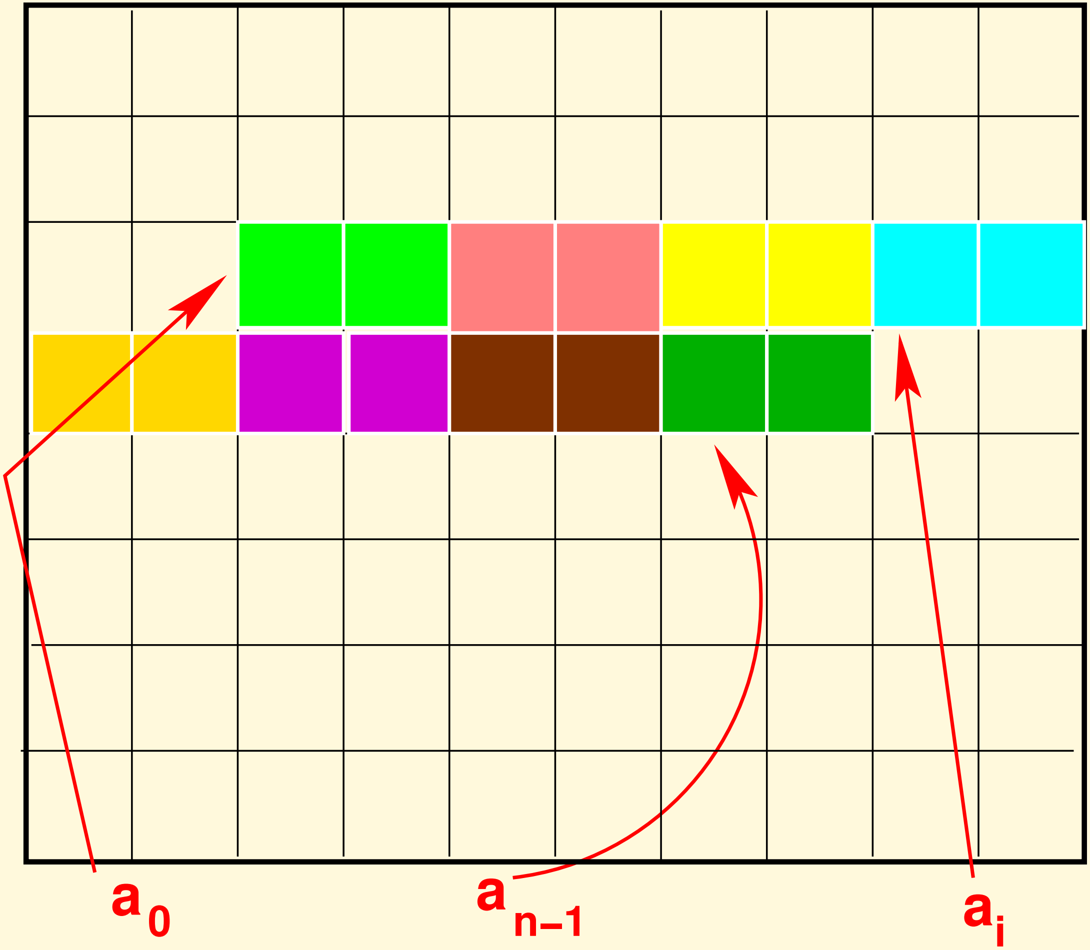
There is a natural "$<$" ordering on indices given by

$$
\begin{array}{lll}
(i,j) < (k,l) & \iff & \\
(i < k) & & \text{or} \\
(i = k & & \text{and } j < l)
\end{array}
$$

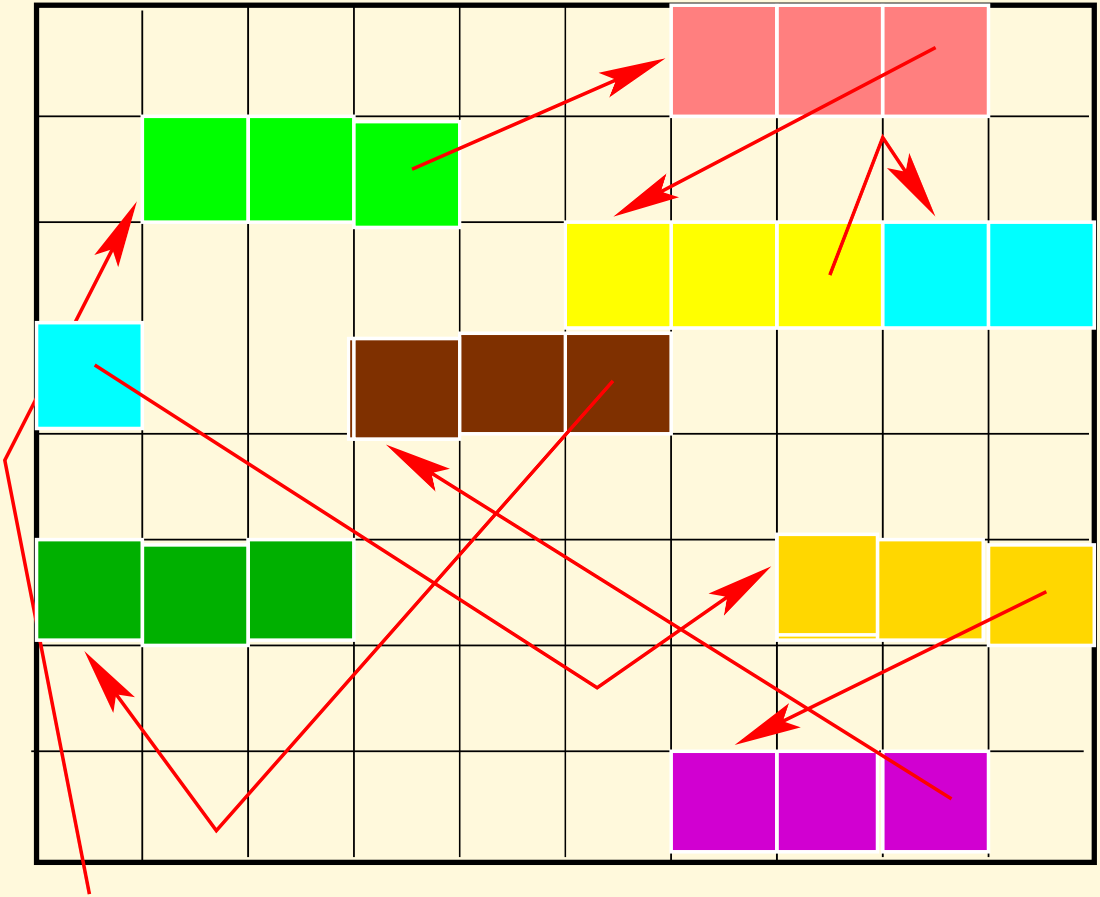
Arrays vs. Lists

Lists	Arrays
Unbounded lengths	Fixed length
Insertions possible	Very complex
Indirect access	Direct access

Arrays: Physical



Lists: Physical



8. A large Example: Tautology Checking

8.1. Large Example: Tautology Checking

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 can*not *know God empirically*, then (*we can*not *know God by experience* and *we can*not *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.

operator	symbol
not	$\neg$
and	$\wedge$
or	$\vee$
if...then...	$\Rightarrow$
equivalent	$\Longleftrightarrow$

Valuations

Given truth values to individual atoms the truth values of compound propositions are evaluated as follows:

p	$\neg p$
true	false
false	true

Valuations

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \iff q$
true	true	true	true	true	true
true	false	false	true	false	false
false	true	false	true	true	false
false	false	false	false	true	true

Tautology

A (compound) proposition is a tautology if it is true regardless of what truth values are assigned to its atoms.

Examples.

- $p \vee \neg p$
- $(p \wedge q) \Rightarrow p$
- $(p \wedge \neg p) \Rightarrow q$

Properties

- Every proposition may be expressed in a logically equivalent form using only the operators $\neg$, $\wedge$ and $\vee$

$$\boxed{(p \iff q) = (p \Rightarrow q) \wedge (q \Rightarrow p)}$$

$$\boxed{(p \Rightarrow q) = (\neg p \vee q)}$$

- De Morgan's laws may be applied to push $\neg$ inward

$$\boxed{\neg(p \wedge q) = \neg p \vee \neg q}$$

$$\boxed{\neg(p \vee q) = \neg p \wedge \neg q}$$

Negation Normal Form

- Double negations may be removed since

$$\boxed{\neg\neg p = p}$$

- Every proposition may be expressed in a form containing only $\wedge$ and $\vee$ with $\neg$ appearing only in front of atoms.

Literals & Clauses

- A literal is either an atom or $\neg$ applied to an atom
- $\vee$ is commutative and associative
- A clause is of the form $\bigvee_{j=1}^{m} l_j$, where each l_j is a literal.

Conjunctive Normal Form

- $\vee$ may be distributed over $\wedge$

$$\boxed{p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)}$$

- $\wedge$ 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*

$$(h_1 \wedge \ldots \wedge h_n) \Rightarrow c$$

is a tautology

Problem

Given an argument $[h_1, \ldots, h_n] | c$,

- determine whether $(h_1 \wedge \ldots \wedge h_n) \Rightarrow c$ 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 <u>if and only if</u> 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 <u>if and only if</u> 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.

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.
- Transfom 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 $\Longleftrightarrow$ and $\Rightarrow$
- `nnf` converts into NNF
- `conj_of_disj` converts into CNF

Rewrite into NNF

- Eliminate $\iff$ and then $\Rightarrow$
- Push $\neg$ inward using De Morgan's laws and eliminate double negations.

$\Longleftrightarrow$ and $\Rightarrow$ Elimination

```
fun rewrite (ATOM a) = ATOM a
|   rewrite (IMP (P, Q)) =
    OR (NOT (rewrite(P)),
        rewrite(Q))
|   rewrite (EQL (P, Q)) =
    rewrite (AND (IMP(P, Q),
                  IMP (Q, P)))
|   ...
```

Proposition made up of only $\neg$, $\wedge$ and $\vee$.

Push ¬ inward

```
fun nnf (ATOM a) =
     ATOM a
|    nnf (NOT (ATOM a)) =
     NOT (ATOM a)
|    nnf (NOT (NOT (P))) =
     nnf (P)
```

Push $\neg$ inward

```
|   nnf (NOT (AND (P, Q))) =
    nnf (OR (NOT (P),
             NOT (Q)))
|   nnf (NOT (OR (P, Q))) =
    nnf (AND (NOT (P),
              NOT (Q)))
|   ...
```

Proposition made up of only $\wedge$ and $\vee$ applied to positive or negative literals.

conj_of_disj

```
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 $\vee$ inward

Use distributivity of $\vee$ over $\wedge$

```
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

Next: Our Computing Tool

Our Computing Tool

Previous: Introduction to Computing

Next: Primitives: Integer & Real

Primitives: Integer & Real (34-60)

Previous: Primitives: Integer & Real

Next: Technical Completeness & Algorithms

User Defined Structured Data Types (452-472)