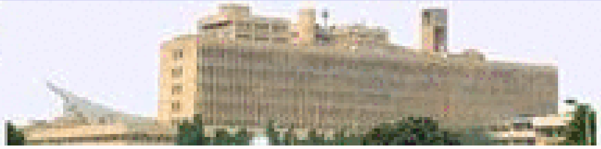
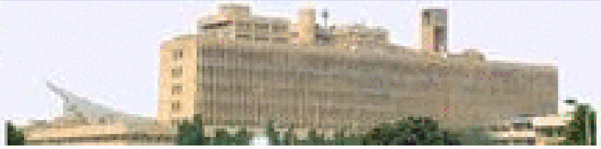


Introduction to Computer Science

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

October 12, 2013

Contents



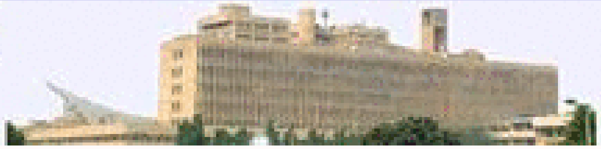
Colour Coding of text

- Black is normal foreground colour.
- Brown is the colour of slide headings.
- Various colours like brown, red and blue are used to provide emphasis or to stand out from the usual text.
- The special colour magenta is used to denote a *hyperlink*. These hyperlinks are "mouse-clickable" and they take you to a different slide often out of order. You may use the GO BACK button to return in the *reverse order* of the traversal of hyperlinks.
- `Blue type-writer` font is used for SML keywords, system responses while running programs etc.

1. Computing

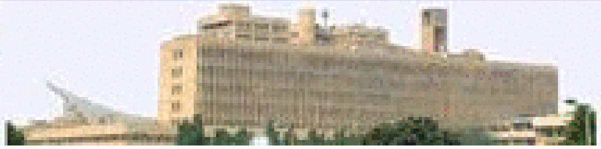
1.1. Introduction to Computing

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!)
- Straight-edge and compass

Straight-edge 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 $\sqrt{10}$

Straight-edge and Compass: Primitives

A more exact description of ruler and compass primitives

Straight-edge: Can specify lines, rays and line segments. Cannot really specify lengths since the ruler is only supposed to be an unmarked straight-edge.

Compass:

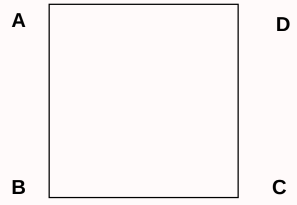
1. Can define arcs and circles,
2. Can specify only *comparative* non-zero lengths and not *absolute* lengths.

Straight-edge and Compass: Can specify points of intersection.

Note that an unmarked straight-edge cannot be used to compare lengths.

Problem: Doubling a Square

Consider an example from “Straight-edge and Compass Constructions”. Given a square, construct another square of twice the area of the original square.



Note.

- Whatever lengths we assume in the solution have to be taken only as symbolic and not absolute since with an unmarked straight-edge and compass only relative comparisons are possible – no absolute lengths are possible.
- The (symbolic) lengths we assume are necessary only in the justification of correctness of the construction.

Solution & Justification: Doubling a Square

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 $\square ACEF = 2a^2$.

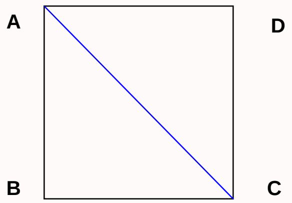
Note.

- Even though this is a simple 2-step algorithm, the steps may not all be primitive operations.
- The non-primitive operations need to be regarded as new problems which have to be solved. This is the process of *refinement* of the algorithm.
- In this case Step 2 is a non-primitive operation which needs to be solved separately. One possible solution is shown in the hyperlink target of step 2.
- Assuming that step 2 can be executed correctly the blue text provides the justification which proves the *correctness* of this algorithm for the given problem.

Execution: Step 1

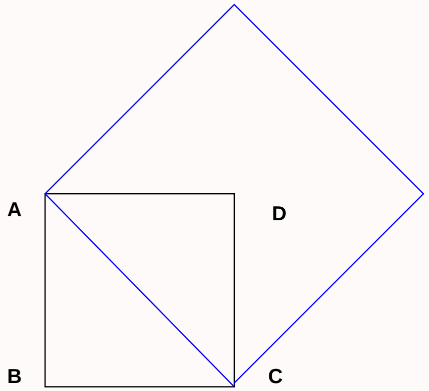
Step 1 is a primitive operation which may be executed using the straight-edge.

Draw the diagonal $\overline{AC}$.



Execution: Step 2

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



Step 2 is however not a primitive operation. So one regards it as a new problem to be solved by *refinement*.

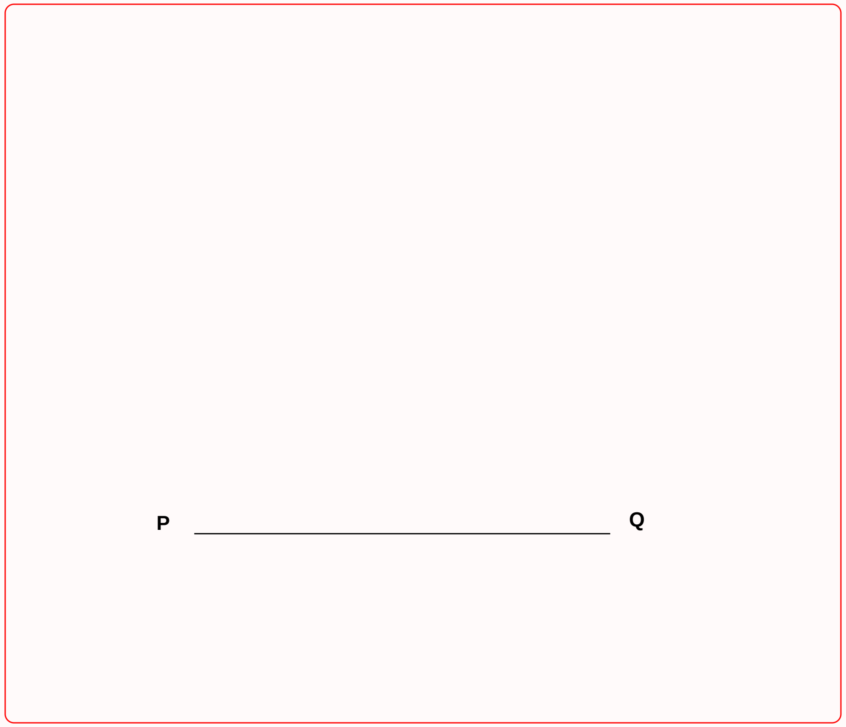
Refinement: Square

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

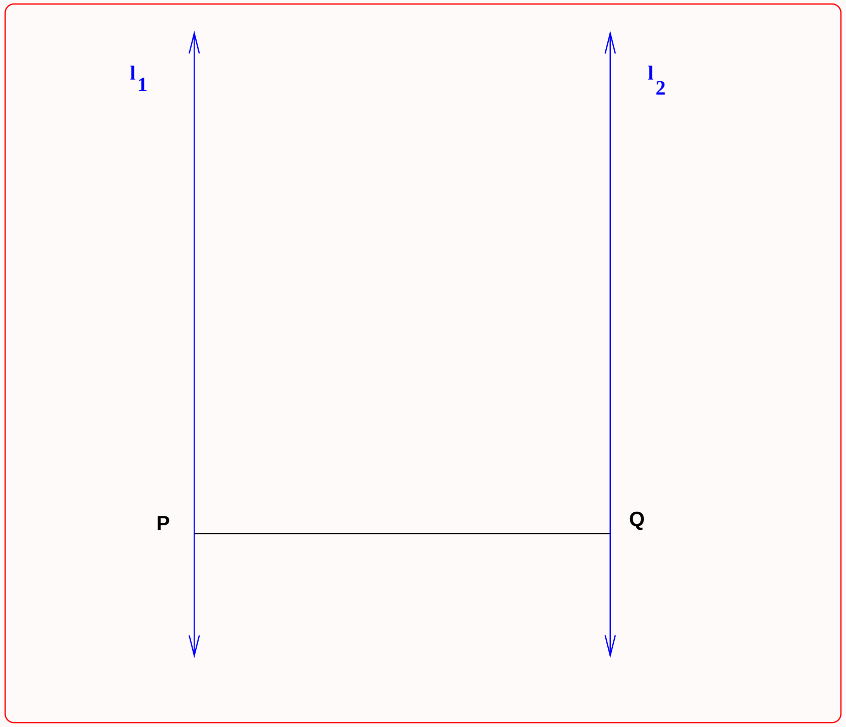
Square on Segment: 0

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



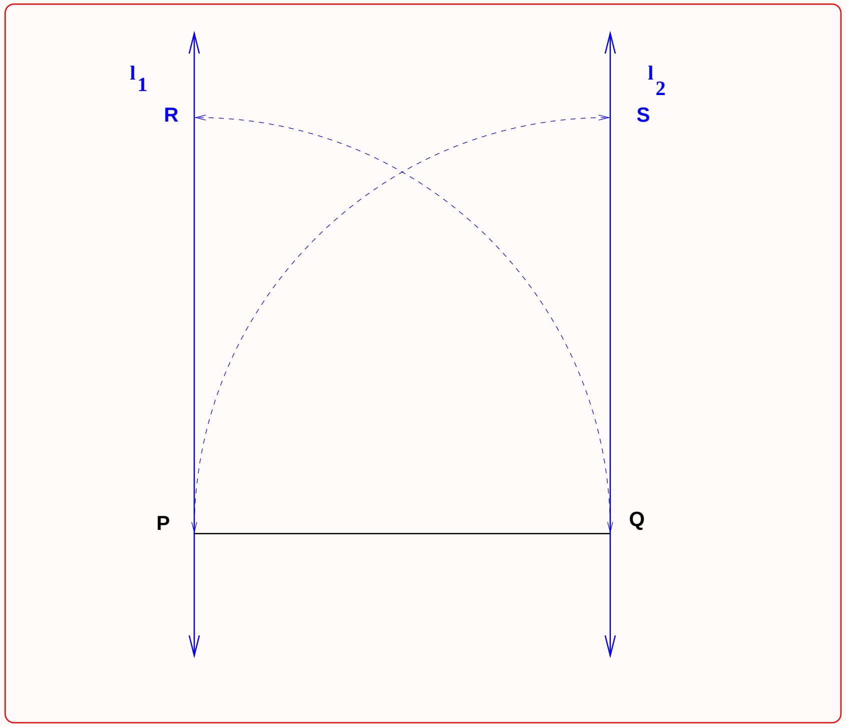
Square on Segment: 1

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



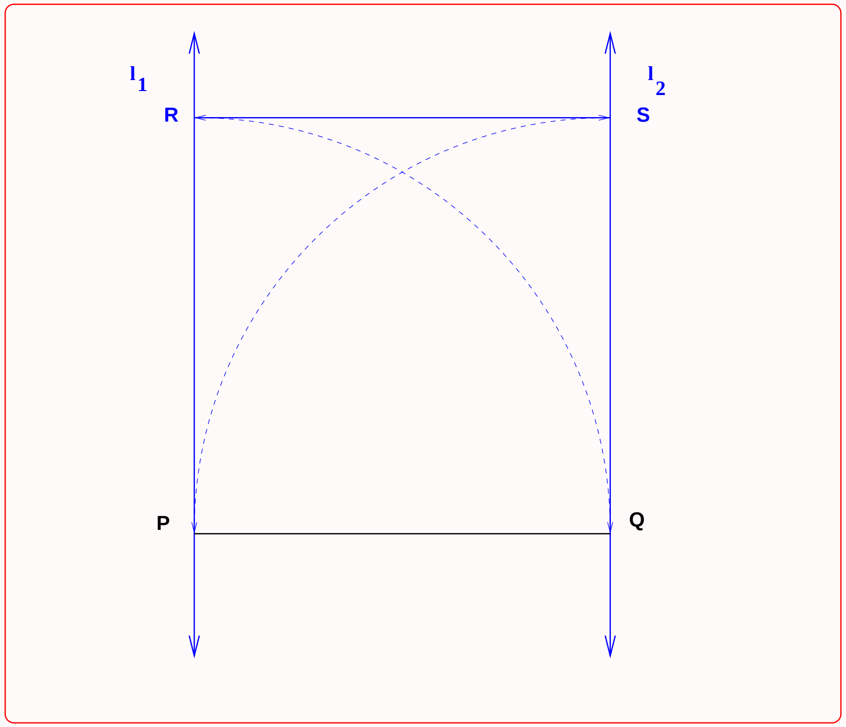
Square on Segment: 2

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

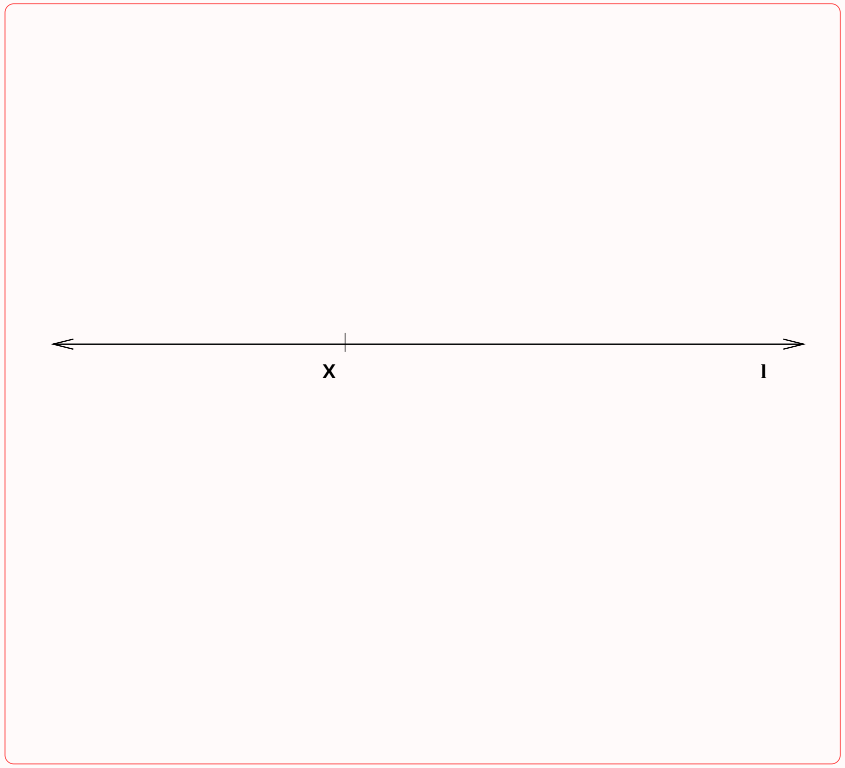


Square Construction algorithm

Refinement 2: Perpendicular at a point

Given a line, draw a perpendicular to it passing through a given point on it.

Assume given a line l containing a point X.

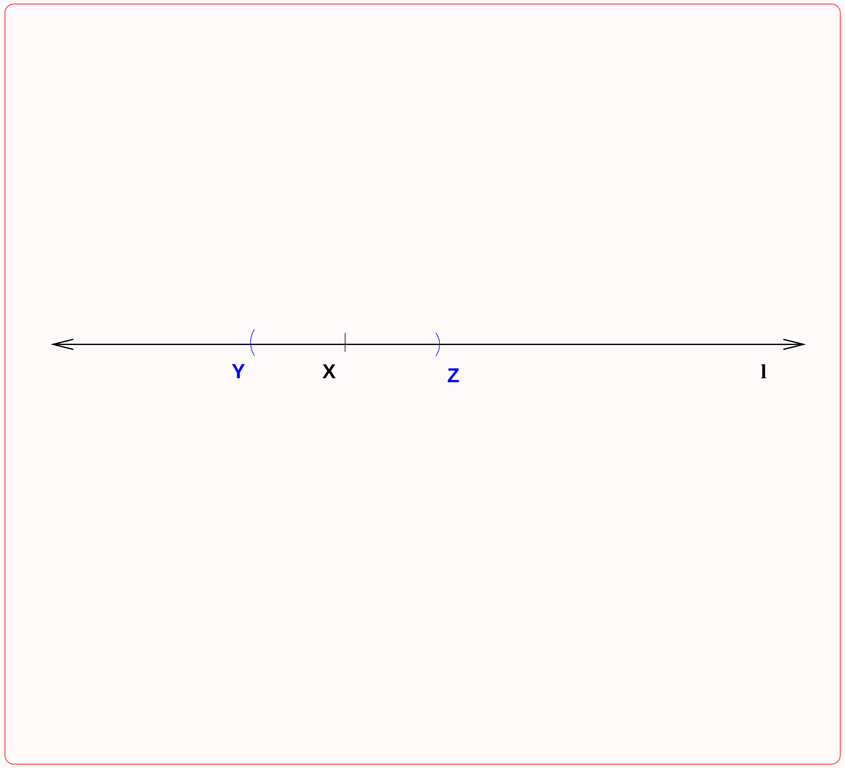


Solution: Perpendicular at a point

1. Choose a length $c > 0$. With X as centre mark off points Y and Z on l on either side of X, such that $YX = c = XZ$. $YZ = 2c$.
2. Draw Circles $C_1(Y, 2c)$ and $C_2(Z, 2c)$ respectively.
3. Join the points of intersection of the two circles.

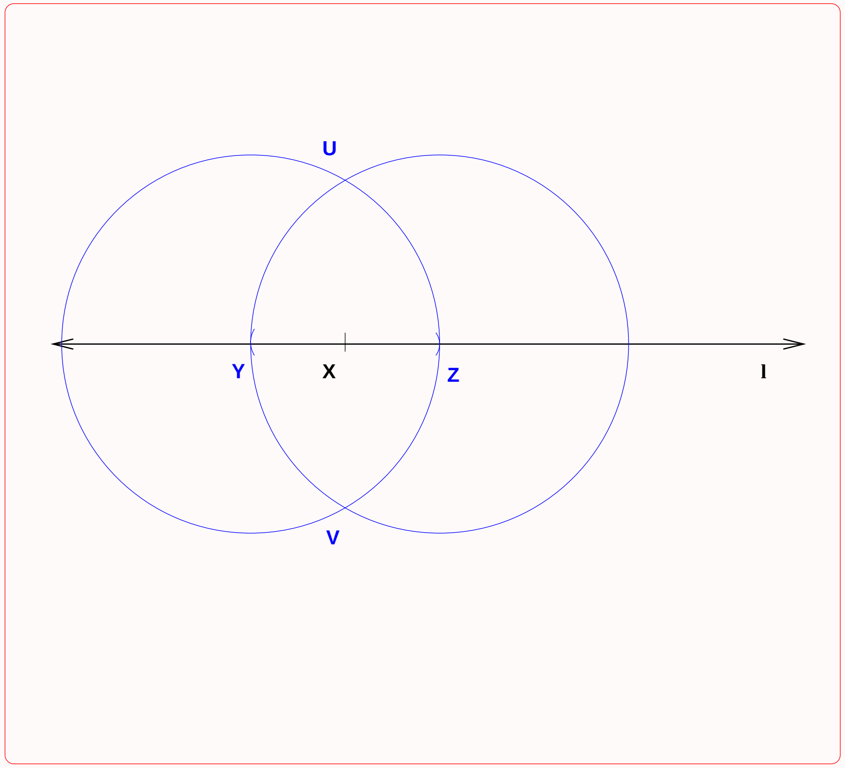
Perpendicular at a Point: 1

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



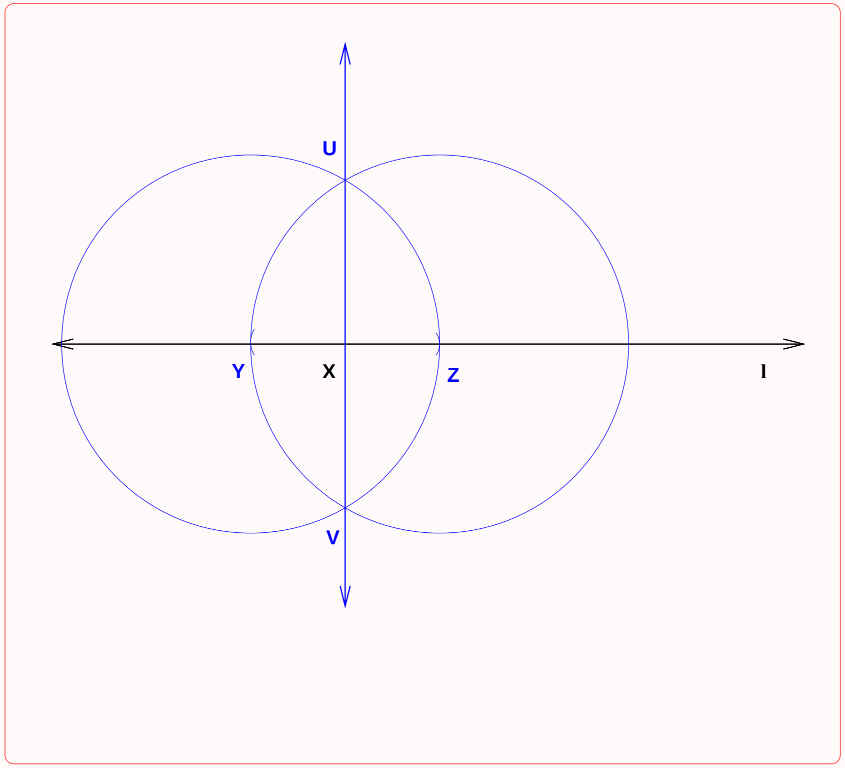
Perpendicular at a Point: 2

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



Perpendicular at a Point: 3

Join the points of intersection of the two circles.



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

The Computational Model

Every reasonable computational model such as the above must have the following capabilities:

Primitive operations and expressions which represent the simplest objects with which the model is concerned.

Methods of combination which specify how the primitive expressions and objects can be combined with one another to obtain compound expressions and compound objects.

Methods of abstraction which specify how compound expressions and objects can be named and manipulated as units.

Methods of Abstraction

The Methods of abstraction in the construction were

1. Perpendicular at a point on a line, which was used to construct the square of a given length.
2. Square Construction algorithm which was used to construct a square on a line segment.

The Use of Abstraction

Methods of abstraction are useful for

- specifying the method crisply and concisely in terms of problems known to be solvable (e.g the two-step solution to the given problem).
- separating logical sub-problems (e.g. the problem of perpendicular at a point on a line is a logically separate sub-problem of drawing a square on a line segment).
- avoiding repetitions of copies of similar solutions (e.g. to construct a square on a segment requires drawing *two* perpendiculars one at each end point of the line-segment – each perpendicular is an instance of the same algorithm viz. drawing a perpendicular to a line on a given point)

1.2. Long Multiplication

Consider the following example from school arithmetic which illustrates multiplication of large numbers by a method called *long multiplication*.

$$\begin{array}{rrrrrrrrr}
 & & & & 5 & 7 & 8 & 3 & 9 \\
 & & & & \times & 9 & 6 & 4 & 7 \\ \hline
 & & & 4 & 0 & 4 & 8 & 7 & 3 \\
 & & 2 & 3 & 1 & 3 & 5 & 6 & 0 \\
 & 3 & 4 & 7 & 0 & 3 & 4 & 0 & 0 \\
5 & 2 & 0 & 5 & 5 & 1 & 0 & 0 & 0 \\ \hline
5 & 5 & 7 & 9 & 7 & 2 & 8 & 3 & 3
\end{array}$$

Many schools and text-books however omit the zeroes (0) at the right end of each of the lines of intermediate products.

One could ask the following questions from a purely mathematical and algorithmic standpoint.

1. What is the *algorithm* behind this method? Can it be expressed in a general fashion without resorting to examples?

2. How does one guarantee that the method is *correct*? In other words, what is the mathematical *justification* of this method?
3. How does this compare with other methods of multiplication (for e.g. multiplication by repeated addition)?

1.2.1. The algorithm

A good way to design algorithms is also to design their justification side by side. Let a and b be two non-negative integers expressed in the usual decimal place-value notation. We use the notation $(a)_{10}$ to indicate that we are looking at the digits of a expressed as a sequence of digits $a_m, \ldots, a_0$ for some $m \geq 0$, where a_0 is the least significant digit (units digit) of a and a_m is the most significant one. Similarly b may be represented by the digits $b_n, \ldots, b_0$ for some $n \geq 0$. The place-value system[1] of representation of numbers gives us the identities $a = \sum_{i=0}^{m} 10^i a_i$ and $b = \sum_{j=0}^{n} 10^j b_j$.

Assuming without loss of generality that a is the multiplicand and b is the multiplier, we get (using the fact that multiplication distributes over addition and that multiplication is commutative and associative)

$$\boxed{a \times b = a \times \sum_{j=0}^{n} 10^j b_j = \sum_{j=0}^{n} a \times 10^j b_j = \sum_{j=0}^{n} 10^j ab_j = ab_0 + ab_1.10 + \cdots + ab_n.10^n \qquad (1)}$$

[1] unlike some other system such as the Roman numerals

Equation (1) is at the heart of the long multiplication method and justifies the method. Notice that when $n > 0$, $ab_0 + ab_1.10 + \cdots + ab_n.10^n = ab_0 + (ab_1 + \cdots + ab_n.10^{n-1}).10$. For completeness, we should mention that when $n = 0$ the method reduces to multiplying by a single digit number.

The method itself may be expressed in a technically complete fashion by the following algorithm. We use "$\dot{-}$" to denote the quotient of integer division and "$\bar{\cdot}$" to denote the remainder of integer division.

$$longmult(a,b) = \begin{cases} ab_0 & \text{if } n = 0 \\ ab_0 + longmult(a,b').10 & \text{if } n > 0 \end{cases} \tag{2}$$

where

$$\begin{aligned} b_0 &= b \;\bar{\cdot}\; 10 \quad \text{and} \\ b' &= b \;\dot{-}\; 10 \end{aligned}$$

b_0 is the units digit of b and equals the remainder obtained by dividing b by 10. b' on the other hand, is the number obtained by "deleting the units digit" of b. More accurately, it is the quotient of $b/10$ and hence $b' = \sum_{j=1}^{n} 10^j b_j$.

Of course, while performing long-multiplications, it is usually not necessary to count or know in advance the number of digits in b. We further notice that the two conditions $n = 0$ and $n > 0$ may equally well be replaced by the conditions $b < 10$ and $b \geq 10$ respectively yielding another version of the same algorithm wherein the dependence on n is removed.

$$longmult(a,b) = \begin{cases} ab_0 & \text{if } b < 10 \\ ab_0 + longmult(a,b').10 & \text{if } b \geq 10 \end{cases} \quad (3)$$

where

$$\begin{aligned} b_0 &= b \;\bar{\cdot}\; 10 \quad \text{and} \\ b' &= b \;\dot{-}\; 10 \end{aligned}$$

1.2.2. Correctness

We prove the following fairly straightforward theorem, to illustrate how proofs of correctness by induction may go.

Theorem 1.1 **(The Correctness theorem).** *For all $a > 0$ and $b \geq 0$, algorithm (2) computes the product of a and b, i.e. $longmult(a,b) = a \times b = ab$.*

Proof: If $b = 0$ then clearly $b_0 = b = 0$ and $ab_0 = 0$ and there is nothing to prove. If $b > 0$ we proceed by induction on $n \geq 0$ such that $10^n \leq b < 10^{n+1}$ (i.e. $n + 1$ is the number of digits in b).

Basis When $n = 0$, $b = b_0$ and $longmult(a,b) = ab_0 = ab$ and the result follows.

Induction hypothesis (IH) Assume $longmult(a,c) = ac$ for all c which have less than $n + 1$ digits, i.e. $0 \leq c < 10^n$.

Induction Step Since $n > 0$ we have that $b > 0$. Then for $b_0 = b \ \overline{\cdot}\ 10$ and $b' = b \ \dot{-}\ 10$ we have

$$b = 10b' + b_0 \tag{4}$$

and hence

$$\begin{array}{rcll} longmult(a, b) & = & ab_0 + longmult(a, b') & \\ & = & ab_0 + ab'.10 & \text{(IH)} \\ & = & a(b_0 + 10b') & \text{(factoring out } a\text{)} \\ & = & ab & \text{(by the identity (4))} \end{array}$$

The above proof does not really change whether we use algorithm (2) or algorithm (3), since we would still be using the value of n as the induction variable.

1.2.3. The Effort of Long Multiplication

In trying to answer the last question we may ask ourselves what exactly is the time or effort involved in executing the algorithm by say a typical primary school student to whom this method is taught. The tools available with such a student are

- the ability to perform addition and
- the ability to perform single-digit multiplication

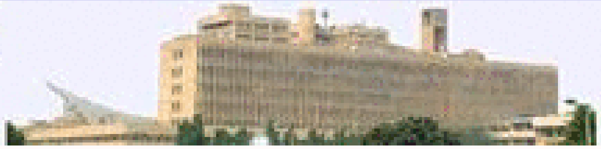
Therefore if a and b are numbers of $m+1$ digits and $n+1$ digits respectively, $n+1$ single digit multiplications of a would have to be performed resulting in $n+1$ intermediate products to be summed up i.e. n addition operations would have to be performed.

1.2.4. Comparison

At the primary school, multiplication of numbers is usually introduced as “repeated addition”. In effect this process could be described by the following algorithm.

$$repadd(a,b) = \begin{cases} 0 & \text{if } b = 0 \\ a + repadd(a, b-1) & \text{if } b > 0 \end{cases} \tag{5}$$

We leave the proof of correctness of this algorithm as an exercise for the interested reader and mention only that it is easy to see that for $b > 0$ this algorithm would require $b-1$ addition operations of numbers that are at least $m+1$ digits long, and is hence quite time-consuming for large values of b as compared to the algorithm (3) which uses the advantages of the place-value system to yield an algorithm which requires about $\log b$ additions and $\log b$ multiplications of a by a single digit. One wonders how the Romans ever managed to multiply even numbers which are less than 10000. Imagine what “MMMMDCCCLXXXVIII × MMMDCCLXXIX” would work out to be!



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: Summary

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

  Straight-edge: Can specify lines, rays and line segments.

  Compass: Can define arcs and circles. Can compare lengths along a line.

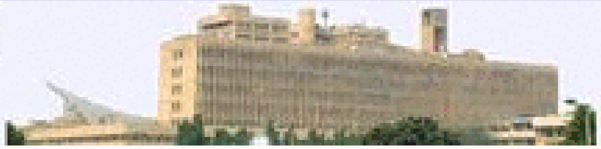
  Straight-edge and Compass: Can specify points of intersection.

- The primitives may be combined in various ways to perform a computation.
- Example Constructing a perpendicular 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.



1.3. Our Computing Tool

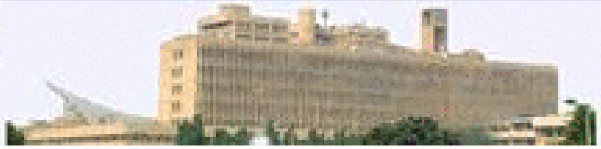
Previous: Introduction to Computing

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 called *pseudo-code*). 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.
- Even mathematical notation is often not quite precise and certain intermediate objects or steps are left to the understanding of the reader (e.g. the use of "$\cdots$" and "$\vdots$").

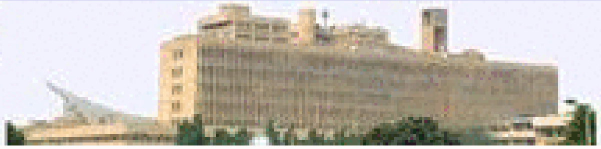
Functional Pseudo-Code

- *Functional pseudo-code* avoids most such implicit assumptions and attempts to make definitions more precise (e.g. the use of induction).
- *Functional pseudo-code* is still concise though more formal than standard mathematical notation.
- However *functional pseudo-code* is not formal enough to be termed a programming language (e.g. it does not satisfy strict grammatical rules and neither is it linear as most formal languages are).
- But *functional pseudo-code* is precise enough to analyse the correctness and complexity of an algorithm, whereas standard mathematical notation may mask many important details.

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 (called the syntax of the language).
- Each program has to be written following rigid grammatical rules (syntactic rules).
- A programming language and the computer together form our single computing tool
- Each program uses *only* the primitives of the computing tool

Programs

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

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

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

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

Programming

The act of writing programs and testing them is programming

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

Programming languages differ from each other in terms of the convenience and facilities they offer even though they are all equally “powerful” in terms of what they can actually compute.

Computing Models

We consider mainly two models.

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

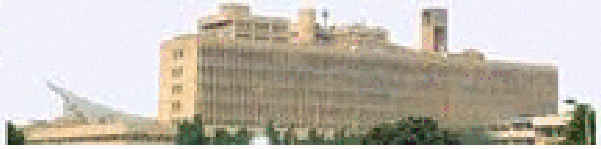
Programming languages also come mainly in these two flavours. We will often identify the computing model with the programming language.

Primitives

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

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

The functional model



Primitive expressions

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

- basic data elements: numbers, characters, truth values etc.
- basic operations on the data elements: addition, subtraction, multiplication, division, boolean operations, string operations etc.
- a naming mechanism for various 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
- Interactive and so answers are immediately available
- The closeness to mathematics makes it convenient for developing, testing, proving and analysing algorithms.

Standard ML

Explanations and Remarks

There are several models of computation available even on a modern digital computer. This is what makes the modern computer a universal computing mechanism. Most commonly available programming languages include Basic, C, C++ and Java. These programming languages are called *imperative* programming languages and have evolved mainly from the hardware of a modern digital computer. In many ways these languages remain very close to the details of hardware of a modern computer.

At the other end of the spectrum are models and programming languages which have evolved from mathematics and logic. Prominent among these are the *functional* programming languages. They may be loosely described as being closer to the language of mathematics than of modern digital hardware.

As we will see later both models have their distinct uses. The imperative models are much closer to the hardware facilities provided by modern digital computers. But a purely imperative model is too detailed and can become quite cumbersome to analyse and reason about. On the other hand, a purely functional model can become quite difficult to use with several existing hardware and software facilties especially when it comes to handling files, disks and other peripheral units.

Besides the above models there are other computing models too such as declarative, object-oriented, aspect-oriented models etc. which are outside the scope of discussion in this course.

We have preferred the functional model to other models mainly because this course is more about *computing* than about *computers* *per se*, though of course we use a modern digital computer to do most of the tasks in this course. Modern digital computers are tremendously complex objects – both in terms of hardware and software – and contain far more detail than a single mind can cope with at any instant of time. As a result, any user of the computer at any instant pays attention to only some of the facets of the computer. While writing documents a user only views the computer as a word processor, while using a spread-sheet the user sees it as a spread-sheet calculator and nothing else. A programmer views it simply as a language processing machine. The hardware and software together present an abstracted view of the machine to the user, depending on the use that is being made of the computer.

Computing however, involves not simply writing programs for a given model but analysing the methods employed and the properties of the methods and the programs as well. Our emphasis is on problem-solving – developing, proving and analysing algorithms and programs. These tasks are perhaps performed more rigorously and conveniently with the tools and the notation of mathematics available to us and it therefore is easier to present them in a concise and succinct manner and reason about them in a functional programming paradigm than through other paradigms. At the same time it is possible for the reader to dirty her hands by designing and running the code on a modern digital computer.

Mathematical Notation 1: Factorial

For simplicity we assume that $n! = 1$ for all integers less than 1. Then informally we may write it as

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

Mathematical Notation 2: Factorial

Or more formally we use *mathematical induction* to define it as

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

A Functional Program: Factorial

```
fun fact n = if n < 1 then 1 else n * fact (n-1)
```

Refer here for a more detailed explanation of the syntax of SML and how to understand its responses.

Factorial in SML

If we were to start the SML system on a terminal and input the above line, what we see is this

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

The lines starting with - are the response of the system to our input from the keyboard. Here SML has recognized that we have defined a function called `fact` which takes a single integer argument (formally referred to as `n` in the definition).

Evaluation: Factorial

If we were to give a further input `fact 4;` what we see on the terminal is the following.

```
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 4;
val it = 24 : int
-
```

An evaluation has been performed viz. `fact 4` whose result is an integer value 24.

Computations in General

A computation takes place by a process of *simplification* (called *evaluation*) which may be understood as follows:

- Evaluation requires replacement of

  simple variables by their values,

  function names by their definitions,

  argument names by the values of the actual arguments. In certain cases actual arguments may have to be first evaluated

- Evaluation takes place from left to right unless bracketing and association of operators dictate otherwise.
- Priority of operators is usually consistent with mathematical notation but may vary from language to language.

A Computation 1: Factorial

This process is akin to the way we compute in mathematics. For example, going strictly by the definition we have given we have the following *computations* for various values.

$$\begin{aligned} & (-2)! \\ = \ & 1 \end{aligned}$$

$$\begin{aligned} & 1! \\ = \ & 1 \times (1-1)! \\ = \ & 1 \times 0! \\ = \ & 1 \times 1 \\ = \ & 1 \end{aligned}$$

A Computation 2: Factorial

$$
\begin{aligned}
& 3! \\
= \; & 3 \times (3-1)! \\
= \; & 3 \times 2! \\
= \; & 3 \times (2 \times (2-1)!) \\
= \; & 3 \times (2 \times 1!) \\
= \; & 3 \times (2 \times (1 \times (1-1)!)) \\
= \; & 3 \times (2 \times (1 \times 0!)) \\
= \; & 3 \times (2 \times (1 \times 1)) \\
= \; & 3 \times (2 \times 1) \\
= \; & 3 \times 2 \\
= \; & 6
\end{aligned}
$$

The bracketing and association shown above are followed by the machine.

A Computation: Factorial

Computations performed by a machine are very similar to the way we would compute except that it is done so mechanically that even associative laws and commutative laws are not known to the machine. As a result the bracketing that we see above is strictly to be followed.

However what we actually see displayed in the SML system for various values of n is as follows:

```
- fact 3;
val it = 6 : int
- fact 8;
val it = 40320 : int
- fact 9;
val it = 362880 : int
-
```

The system does not show you the various steps of the computation.

Mathematical Notation 3: Factorial

How about this definition?

$$n = \begin{cases} 1 & \text{if } n < 1 \\ (n+1)!/(n+1) & \text{otherwise} \end{cases} \tag{6}$$

Mathematically correct since a definition implicitly defines a mathematical equality or identity. But . . .

Computationally Incorrect Definitions

The last definition is computationally incorrect! Since the system would execute "*blindly*" by a process of replacing every occurrence of $k!$ by the right hand-side of the definition (in this case it would be $(k+1)!/(k+1)$). So for $3!$ we would get

$$\begin{aligned} & 3! \\ = \; & (3+1)!/(3+1) \\ = \; & 4!/4 \\ = \; & ((4+1)!/(4+1))/4 \\ = \; & (5!/5)/4 \\ = \; & \vdots \\ = \; & \vdots \end{aligned}$$

which goes on forever!

Algorithms & Nontermination

- An algorithm should guarantee that all its computations *terminate*. By this criterion the definition given above is not an algorithm whereas the one given previously is an algorithm.
- In fact for all our algorithmic definitions we would have to prove guaranteed termination.
- It is possible to write definitions in any programming language whose computations do not terminate for some legally valid arguments.

1.4. Some Simple Algorithmic Definitions

The functional model is very close to mathematics; hence functional algorithms are easy to analyze in terms of correctness and efficiency. We will use the SML interactive environment to write and execute functional programs. This will provide an interactive mode of development and testing of our early algorithms. In the later chapters we will see how a functional algorithm can serve as a specification for development of algorithms in other models of computation.

In the functional model of computation every problem is viewed as an evaluation of a function. The solution to a given problem is specified by a complete and unambiguous functional description. Every reasonable model of computation must have the following facilities:

Primitive expressions which represent the simplest objects with which the model is concerned.

Methods of combination which specify how the primitive expressions can be combined with one another to obtain compound expressions.

Methods of abstraction which specify how compound objects can be named and manipulated as units.

In what follows we introduce the following features of our functional model:

1. The *Primitive* expressions.
2. Definition of one function in terms of another (substitution).

3. Definition of functions using conditionals.
4. Inductive definition of functions.

The Basic Sets

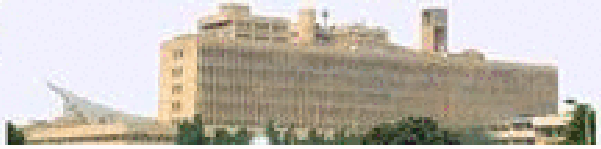
- $\mathbb{N}$: The set of *natural numbers* i.e. the *non-negative integers*
- $\mathbb{P}$: The set of *positive integers*
- $\mathbb{R}$: The set of *reals*
- $\mathbb{B}$: The set of *booleans* i.e. the truth values *true* and *false*

The Primitive Expressions:1

- Primitive functions of the type $f : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ and $f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ which we assume to be available in our functional model are addition $(+)$, subtraction $(-)$, multiplication $(*)$.
- We will also assume the div and mod functions of the type $f : \mathbb{N} \times \mathbb{P} \rightarrow \mathbb{N}$. Note that if $a \in \mathbb{N}$ and $b \in \mathbb{P}$ and $a = q * b + r$ for some integers $q \in \mathbb{N}$ and $0 \leq r < b$ then $div(a, b) = q$ and $mod(a, b) = r$.
- The division function $/ : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ will be assumed to be valid only for real numbers.

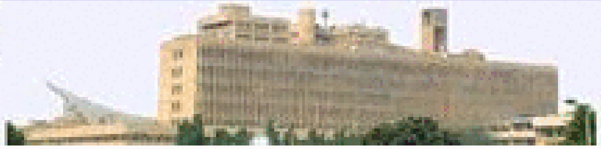
The Primitive Relations

- All relations are expressed as *boolean* valued functions.
- The *relations* $=$, $\leq$, $<$, $\geq$, $>$ and $\neq$. are functions of the type $f : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{B}$ or $f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{B}$ depending on the context,
- In addition, relations may be combined using the functions $\wedge : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ (and), $\vee : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ (or) and $\neg : \mathbb{B} \rightarrow \mathbb{B}$ (not).



1.5. Substitution of functions

We give a few examples of definitions of one function in terms of another and the evaluation of such functions through substitution.



Example: Square

Finding the square of a natural number.
We can directly specify the function $square$, which is of the type $square : \mathbb{N} \rightarrow \mathbb{N}$ in terms of the standard *multiplication* function $* : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ as

$$square(n) = n * n$$

Here, we are assuming that we can *substitute* one function for another provided they both return an item of the same type. To evaluate, say, $square(5)$, we have to thus evaluate $5 * 5$.

Example 1: Substitution

We can build more complex functions from simpler ones. As an example, let us define a function to compute $x^2 + y^2$.
We can define a function $sum\_squares : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ as follows

$$sum\_squares(x, y) = square(x) + square(y)$$

The function $sum\_squares$ is thus defined in terms of the functions $+$ and $square$.

Example 2: Substitution

As another example, let us consider the following function $f : \mathbb{N} \rightarrow \mathbb{N}$ defined as follows

$$f(n) = sum\_squares((n+1),(n+2))$$

Example 1: The Conditional

We give a few examples of function definitions using conditionals. Finding the absolute value of x. We define the function

$$abs : \mathbb{Z} \rightarrow \mathbb{N}$$

as

$$abs(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -x & \text{if } x < 0 \end{cases}$$

Example 2: The Conditional

Finding the larger of two numbers.

$$max : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$max(a, b) = \begin{cases} a & \text{if } a \geq b \\ b & \text{otherwise} \end{cases}$$

While defining the functions abs and max, we have assumed that we can compare two natural numbers, and determine which is larger. The basic primitive used in this case is *if-then-else*. Thus if $a \geq b$, the function max returns a as the output, else it returns b. Note that for every pair of natural numbers as its input, max returns a unique number as the output.

In the case of abs, even though there seem to be three possible cases, the basic primitive used is still the *if-then-else* which may be read as

$$\begin{array}{rll} abs(x) = x & \textit{if}\ x > 0, \\ \textit{otherwise}\ abs(x) = 0 & \textit{if}\ x = 0 \\ \textit{otherwise}\ abs(x) = -x & \textit{if}\ x < 0 \end{array}$$

Equivalently we may read it as

$$\begin{array}{lll} \textit{if} & x > 0\ \textit{then} & abs(x) = x \\ \textit{else if} & x = 0\ \textit{then} & abs(x) = 0 \\ \textit{else} & & abs(x) = -x \end{array}$$

What is *not* an Algorithm?

Not all mathematically valid specifications of functions are algorithms. For example,

$$sqrt(n) = \begin{cases} m & \text{if } m * m = n \\ 0 & \text{if } \nexists m : m * m = n \end{cases}$$

is mathematically a perfectly valid description of a function of the type $sqrt : \mathbb{N} \rightarrow \mathbb{N}$. However the mathematical description does not tell us *how* to evaluate the function, and hence it is not an algorithm.

Algorithmic Descriptions

An algorithmic description of the function may start with $m = 1$ and check if $m * m = n$ for all subsequent increments of m by 1 till either such an m is found or $m * m > n$. We will soon see how to describe such functions as computational processes or algorithms such that the computational processes terminate in finite time.

As another example of a mathematically valid specification of a function which is *not* an algorithm, consider the following functional description of $f : \mathbb{N} \rightarrow \mathbb{N}$

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n+1) - 1 & \text{otherwise} \end{cases} \tag{7}$$

The above specification has an infinite number of candidate solutions. For any constant $k \in \mathbb{N}$, the functions

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

are all solutions that satisfy the given specification (7). However, the specfcation (7) is not a valid algorithm because in order to evaluate $f(1)$ we have to evaluate $f(n+1)$ for $n = 1, 2, 3, \ldots$ which leads to an infinite computational process. One can rewrite the specification of the above function, in an *inductive* form, as a function $g : \mathbb{N} \rightarrow \mathbb{N}$

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

Now this indeed defines a valid algorithm for computing $f_1(n)$. Mathematically the specifications for $f_1(n)$

and $g(n)$ are equivalent in that they both define the same function. However, the specification for $g(n)$ constitutes a valid algorithm whereas that for $f(n)$ does not. For successive values of n, $g(n)$ can be computed as

$$\begin{array}{rcl} g(0) & = & 0 \\ g(1) & = & g(0)+1=1 \\ g(2) & = & g(1)+1=g(0)+1+1=2 \\ & \vdots & \end{array}$$

Exercise 1.1

1. *Prove that* $f_1(n) = g(n)$ *for all* $n \in \mathbb{N}$*.*
2. *For each* $k \in \mathbb{N}$*, give an algorithmic definition to compute the function* f_k*.*

Similarly, consider the definition

$$f(n) = f(n)$$

Every (unary) function is a solution to the above but it is computationally undefined.

1.6. Functions as inductively defined computational processes

All the examples of algorithmic descriptions we have presented so far are of functions which can be evaluated by substitutions or by evaluation of conditions. We give a few more examples of algorithms which are defined by induction.

1.7. Algorithmic Definitions which are well-founded

We give an example of an algorithm for computing the *greatest common divisor* (GCD) of two positive integers. This algorithm is a variation of the one invented by Euclid and at first sight does not appear to be inductive at all. But in fact it uses a generalization of mathematical induction called well-founded induction.

Example 1.2 *Computing the GCD of two positive integers. We know that the greatest common divisor is a function which for any pair of non-negative integers, yields the greatest of all the common divisors of the two numbers.*

Assume a and b are non-negative integers.

$$gcd(a,b) = \begin{cases} m & \text{if } a = 0 \vee b = 0 \vee m = n \\ gcd(m-n, n) & \text{otherwise} \end{cases} \tag{8}$$

where $m = max(a,b)$ and $n = min(a,b)$. Here and everywhere else in these notes $\vee$ stands for (inclusive) “or” of conditions. Various questions arise regarding whether

1. as an identity, equation (8) correctly defines a property of the greatest common divisor function for non-negative integers.
2. whether the identity may be used as an algorithm which is guaranteed to terminate i.e. does it define a finite computational process?

The following theorems could settle the two questions.

Theorem 1.3 **Correctness of GCD identity***. For any two non-negative integers a and b,*

$$gcd(a,b) = \begin{cases} m & \text{if } a = 0 \vee b = 0 \vee m = n \\ gcd(m-n, n) & \text{otherwise} \end{cases}$$

where $m = max(a,b)$ and $n = min(a,b)$.

Proof:

Case $a = 0 \vee b = 0 \vee m = n$**.** *Notice that $a, b \geq 0$ and if one of them is 0 then the other is indeed the greatest common divisor, since every integer is a divisor of 0. On the other hand, $m = max(a,b) = n = min(a,b)$ is possible if and only if $a = b$. Then again we have $gcd(a,b) = a = b = m = n$.*

Case $a \neq 0 \neq b$ **and** $m \neq n$**.** *In this case we have $a, b > 0$ and hence $m > n > 0$. Without loss of generality we assume $a > b > 0$ (since the case $b > a > 0$ can be proven similarly). Then $m = max(a,b) = a$ and $n = min(a,b) = b$. Further $m - n = a - b$. We then prove the following claims.*

1. *Claim. Every common divisor of a and b is also a common divisor of $a - b$ and b.*

Proof: Since $a, b > 0$, they have at least one positive common divisor since 1 is a divisor of every non-negative integer. Let $d \geq 1$ be a common divisor of a and b. Then there exist $a', b' > 0$ such that $a = da'$ and $b = db'$. We then have $a - b = d(a' - b')$. Hence d is a common divisor of $a - b$ and b.

2. *Claim.* $gcd(a, b) = gcd(a - b, b)$.

   Proof: Let $h = gcd(a, b)$ and let $d > 0$ be any common divisor of a and b. Clearly $h \geq d$ and by the previous claim h is also a common divisor of $a - b$ and b and indeed the greatest of them.

The above theorem justifies that equation (8) is a valid mathematical identity. The next theorem shows that equation (8) does define a finite computational process.

Theorem 1.4 *Equation (8) defines a finite computational process for non-negative integers.*

Proof:

Case $a = 0 \vee b = 0 \vee m = n$**.** In all cases when $a = b$ or one of the two numbers is 0 it is clear that the first condition of the identity immediately yields a result in one step.

Case $a \neq 0 \neq b$ **and** $m \neq n$**.** In this case $a, b, m, n > 0$ and we may define a measure given by m, and it is easy to see that $a > m - n > 0$ whenever $a > b$ and $b > m - n > 0$ whenever $b > a$. It is then clear that $max(a, b) > m - n \geq 0$ always and hence the computational process is bounded above by $max(a, b)$ and below by 0 and the larger of the two arguments in the computational process traverses down a subset of elements from the strictly decreasing finite sequence $m, m - 1, m - 2, \ldots, 1, 0$, and hence is guaranteed to terminate in a finite number of steps.

gcd as defined by equation (8) is a function because for every pair of non-negative integers as arguments, *it yields a non-negative integer. It is also a finite computational process, because given any two non-negative integers as input, the description tells us, unambiguously, how to compute the solution and the process terminates after a finite number of steps. For instance, for the specific case of computing* $gcd(18, 12)$*, we have*

$$gcd(18, 12) = gcd(12, 6) = gcd(6, 6) = 6.$$

The larger of the two arguments in this computational process for instance goes through a subset of the elements in the finite sequence

$$18, 17, \ldots, 6, 5, 4, 3, 2, 1$$

.

Thus **a specification of a function is an algorithm** only if it actually defines a finite computational process to evaluate it and this computational process may not always be inductive, though it must be well-founded in order that the process terminates.

Algorithmic Descriptions

Any of the following constitutes an algorithmic description:

1. It is directly specified in terms of a pre-defined function which is either primitive or there exists an algorithm to compute it.
2. It is specified in terms of the evaluation of a condition.
3. It is inductively defined and the validity of its description can be established through the Principle of Mathematical Induction or through well-founded induction.
4. It is obtained through any finite number of combinations of the above three steps using substitutions.

Complex functions can be algorithmically defined in terms of two main types of processes - *recursive* and *iterative*.

Recursive computational processes are characterized by a chain of deferred operations. As an example, we will consider an algorithm for computing the factorial of non-negative integer n.

Example 1.5

$$factorial : \mathbb{N} \rightarrow \mathbb{N}$$

$$factorial(n) = \begin{cases} 1 & \textit{if } n = 0 \\ n \times factorial(n-1) & \textit{otherwise} \end{cases}$$

It is instructive to examine the computational process underlying the above definition. The computational process, in the special case of $n = 5$, looks as follows

$$\begin{aligned} & factorial(5) \\ & \quad = (5 \times factorial(4)) \\ & \quad = (5 \times (4 \times factorial(3))) \\ & \quad = (5 \times (4 \times (3 \times factorial(2)))) \\ & \quad = (5 \times (4 \times (3 \times (2 \times factorial(1))))) \\ & \quad = (5 \times (4 \times (3 \times (2 \times (1 \times factorial(0)))))) \\ & \quad = (5 \times (4 \times (3 \times (2 \times (1 \times 1))))) \end{aligned}$$

$$
\begin{aligned}
&= (5 \times (4 \times (3 \times (2 \times 1)))) \\
&= (5 \times (4 \times (3 \times 2))) \\
&= (5 \times (4 \times 6)) \\
&= (5 \times 24) \\
&= 120
\end{aligned}
$$

A computation such as this is characterized by a *growing* and *shrinking* process. In the *growing* phase each "call" to the function is replaced by its "body" which in turn contains a "call" to the function with different arguments. In order to compute according to the inductive definition, the actual multiplications will have to be postponed till the base case of $factorial(0)$ can be evaluated. This results in a growing process. Once the base value is available, the actual multiplications can be carried out resulting in a shrinking process. Computational processes which are characterized by such "deferred" computations are called *recursive*. This is not to be confused with the notion of a *recursive procedure* which just refers to the syntactic fact that the procedure is described in terms of itself.

Note that by a computational process we require that a machine, which has only the capabilities provided by the computational model, be able to perform the computation. A human normally realizes that multiplication is commutative and associative and may exploit it so that he does not have to defer performing the multiplications. However if the multiplication operation were to be replaced by a non-associative operation then even the human would have to defer the operation. Thus it becomes necessary to perform all recursive computations through deferred operations.

Exercise 1.2 Consider the following example of a function $f : \mathbb{N} \to \mathbb{Z}$ defined just like factorial except that multiplication is replaced by subtraction which is not associative.

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

1. Unfold the computation, as in the example of $factorial(5)$ above, to show that $f(5) = 2$.
2. What properties will you use as a human computer in order to avoid deferred computations?

1.9. Correctness

The correctness of the functional algorithm defined in example 1.5 can be established by using the Principle of Mathematical Induction (**PMI**). The algorithm adheres to an inductive definition and, consequently, can be proved correct by using **PMI**. Even though the proof of correctness may seem obvious in this instance, we give the proof to emphasize and clarify the distinction between a mathematical specification and an algorithm that implements it.

To show that: For all $n \in \mathbb{N}$, $factorial(n) = n!$ (i.e., the function $factorial$ implements the factorial function as defined below).

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1 \times 2 \times \ldots \times n & \text{otherwise} \end{cases}$$

Proof: By **PMI** on n.

Basis. When $n = 0$, $factorial(n) = 1 = 0!$ by definitions of $factorial$ and $0!$.

Induction hypothesis. For $k = n - 1$, $k \geq 0$, we have that $factorial(k) = k!$.

Induction step. Consider $factorial(n)$.

$$\begin{aligned} factorial(n) &= n \times factorial(n-1) & \\ &= n \times (n-1)! & \text{by the induction hypothesis} \\ &= n! & \text{by the definition of } n! \end{aligned}$$

Hence the function $factorial$ implements the factorial function $n!$.

Exercise 1.3 *In the case of factorial we proved that the recursive definition of factorial equals a non-recursive specification for all natural numbers n. Consider the function f defined in exercise 1.2.*

1. *What does the function f actually compute (give a non-recursive description)?*
2. *Prove that your non-recursive specification equals the recursive specification for all naturals n.*

1.10. Complexity

The other important aspect in the analysis of an algorithm is the issue of efficiency - both in terms of *space* and *time*. The efficiency of an algorithm is usually measured in terms of the space and time required in the execution of the algorithm (the space and time complexities). These are functions of the input size n.

A careful look at the above computational process makes it obvious that in order to compute $factorial(n)$, the n integers will have to be remembered (or stacked up) before the actual multiplications can begin. Clearly, this leads to a space requirement of about n. We will call this the *space complexity*.

The time required to execute the above algorithm is directly proportional (at least as a first approximation) to the number of multiplications that have to be carried out and the number of function calls required. We can evaluate this in the following way. Let $\mathcal{T}(factorial(n))$ be the number of multiplications required for a problem of size n (when the input is n). Then, from the definition of the function $factorial$ we get

$$\mathcal{T}(factorial(n)) = \begin{cases} 0 & \text{if } n = 0 \\ 1 + \mathcal{T}(factorial(n-1)) & \text{otherwise} \end{cases} \qquad (9)$$

$\mathcal{T}(factorial(0))$ is obviously 0, because no multiplications are required to output $factorial(0) = 1$ as the result. For $n > 0$, the number of multiplications required is one more than that required for a problem of size $n - 1$. This is a direct consequence of the recursive specification of the solution. We may solve Equation 9 by telescoping, i.e.,

$$\begin{aligned} \mathcal{T}(factorial(n)) &= 1 + \mathcal{T}(factorial(n-1)) \qquad (10) \\ &= 2 + \mathcal{T}(factorial(n-2)) \\ &\ \ \vdots \end{aligned}$$

$$\begin{aligned} &= n + \mathcal{T}(factorial(0)) + n \\ &= n \end{aligned}$$

Thus n is the number of multiplications required to compute $factorial(n)$ and this is the time complexity of the problem.

To estimate the space complexity, we have to estimate the number of deferred operations which is about the same as the number of times the function $factorial$ is invoked.

Exercise 1.4 Show, in a similar way, that the number of invocations of $factorial$ required to evaluate $factorial(n)$ is $n+1$.

Equation 9 is called a recurrence equation and we will use such equations to analyze the time complexities of various algorithms in these notes. Note that the measure of space and time given above are independent of how fast a computing machine is. Rather, it is given in terms of the amount of space required and the number of multiplications and function calls that are required. The measures are thus independent of any computing machine.

1.11. Efficiency, Why and How?

Modern technological advances in silicon have seen processor sizes fall and computing power rise dramatically. The microchip one holds in the palm today packs more computing power – both processor speed and memory size – than the monster monoliths that occupied whole rooms in the 50's. A sceptic is therefore quite entitled to ask: who cares about efficient algorithms anyway? If it runs too slow, just throw the processor away and get a larger one. If it runs out of space, just buy a bigger disk! Let's perform some simple back-of-the-envelope calculations to see if this scepticism is justified.

Consider the problem of computing the *determinant* of a matrix, a problem of fundamental importance in numerical analysis. One method is to evaluate the determinant by the well known formula:

$$\det A = \sum_{\sigma} (-1)^{\mathrm{sgn}\sigma} A_{1,\sigma(1)} \cdot A_{2,\sigma(2)} \cdots A_{n,\sigma(n)}.$$

Suppose you have implemented this algorithm on your laptop to run in $10^{-4} \times 2^n$ seconds when confronted with any $n \times n$ matrix (it will actually be worse than this!). You can solve an instance of size 10 in $10^{-4} \times 2^{10}$ seconds, i.e., about a tenth of a second. If you double the problem size, you need about a thousand times as long, or, nearly 2 minutes. Not too bad. But to solve an instance of size 30 (not at all an unreasonable size in practice), you require a thousand times as long again, i.e. even running your laptop the whole day isn't sufficient (the battery would run out long before that!). Looking at it another way, if you ran your algorithm on your laptop for a whole year (!) without interruption, you would still only be able to compute the detrminant of a 38×38 matrix!

Well, let's buy new hardware! Let's go for a machine that's a hundred times as fast – now this is getting almost into supercomputing range and will cost you quite a fortune! What does it buy you in computing power? The same algorithm now solves the problem in $10^{-6} \times 2^n$ seconds. If you run it for a whole year non–stop (let's not even think of the electricity bill!), you can't even compute a 45×45 determinant! In practice, we will routinely encounter much larger matrices. What a waste!

Exercise 1.5 *In general, show that if you were previously able to compute $n \times n$ determinants in some given time (say a year) on your laptop, the fancy new supercomputer will only solve instances of size*

$n + \log 100$ *or about* $n + 7$ *in the same time.*

Suppose that you've taken this course and invest in algorithms instead. You discover the method of *Gaussian elimination* (we will study it later in these notes) which, let us assume, can compute a $n \times n$ determinant in time $10^{-2}n^3$ on your laptop. To compute a 10×10 determinant now takes 10 seconds, and a 20×20 determinant now requires between one and two minutes. But patience! It begins to pay off later: a 30×30 determinant takes only four and a half minutes and in a day you can handle 200×200 determinants. In a years's computation, you can do monster 1500×1500 determinants.

1.12. Asymptotic analysis and Orders of growth

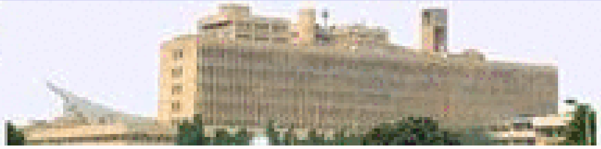
You may have noticed that there was something unsatisfactory about our way of doing things – the calculation was tuned too closely to our machine. The figure of $10^{-4} \times 2^n$ seconds is a bit arbitrary – the time to execute on one particular laptop – and has no other absolute significance for an analysis on a different machine. We would like to remedy this situation so as to have a mode of analysis that is applicable to *any* machine. It should tell us precisely how the problem *scales* – how does the resource requirement grow as the size of the input increases – on any machine.

We now introduce one such machine independent measure of the resources required by a computational process – the *order of growth*. If n is a parameter that measures the size of a problem then we can measure the resources required by an algorithm as a function $R(n)$. We say that the function $R(n)$ has an order of growth $O(f(n))$ (of order $f(n)$), if there exist constants K and n_0 such that $R(n) \leq Kf(n)$ whenever $n \geq n_0$.

In our example of the computation of $factorial(n)$, we found that the space required is n, whereas the number of multiplications and function calls required are n and $n+1$ respectively. We see, that according to our definition of order of growth, each of these are $O(n)$. Thus, we can say that the *space complexity* and the *time complexity* of the algorithm are both $O(n)$. In the example of determinant computation, regardless of the particular machine and the corresponding constants, the algorithm based on Gaussian elimination has time complexity $O(n^3)$.

Order of growth is only a crude measure of the resources required. A process which requires n steps and another which requires $1000n$ steps have both the same order of growth $O(n)$. On the other hand, the $O(\cdot)$ notation has the following advantages:

- It *hides constants*, thus it is *robust across different machines*.
- It gives fairly precise indication of how the algorithm *scales* as we increase the size of the input. For example, if an algorithm has an order of growth $O(n)$, then doubling the size of the input will very nearly double the amount of resources required, whereas with a $O(n^2)$ algorithm will *square* the amount of resources required.
- It tells us which of two competing algorithms will win out eventually *in the long run*: for example, however large the constant K may be, it is always possible to find a break point above which Kn will always be smaller than n^2 or 2^n giving us an indication of when an algorithm with the former complexity will start working better than algorithms with the latter complexities.
- Finally the very fact that it is a crude analysis means that it is frequently much easier to perform than an exact analysis! And we get all the advantages listed above.



1.13. The Analysis of Algorithms: The Big "O" notation

Analysis of an algorithm means analysing the resources required by the algorithm. The resources here are primarily the amount of space required to store values (including intermediate ones) and the amount of the time required to execute the algorithm, in terms of a number of "steps". The notion of a step requires some units as one machine's step may be different from another machine's step. We may even consider for the puproses of asymptotic analysis, a large-ish step such as the number of invocations of a function in order to simplify analysis.

There are other situations where other resources such as the band-width, the number of logic-gates or the number of messages sent may be useful measures for analysing the resources required by a system. However in the case of our algorithms our primary concern will be with the requirement of time and storage space required in the worst case by the algorithm.

In general, both measures depend upon the value or length of the input. It is normal in most books on Algorithms to launch into a rather long excursus on the machine model used to analyse the algorithm. We will however dispense with such things since everybody who reads these notes probably has a fair idea of the digital computer and anyway nobody reads that kind of trash.

Hence in our analysis we will primarily make use of functions $f : \mathbb{N} \longrightarrow \mathbb{R}$ and try to analyse the running time and the space requirements of algorithms in terms of these functions and compare the complexity of different algorithms for the same problem in terms of the asymptotic behaviour of functions which characterise their space requirements and/or running time as functions of the input size.

But we need to formalise these notions of asymptotic analysis. Asymptotic analysis involves determining the growth-rate of a function for large values. Such a growth rate could easily be determined by computing the first derivative of the function if the function were continuously differentiable for large values. However, since our functions depend upon the input size, they turn out to be functions whose domain is the set of natural numbers rather than real numbers. Hence these "*discrete*" functions are not easily amenable to the tools of differential and integral calculus.

In figures 1, 2 we show plots of some of these functions. Notice that the scale has been chosen to make them look continuous and differentiable, but they are indeed discrete functions. Further note that the y-axis in figure 2 is in logarithmic scale to accommodate the various functions.

Asymptotically Non-negative Functions

Definition 1.6 *A function $f : \mathbb{N} \longrightarrow \mathbb{R}$ is said to be* **asymptotically non-negative** *if for some $k \geq 0$ and for all $m \geq k$, $f(m) \geq 0$.*

That is, $f : \mathbb{N} \longrightarrow \mathbb{R}$ is asymptotically non-negative if for all sufficiently large values of n, $f(n)$ is guaranteed to be non-negative.

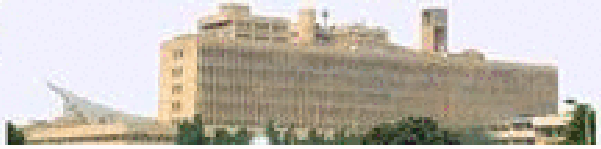
Example 1.7 *The function $f(n) = n^2 - 100$ is asymptotically non-negative for all values of $n \geq k = 10$.*

Exercise 1.6 *Is $f(n) = \sin n$ an asymptotically non-negative function? What about $g(n) = 1 + \sin n$ and $h(n) = 2 + \cos n$?*

Let

$$\boxed{\mathbf{\Phi} = \{f : \mathbb{N} \longrightarrow \mathbb{R} \mid f \text{ is asymptotically non-negative}\}}$$

be the set of all asymptotically non-negative functions.



The Order Θ

Definition 1.8 *For any given function* $g : \mathbb{N} \longrightarrow \mathbb{R}$*, that is asymptotically non-negative,* $\Theta(g(n)) \subseteq \mathbf{\Phi}$ *is the class of functions such that*

$$\Theta(g(n)) = \{f(n) \mid \exists c_0, c_1 \in \mathbb{R}, n_0 > 0 : \forall n \geq n_0 : 0 \leq c_0.g(n) \leq f(n) \leq c_1.g(n)\}$$

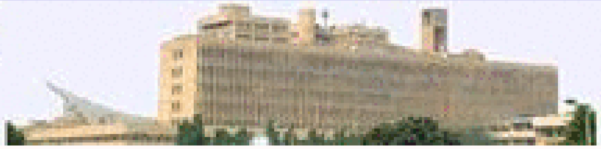
For any $f(n) \in \Theta(g(n))$ *we write (by abuse of notation)* $f(n) = \Theta(g(n))$*.*

If $f(n)$ can be bounded from above and below by scalar multiples of $g(n)$ for all sufficiently large values of n then $f(n) \in \Theta(g(n))$.

Exercise 1.7

1. Is $1 + \sin n = \Theta(1 + \cos n)$*?*

2. Is $2 + \sin n = \Theta(2 + \cos n)$



Θ: Simple Examples: 1

Example 1.9 *Given the function* $g(n) = 1$ *every constant function* $f(n) = k \geq 0$ *is a member of* $\Theta(g(n))$.

Example 1.10 *Given a linear function* $g(n) = 3n + 2$, *every linear function of* n *belongs to* $\Theta(g(n))$. *For example for a linear function* $f(n) = 10^6 n + 10^5$ *there exist constants* $c_0 = 1$, $c_1 = 10^6$ *and an* $n_0 = 1$, *such that for all* $n \geq n_0$, *we have* $0 \leq c_0.g(n) \leq f(n) \leq c_1.g(n)$

Θ: Simple Examples: 2

Example 1.11 *The function $f(n) = 3n^2 + 4n - 10 \notin \Theta(g(n^3))$ because it is impossible to find some n_0 which will guarantee for some c_0 that $c_0.g(n) \leq f(n)$ holds for all $n \geq n_0$.*

Example 1.12 *For any function $f(n) = 4n^2 + 8n + 2 \in \Theta(n^2)$. However $f(n) \notin \Theta(n)$ and $f(n) \notin \Theta(n^k)$ for any constant $k > 2$.*

The above example shows that lower-order terms become asymptotically insignificant when n is large.

Exercise 1.8 *Let $f(n) = an^2 + bn + c \in \mathbf{\Phi}$ for suitably chosen coefficients a, b and c. Determine values of c_0, c_1 and n_0 as functions of a, b and c such that $f(n) \in \Theta(n^2)$.*

Θ: The Polynomials

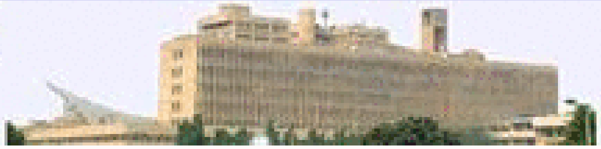
Example 1.13 *Every polynomial (in n) of degree $k \geq 0$ belongs to the class $\Theta(n^k)$.*

Notice that in definition 1.8, it is necessary that $g(n)$ be asymptotically non-negative and every member of $\Theta(g(n))$ is also asymptotically non-negative.

Θ: Some Simple Facts

Fact 1.14

1. *Every asymptotically non-negative function* $g(n)$ *is a member of* $\Theta(g(n))$*.*
2. *If* $f(n) \in \Theta(g(n))$ *then* $g(n) \in \Theta(f(n))$*.*
3. *If* $f(n) \in \Theta(g(n))$ *and* $g(n) \in \Theta(h(n))$ *then* $f(n) \in \Theta(h(n))$



Operations

Theorem 1.15 *Let $d(n)$, $e(n)$, $f(n)$ and $h(n)$ be asymptotically non-negative functions.*

1. If $d(n) \in \Theta(f(n))$ then $a.d(n) \in \Theta(f(n))$ for any real constant a.

2. If $d(n), e(n) \in \Theta(f(n))$ then $d(n) + e(n) \in \Theta(f(n))$.

3. If $d(n) \in \Theta(f(n))$ and $e(n) \in \Theta(g(n))$ then

(a) $d(n) + e(n) \in \Theta(f(n) + g(n))$

(b) $d(n).e(n) \in \Theta(f(n).g(n))$

The facts 1.14 and theorem 1.15 must convince us that there exists an equivalence relation on asymptotically non-negative functions of n which reflects the fact that for large values of n the various functions separate out into various classes that do not intermingle, i.e. for example

1. all linear functions regardless of their coefficients remain in the same class and all quadratic functions remain in the same class, and
2. linear functions of n do not mingle with quadratic functions of n and polynomials of degree k do not mingle with those of degree k' for $k \neq k'$.

$=_\Theta$: An Equivalence

Definition 1.16 *Let* $=_\Theta \subseteq \ \boldsymbol{\Phi} \times \boldsymbol{\Phi}$ *be the binary relation such that* $f(n) =_\Theta g(n)$ *if and only if* $f(n) \in \Theta(g(n))$*.*

From facts 1.14 it follows easily that

Corollary 1.17

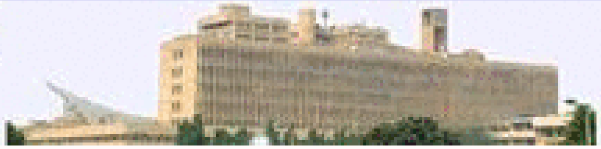
1. $=_\Theta$ *is* reflexive *i.e.* $f(n) =_\Theta f(n)$ *for each* $f(n) \in \boldsymbol{\Phi}$*.*

2. $=_\Theta$ *is* symmetric *i.e. for each* $f(n), g(n) \in \boldsymbol{\Phi}$*,* $f(n) =_\Theta g(n)$ *if and only if* $g(n) =_\Theta f(n)$*.*

3. $=_\Theta$ *is* transitive *i.e. for each* $f(n), g(n), h(n) \in \boldsymbol{\Phi}$*,* $f(n) =_\Theta g(n)$ *and* $g(n) =_\Theta h(n)$ *implies* $f(n) =_\Theta h(n)$

4. $=_\Theta$ *is an equivalence relation on* $\boldsymbol{\Phi}$*.*

I t will be fairly common for us now to incorporate Θ within our calculations. For example we may use expressions like $n^2 + \Theta(n)$ to mean $n^2 + f(n)$ where $f(n)$ is a linear function of n whose exact expression is either unknown or which is of no importance to us.



Θ in Equations

We may then use Θ in “equations” as follows.

$$\begin{aligned} 4n^3 + 8n^2 - 3n + 10 &= 4n^3 + \Theta(n^2) \\ &= 4\Theta(n^3) \\ &= \Theta(n^3) \end{aligned}$$

where we actually mean

$$\begin{aligned} 4n^3 + 8n^2 - 3n + 10 &=_\Theta 4n^3 + f(n) && \text{for any } f(n) \in \Theta(n^2) \\ &=_\Theta 4.g(n) && \text{for any } g(n) \in \Theta(n^3) \\ &=_\Theta h(n) && \text{for any } h(n) \in \Theta(n^3) \end{aligned}$$

The Big O

While the Θ notation asymptotically bounds from both above and below, we are often mostly interested in what might be called the "worst-case" scenario in the analysis of an algorithm. We would like to simply give an upper bound based on the "worst-case" input. Any Θ bound based on the worst-case would not automatically bound all cases of input. It is simpler to take the worst-case bound and only bound it from above rather than below.

Definition 1.18 *For any given function $g : \mathbb{N} \longrightarrow \mathbb{R}$, that is asymptotically non-negative, $O(g(n)) \subseteq \mathbf{\Phi}$ is the class of functions such that*

$$O(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}, n_0 > 0 : \forall n \geq n_0 : 0 \leq f(n) \leq c.g(n)\}$$

For any $f(n) \in O(g(n))$ we write (by abuse of notation) $f(n) = O(g(n))$.

The "O" notation provides an asymptotic way of saying that a function is "less than or equal to" another function

The Big o: Examples

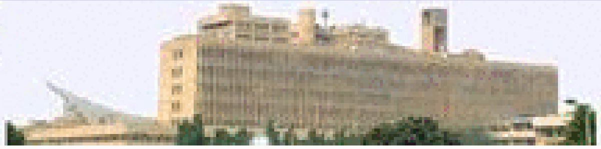
Example 1.19 *The combination of definition 1.18 and confusing notation can yield the following strange looking facts.*

1. $n = O(n)$ *but* $n \neq O(1)$
2. $n = O(n \log n)$ *but* $n \neq O(\log n)$
3. $n = O(n^2)$
4. $n = O(2^n)$

The Big O and Θ

Fact 1.20

1. *For any* $g(n) \in \mathbf{\Phi}$, $\Theta(g(n)) \subseteq O(g(n))$. *Hence If* $f(n) \in \Theta(g(n))$ *then* $f(n) \in O(g(n))$ *for any* $f(n) \in \mathbf{\Phi}$.
2. *If* $f(n) = O(g(n))$ *then* $O(f(n)) \subseteq O(g(n))$.
3. *If* $f(n) = O(g(n))$ *and* $g(n) = O(f(n))$ *then* $f(n) = \Theta(g(n))$ *and* $g(n) = \Theta(f(n))$.
4. *If* $f(n) = O(g(n))$ *but* $g(n) \neq O(f(n))$ *then* $O(f(n)) \subset O(g(n))$.



The Big o: More Facts

Theorem 1.21 *Let* $d(n)$*,* $e(n)$*,* $f(n)$ *and* $h(n)$ *be asymptotically non-negative functions.*

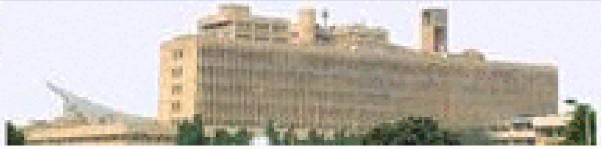
1. If $d(n) = O(f(n))$ *then* $a.d(n) = O(f(n))$ *for any real constant* a*.*

2. If $d(n) = O(f(n))$ *and* $e(n) = O(g(n))$ *then*

(a) $d(n).e(n) = O(f(n).g(n))$

(b) $d(n) + e(n) = O(f(n) + g(n))$

3. If $d(n) = O(f(n))$ *and* $f(n) = O(g(n))$ *then* $d(n) = O(g(n))$



The Relation $\leq_O$

Compare fact 1.20(2) and fact 1.14(2).
Analogous to the equivalence relation $=_\Theta$ we have

Definition 1.22

- *For all functions* $f(n), g(n) \in \mathbf{\Phi}$, $f(n) \leq_O g(n)$ *if and only if* $f(n) \in O(g(n))$. *Further,*
- $f(n) <_O g(n)$ *if and only if* $f(n) \leq_O g(n)$ *and* $g(n) \notin O(f(n))$.

While Θ defines an equivalence relation $=_\Theta$, O actually defines an ordering on functions. Just look at the strange facts and also look at the facts 1.20.

$\leq_O$: A Quasi Order

Fact 1.23

1. $\leq_O$ *is* reflexive *i.e.* $f(n) \leq_O g(n)$ *for all* $f(n)$.
2. $f(n) \leq_O g(n)$ *and* $g(n) \leq_O f(n)$ *implies* $f(n) =_\Theta g(n)$ *for all* $f(n), g(n) \in \mathbf{\Phi}$.
3. $\leq_O$ *is* transitive *i.e.* $f(n) \leq_O g(n)$ *and* $g(n) \leq_O h(n)$ *implies* $f(n) \leq_O h(n)$

Fact 1.23(2) also relates $=_\Theta$ and $\leq_O$.

Big O in Equations

In a fashion similar to whatever we did in the case of Θ we may also incorporate "O" in our equations.

$$\begin{aligned} 4n^3 + 8n^2 - 3n + 10 &= 4n^3 + O(n^2) \\ &= 4O(n^3) \\ &= O(n^3) \end{aligned}$$

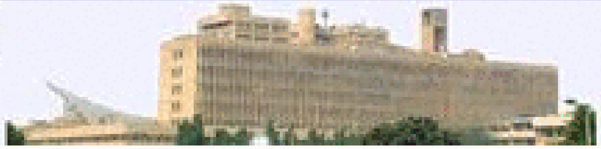
2. More examples of recursive algorithms

Now that we have established methods for analyzing the correctness and efficiency of algorithms, let us consider a few more examples of fundamental recursive algorithms.

Example: Powering - Algorithm

Example 2.1 *Computing x^n: Given an integer or a real number $x \neq 0$, compute x^n, where $n \geq 0$ is an integer.*

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x.power(x, n-1) & \text{otherwise} \end{cases} \tag{11}$$



Example: Powering - Correctness & Complexity

Here again it is easily established (by induction on n) that

Correctness. $power(x,n) = x^n$ for all $x \neq 0$ and $n \geq 0$.

Complexity The number of multiplications required is given by the recurrence

$$\begin{aligned}\mathcal{T}(power(x,n)) &= \begin{cases} 0 & \text{if } n = 0 \\ 1 + \mathcal{T}(power(x, n-1)) & \text{if } n > 0 \end{cases} \\ &= n \\ &= O(n)\end{aligned}$$

and the space complexity assuming that $\mathcal{S}(power(x,0)) = 1$ (at least one unit of space is required for the answer) is for $n > 0$, $\mathcal{S}(power(x,n)) = 1 + \mathcal{S}(power(x,n-1)) = n + 1 = O(n)$. The number of invocations of $power$ is given by the same recurrence as the space complexity (i.e. there is at least one invocation of $power$).

Example: Fast Powering

Example 2.2 *A faster method of powering depends upon the following correctness argument with the same conditions on x and n as in example 2.1.*

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ (x^2)^{n \overset{\cdot}{-} 2} & \text{if } n > 0 \text{ is even} \\ x.(x^2)^{n \overset{\cdot}{-} 2} & \text{if } n > 0 \text{ is odd} \end{cases} \qquad (12)$$

Note that an integer n is even if and only if $n \;\overline{\cdot}\; 2 = 0$ and is odd otherwise. We then obtain the following algorithm.

$$fastpower(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ fastpower(x.x, n \overset{\cdot}{-} 2) & \text{if } n > 0 \wedge n \;\overline{\cdot}\; 2 = 0 \\ x.fastpower(x.x, n \overset{\cdot}{-} 2) & \text{otherwise} \end{cases} \qquad (13)$$

Fast Powering: Correctness

We may then establish the following by induction on n.
It suffices to show by induction on n that the functions defined by equation 12 and algorithm 13 are the same for all values of n.
The computation of $fastpower$ in the worst case (i.e. when the exponent always remains odd till it reaches a value 1 at the k-th step), proceeds as follows.

Fast Powering: Worst-case Computation

$$\begin{array}{ll}
 & fastpower(x_0, n_0) \\
\leadsto & x_0.fastpower(x_1, n_1) \\
\leadsto & x_0.(x_1.fastpower(x_2, n_2)) \\
 & \vdots \\
\leadsto & x_0.(x_1.(\cdots(x_{k-2}.fastpower(x_{k-1}, n_{k-1}))\cdots)) \\
\leadsto & x_0.(x_1.(\cdots(x_{k-2}.x_{k-1}.fastpower(x_k, 0))\cdots)) \\
\leadsto & x_0.(x_1.(\cdots(x_{k-2}.(x_{k-1}.1)\cdots))) \\
\leadsto & x_0.(x_1.(\cdots.x_{k-1})\cdots) \\
\leadsto & \vdots
\end{array}$$

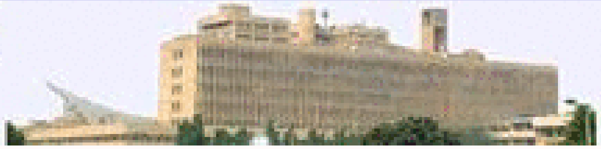
Fast Powering: Worst-case Computation

where $x_0 = x$, $n_0 = n$ and for all $i \geq 1$, $x_{i+1} = x_i^2$, $n_{i+1} = n_i \mathbin{\dot{-}} 2$ and n_i is odd and finally $n_{k-1} = 1$. It is easy to see that

Fact 2.3 *If $n_0 = 2^k - 1$ for some $k > 0$, then $n_i = n_{i-1} \mathbin{\dot{-}} 2 = 2^{(k-i)} - 1$ for each i, $0 < i \leq k$.*

The above fact may be proven by induction on i and is left as an easy exercise to the student.

Fact 2.4 *Every positive integer n lies between two consecutive powers of 2. That is, for any $n > 0$, there exists a smallest non-negative integer k such that $n < 2^k$. Hence if $k > 0$ we also have $2^{k-1} \leq n < 2^k$, which implies $k = \lceil \log_2 n \rceil$.*



Fast Powering: Running Time

The number of multiplications in the worst-case is given by

$$
\begin{aligned}
\mathcal{T}(fastpower(x,n)) &= \begin{cases} 0 & \text{if } n = 0 \\ 1 + \mathcal{T}(fastpower(x^2, n \mathbin{\dot{-}} 2)) & \text{if } n > 0 \wedge n \mathbin{\bar{\cdot}} 2 = 0 \\ 2 + \mathcal{T}(fastpower(x^2, n \mathbin{\dot{-}} 2) & \text{otherwise} \end{cases} \\
&\leq \underbrace{2 + 2 + \cdots + 2}_{k \text{ times}} \\
&= 2k \\
&= O(\log_2 n)
\end{aligned}
$$

Since we are interested only in the number of multiplications required for large values of n we may safely assume $n \gg 0$.

Fast Powering: Space Requirement

The maximum amount of space required is therefore given by

$$
\begin{aligned}
& \mathcal{S}(fastpower(x_0, n_0)) \\
= \; & 1 + \mathcal{S}(fastpower(x_1, n_1)) \\
& \vdots \\
= \; & (k-1) + \mathcal{S}(fastpower(x_{k-1}, 1)) \\
= \; & k + \mathcal{S}(fastpower(x_k, 0)) \\
= \; & O(k)
\end{aligned}
$$

and is proportional to the space required to store the k values $x_0, \ldots, x_{k-1}, 1$ and whatever space may be required for the expression $fastpower(x_i, n_i)$ which is a constant amount of space. Hence the space complexity is also $O(k) = O(\log_2 n)$.

Improving Fast Powering

Example 2.5 *It is possible to improve the space complexity of algorithm 13. Notice that in the typical worst-case computation shown for algorithm 13 no simplification of the product is actually performed for the first k steps. This is because of the grouping of the multiplication operation in the form*

$$\leadsto x_0.(x_1.(\cdots(x_{k-2}.(x_{k-1}.1)\cdots))) \tag{14}$$

which defers the evaluation of the "." till all invocations of $fastpower$ are completed. If we could arrange the computation to occur in such a manner that whatever product operation that could be performed actually gets evaluated and only the result is stored we could save on the space requirement. Hence the individual products need to be grouped differently so that the evaluation of the multiplication operation rather than being deferred to the end is performed within each invocation of the function.

Fast Powering 2

Let us call this new function $fastpower2$ which we define as follows.

$$fastpower2(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ fastpower\_tr(x,n,1) & \text{if } n > 0 \end{cases} \qquad (1$$

$$where$$

$$fastpower\_tr(y,m,p) = \begin{cases} p & \text{if } m = 0 \\ \\ fastpower\_tr(y^2, m \,\dot{-}\, 2, p) & \text{if } m > 0 \wedge m \;\bar{\cdot}\; 2 \;(1 \\ \\ fastpower\_tr(y^2, m \,\dot{-}\, 2, p.y) & \text{otherwise} \end{cases}$$

Fast Powering 2: Correctness

But this algorithm looks so different from the original problem of computing the power of a number that its correctness itself would be in serious doubt. Hence before we prove any results about its space complexity we need to prove that it indeed computes powers of numbers.

The proof of correctness of algorithm (15) is complicated by the use of an auxiliary algorithm (16) on which the main function $fastpower2$ depends. Hence a proof of correctness of the expected behaviour of $fastpower\_tr$ is required. The correctness of $fastpower2$ will then depend on this expected behaviour. We state a correctness lemma for $fastpower\_tr$ before we proceed with the proof of correctness of $fastpower2$.

Fast Powering 2: Correctness Lemma

Lemma 2.6 *For all real* $y, p > 0$, *and integer* $m \geq 0$, $fastpower\_tr(y, m, p) = p.y^m$

Fast Powering 2: Proof of Lemma

Proof: We prove this by induction on m.

Basis m = 0. From the definition we get $fastpower\_tr(y, 0, p) = p = p.y^0$.

Induction hypothesis (IH) For all real $y, p > 0$ and integer k, $0 \leq k < m$, $fastpower\_tr(y, k, p) = p.y^k$.

Induction Step Assume $m > 0$, then we have two cases to consider.

Case m is even, say $m = 2j > 0$. Then clearly $0 < j < m$ and

$$fastpower\_tr(y, 2j, p) = fastpower\_tr(y^2, j, p)$$

since $2j \dot{-} 2 = j < m$. By the induction hypothesis we know $fastpower\_tr(y^2, j, p) = p.(y^2)^j = p.y^{2j} = p.y^m$ and the claim is proved.

Case m is odd, say $m = 2j + 1 > 0$. Then again we have $0 < j < m$ and

$$fastpower\_tr(y, 2j + 1, p) = fastpower\_tr(y^2, j, p.y)$$

since $(2j + 1) \dot{-} 2 = j < m$. Again by the induction hypothesis we have $fastpower\_tr(y^2, j, p.y) = p.y.(y^2)^j = p.y^{2j+1} = p.y^m$.

Fast Powering 2: Correctness Final

Now using this lemma it is easy to see that for any real $x > 0$ and integer $n \geq 0$, by the definition of $fastpower2$ the two cases to consider are

$\underline{Case\ n = 0.}$ In this case, by definition $fastpower2(x, 0) = 1 = x^0$.

$\underline{Case\ n > 0.}$ In this case, again by definition and the previous lemma we get $fastpower2(x, n) = fastpower\_tr(x, n, 1) = 1.x^n = x^n$.

Hence the algorithm is indeed correct.

Fast Powering 2: Analysis

A typical computation is as follows

$$\begin{array}{ll} & fastpower2(x_0, n_0) \\ \leadsto & fastpower\_tr(x_0, n_0, p_0) \\ \leadsto & fastpower\_tr(x_1, n_1, p_1) \\ & \vdots \\ \leadsto & fastpower\_tr(x_k, 0, p_k) \\ \leadsto & p_k \end{array}$$

Corollary 2.7 (Invariant) *For all i, $0 \leq i \leq k$, the following* (invariant) *property holds*

$$\boxed{x_0^{n_0}.p_0 = x_i^{n_i}.p_i}$$

Fast Powering 2: Worst Case Running Time

In the worst case we have that each of $n_0, \ldots, n_{k-1} = 1$ is an odd number

and $x_0 > 1$, $n_0 = 2^k - 1$ for some $k > 0$, each $x_{i+1} = x_i^2$, $n_{i+1} = n_i \mathbin{\dot{-}} 2 = 2^{k-(i+1)}1 - 1$ $p_i = \prod_{j=0}^{i-1} x_j$ for each i, $0 \leq i < k$. Notice that $n_k = 0$, $p_0 = 1$ Hence it is easily shown that the number of multiplications and division by 2 that need to be performed is given by

$$
\begin{array}{ll}
 & \mathcal{T}(fastpower2(x_0, n_0)) \\
\leadsto & \mathcal{T}(fastpower\_tr(x_0, n_0, p_0)) \\
\leadsto & 2 + \mathcal{T}(fastpower\_tr(x_1, n_1, p_1)) \\
 & \vdots \\
\leadsto & 2k + \mathcal{T}(fastpower\_tr(x_k, 0, p_k)) \\
\leadsto & O(k)
\end{array}
$$

Fast Powering 2: Worst Case Space Requirement

and the space required is given by

$$\begin{array}{rl} & \mathcal{S}(fastpower2(x_0, n_0)) \\ \leadsto & \mathcal{S}(fastpower\_tr(x_0, n_0, p_0)) \\ \leadsto & \mathcal{S}(fastpower\_tr(x_1, n_1, p_1)) \\ & \vdots \\ \leadsto & \mathcal{S}(fastpower\_tr(x_k, 0, p_k)) \\ \leadsto & O(1) \end{array}$$

Standard ML

- Originated as part of a theorem-proving development project
- Runs on both Windows and UNIX environments
- Is free like many other programming language systems.
- 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.

3.1. Primitives: Integer & Real

Previous: Primitives: Integer & Real

Next: Technical Completeness & Algorithms

Preamble: The Primitive Data types

We have mentioned that to solve a problem in a given computing model we need to evolve *a combination of the primitives of the model in a suitable order*. An algorithm is simply an explicit statement of this combination in an appropriate order. Further we require to express this in a clear, precise and unambiguous fashion (preferably mathematically). It is then necessary to translate this into a program of a formal programming language in order to be able to run it on a machine which has an implementation of the programming language.

We have chosen Standard ML as our vehicle for programming and it is necessary now to get familiar with the rudiments of the language. Most programming languages are fairly large and to formally learn the grammar and grammatical constructions can be tedious and boring. However we do have the advantage that (unlike natural languages), any errors we make in vocabulary (spelling) or grammar would be detected by the *compiler* of the language and pointed out to us. In fact, unless the program gets compiled without errors, it cannot be executed. We will use this feature to understand grammatical constructions as we go along, after learning and understanding just a few of the basic primitives, words and names in the language. This has the advantage that we very quickly start programming in the language without actually mastering all of it.

Most programming language systems however, are also organised as a collection of *modules* or *libraries*. With knowledge of only a few simple grammatical constructions and an understanding of some of the primitives of the modules that we want to use, we may very quickly start constructing and running programs in the language. It is therefore important to understand at least some of the most useful primitives in a module to be able to use them in the construction of programs.

The modules of the SML system are classified in terms of *data types*. Each data type may be regarded as a set of elements along with various operations and relations on the data type and between various distinct data types.

The Integer data type consists of the set of integers along with various operations like addition, multiplication, subtraction and division, as well as various important relations like equality, less than, greater than etc. The concept of a data type therefore may be understood to mean a set of elements along with various operations and relations between elements of the set as well as operations and relations between it and other data types in the system.

The Real data type similarly consists of the set of real numbers along with various operations and relations on the reals. Some of the other data types that we will get familiar with are the data types of booleans, strings, lists etc.

Perhaps the most basic computation need since the dawn of history is that of counting. With counting came various operations on natural numbers such as addition and multiplication which finally led to the theory of integers. So it is natural that we begin our programming journey with the integers.

Algorithms & Programs

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

Factorial

SML: Primitive Integer Operations 1

Our foray into the SML system begins by first opening a text terminal and typing the word “`sml`” followed by the ENTER key on the terminal.

```
sml
Standard ML of New Jersey, ...
-
```

The SML system responds with the line giving the version of SML that the system currently supports and the SML prompt “`-`” which indicates that the system is expecting further input from the user.

SML As Integer Calculator: 1

Initially we deal with simple integer expressions. So right now we use it simply as a calculator for integer expressions. In each case we write the user input in blue followed by the ML system response also in blue.

```
- 5;
val it = 5 : int
-
```

Here ML has simply recognised that the user has typed the integer 5.

Notice that each time the user has to terminate the input by a ; and hit the ENTER key for ML to evaluate the input.

SML As Integer Calculator: 2

We may add two integers, for example.

```
- 3+4;
val it = 7 : int
-
```

SML As Integer Calculator: 3

We may subtract one integer from another.

```
- 5 - 7;
val it = ~2 : int
-
```

Notice that ML uses the symbol “~” instead of the usual minus sign “–”, to denote negative numbers.
But the binary operation of subtraction is denoted using the minus sign.

SML As Integer Calculator: 4

We may give fairly complex expressions.

```
- 2*8-4+6;
val it = 18 : int
-
```

SML As Integer Calculator: 5

We may also give bracketed expressions using the parentheses symbols “(” and “)” for grouping expressions. ML follows the usual operator precedence conventions that we follow in mathematics (e.g. the so-called “BODMAS” rule).

```
- 2*(8-4) + 6;
val it = 14 : int
- ((8-3)* (4+2)) * (9-3);
val it = 180 : int
-
```

However other bracketing symbols such as “[” and “]” and “{” and “}” are reserved for other purposes and are not available for use in arithmetic expressions.

SML As Integer Calculator: 6

Integer division consists of two operations – “div” for the quotient and “mod” for the remainder.

```
- ~(5-9)*3 div 4;
val it = 3 : int
- ((8-3)* (4+2)) mod 7;
val it = 2 : int
-
```

SML As Real Calculator: 1

Calculations with real numbers are similar

```
- 3.28 - 4.32/4.89+3.84;
val it = 6.23656441718 : real
- (3.28 - 4.32)/(4.89+3.84);
val it = ~0.119129438717 : real
```

except that there is only division operator denoted by “/”.

SML As Real Calculator: 2

However integers and reals cannot appear in the same calculation.

```
-  3.28 - 4.32/4.89 + 3;
stdIn:3.2-3.20 Error: operator and operand don't agree [
  operator domain: real * real
  operand:         real * int
  in expression:
    3.28 - 4.32 / 4.89 + 3
```

The integer "3" is the culprit in this case.

SML As Real Calculator: 3

Integers have to be first converted into reals in a real number computation. The previous expression can be calculated as follows:

```
-  3.28 - 4.32/4.89 + 3.0;
val it = 5.39656441718 : real
-
```

The representation “3.0” is the real number representation of the real number 3.

SML As Real Calculator: 4

Alternatively integers can be first converted into reals in a real number computation. The previous expression therefore can be calculated as follows:

```
-  3.28 - 4.32/4.89 + real (3);
val it = 5.39656441718 : real
-
```

The function "real" when applied to the integer "3" converts it into a real number representation for 3.

SML: Primitive Integer Operations 1

Suppose the user now types "`val x = 5`" and presses the ENTER key.

```
sml
Standard ML of New Jersey, ...
- val x = 5;
```

"`val`" is an SML **keyword** (part of SML's vocabulary) indicating that a new name is going to be *defined*. The new name in this case "`x`" is being *defined* (indicated by the "`=`" symbol) to stand for the *value* "`5`". The expression is terminated by the semicolon ("`;`").

Keywords in Programming Languages

Keywords form the main vocabulary of any programming language. For all practical purposes we may regard the (single symbol) operators of the language e.g. `+`, `*`, `~`, `-` etc. also as keywords. Some of the other keywords in SML that we shall come across are `if`, `then`, `else`, `and`, `let`, `in`, `end` etc. Normally a keyword should not be used as the name of any entity defined by the programmer. Sometimes the same keyword may take different meanings depending on the context in which it appears.

Besides keywords, values and programmer and language defined names, a typical programming language also has operators, punctuation and grouping symbols (bracketing). We will encounter these as we go along.

SML: Primitive Integer Operations 1

SML responds as follows:

```
val x = 5 : int
-
```

to indicate that the name `x` has been defined to have the value `5` which is an integer (indicated by the “`: int`”).

Note that it was not necessary to inform SML that the value `5` is an integer. This is the simplest instance of SML’s *type inference system*. In most cases SML will be able to classify the type to which values should belong without being informed by the user.

From this point on, the name `x` may be used in other expressions in place of the integer value 5. Now SML is ready with the prompt “`-`” on a new line for further input from the user.

SML: Primitive Integer Operations 1

Let us define another name “y” and give it a value “6”:

```
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

Now let us find the value of the expression “`x+y`”.

```
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
-
```

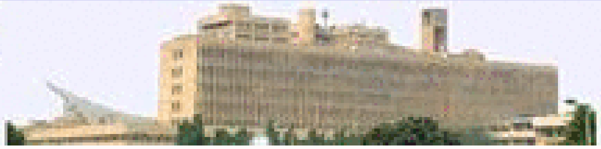
The word “`it`” is a special SML **keyword** which stands for the last value computed by SML.

SML: Primitive Integer Operations 1

Now let us compute other expressions say involving negative integers. The symbol “~” is used to indicate the negative of a number in SML. However the (binary) operation of subtraction is represented by “-”.

```
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
-
```

Notice how the value of `it` keeps changing with each new calculation that is performed. `it` may be thought of as a special container storing the value of the last expression that was evaluated.



SML: Primitive Integer Operations 1

Once `it` has a value, it may be used in other calculations too.

```
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
-
```

In the last two expressions above notice how the value of "`it`" changes from "`~1`" to "`4`" after the addition of the integer "`5`".

SML: Primitive Integer Operations 2

Multiplication of numbers is represented by “*”.

```
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

Let’s define a few more names.

```
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

Integer division is represented by the SML **keyword** “`div`”.

```
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
-
```

Remember: Never divide by zero!

SML: Primitive Integer Operations 2

The SML **keyword** “`mod`” represents the operation of finding the remainder obtained by dividing one integer by another non-zero integer.

```
- 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
-
```

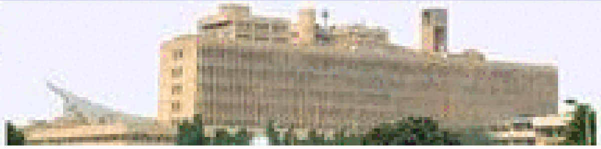
The line containing the phrase “`GC #0.0.0.0.2.45:   (0 ms)`” may be safely ignored by a beginning programmer.

SML: Primitive Integer Operations 3

One may want to check certain properties.

```
- 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
-
```

The line "`a = b*q + r;`" is an expression asking SML to determine whether the given statement is true. Note that the expression does not begin with any SML keyword such as "`val`" and is therefore not a definition.



SML: Primitive Integer Operations 3

Let us look at operations involving negative integers.

```
- 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
-
```

Notice that the result of dividing 21 by -7 seems to yield a quotient of -4. Why do you think that happens?

SML: Primitive Integer Operations 3

What would be the remainder of the same division?

```
- 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
-
```

The remainder is -3. Hmm!

SML: Primitive Integer Operations 3

Does our standard property relating the dividend, divisor, quotient and remainder hold true for negative numbers too?

```
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
-
```

Yes, it seems to!

Quotient & Remainder

For any two integers a and b with $b \neq 0$,the quotient q and remainder r are uniquely determined to satisfy the identity

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

such that

- r is of the same sign as b and
- $0 \leq |r| < |b|$ always holds.

Remarks on Integers and Reals

It may be relevant at this point to note the following:

- Due to various representational issues only a finite subset of the integers is actually representable directly on a machine. These issues mainly relate to the *word length* supported by the hardware for storing in memory locations and registers. In general the set of integers supported on a word-length of w ranges from -2^{w-1} to $2^{w-1}-1$. For more details the reader may refer to standard textbooks on computer architecture.
- However it is possible to write programs which will extend the integers to be of arbitrarily large magnitude. In fact by about the middle of this course you should be able to do it in SML.
- Even though our school mathematics training equips us to regard the set of integers as a subset of the set of reals, it unfortunately does not hold on a digital computer. This is mainly because of various internal representation details which the reader may study from books on hardware and computer architecture. Hence for all practical purposes in programming we may think of the integers and reals as two distinct and independent data types with no apparent relation to each other. However there are functions which *convert* integers to reals and vice-versa.
- As in the case of the integers, the word length constrains the reals too to only a finite subset. Moreover all the reals that are represented in hardware are strictly speaking, only *rational numbers* due to the constraint of a finite representation.
- Further the gaps between the reals represented in hardware are determined by certain *precision* constraints which again depend on word-length.

SML: Primitive Real Operations 1

One can go from integers to reals by explicitly converting an integer value

```
sml
Standard ML of New Jersey, ...
- val a = 25;
val a = 25 : int
- val real_a = real a;
val real_a = 25.0 : real
-
```

Here the word “real” occurring on the right hand side of the definition of the name “real_a” is actually a function from integers to reals which yields the real value representation 25.0 of the integer 25.

SML: Primitive Real Operations 1

```
sml
Standard ML of New Jersey, ...
- val real_a = real a;
val real_a = 25.0 : real
- val b = 7;
val b = 7 : int
- real_a + b;
stdIn:40.1-40.11 Error: operator and operand don't agree
  operator domain: real * real
  operand:         real * int
  in expression:
    real_a + b
-
```

Notice that SML yields an error when an integer is added to a real number.

SML: Primitive Real Operations 1

SML expects the second operand to be real since the the first one is real.

```
stdIn:40.1-40.11 Error: operator and operand don't agree
  operator domain: real * real
  operand:         real * int
  in expression:
    real_a + b
- b + real_a;
stdIn:1.1-2.6 Error: operator and operand don't agree [t
  operator domain: int * int
  operand:         int * real
  in expression:
    b + real_a
-
```

In the second case it expects the second operand to be an integer since the first one is an integer.

Overloading of operators

In the case of an operator such as "+" it is used to denote both integer addition as well as real addition even though the underlying implementations may be different. Such operators are said to be **overloaded**. Some of the other overloaded operators are those for subtraction ("-") and multiplication ("*"). Overloading of common arithmetic operators is deliberate in order to conform to standard mathematical notation. Usually the surrounding context of the operator would make it clear whether an integer or real operation is meant to be used. Where the context is not clear we say that it is *ambiguous*.

We may use these overloaded operators and continue our session as follows:

```
- val e = 2.71828182846;
val e = 2.71828182846 : real
- val pi = 3.14159265359;
val pi = 3.14159265359 : real
- pi+e;
val it = 5.85987448205 : real
- e+pi;
val it = 5.85987448205 : real
- e-pi;
val it = ~0.42331082513 : real
- pi-e;
val it = 0.42331082513 : real
- pi*e;
```

```
val it = 8.53973422268 : real
- e*pi;
val it = 8.53973422268 : real
-
```

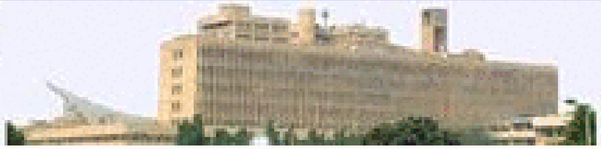
Certain languages like C permit overloaded operators in ambiguous contexts. But then they do have disambiguating rules which clearly specify what the interpretation should be. SML does not however permit the use of such disambiguating rules. For instance in SML one cannot add an integer to a real number, whereas in C this is permitted and rules of whether the addition should be regarded as integer addition or real addition can be quite complex depending upon the context.

SML: Primitive Real Operations 2

Real division is represented by the symbol "`/`".

```
- val a = 25.0;
val a = 25.0 : real
- val b = 7.0;
val b = 7.0 : real
- a/b;
val it = 3.57142857143 : real
- a div b;
stdIn:49.3-49.6 Error: overloaded variable not defined a
  symbol: div
  type: real
GC #0.0.0.0.3.98:   (0 ms)
-
```

Note that the integer division operator "`div`" cannot be used with real numbers.



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
- trunc ~2.36;
val it = ~2 : int
- trunc (~2.36 - 0.5)
val it = ~2 : int
- trunc (~2.55 - 0.5);
val it = ~3 : int
-
```

"`trunc`" is a function which "truncates" a real number to yield only its integer portion. Notice that to round-off a *positive* real number to the nearest integer we may truncate the result of *adding* 0.5 to it. Similarly to round off a *negative* real number we *subtract* 0.5 and truncate the result.

SML: Primitive Real Operations 4

We may define real numbers also using the mantissa-exponent notation.

```
- 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
-
```

Notice the loss of precision in calculating “`d+pi`” and “`d-pi`”.

SML: Precision

The loss of precision gets worse as the difference between the relative magnitudes of the numbers involved increases.

```
- pi + d*10.0;
val it = 300000000003.0 : real
- d*10.0 - pi;
val it = 299999999997.0 : 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
-
```

So much so that there seems to be no difference between the values of “`d*100.0 + pi`” and “`d*100.0 - pi`”

Precision of real arithmetic

Real arithmetic on modern computers is handled by the so-called *floating point processor*. Since this is also fixed precision arithmetic, various *round-off* procedures are adopted in most calculations. So real arithmetic for this reason is always approximate and seldom guaranteed to be accurate.

These practical concerns also affect various basic properties that we expect from real numbers. For one addition and multiplication may not be associative operations i.e. it is not guaranteed that $(a + b) + c = a + (b + c)$. Even the commutativity and distributive properties of operations on real numbers are not guaranteed. Certain implementations may be quite smart. For example, look at this continuation of the previous SML session. As we have already seen there seems to be no difference in the values of "`d*100.0 + pi`" and "`d*100.0 - pi`". However consider the following fragment

```
- (d*100.0 + pi)- (d*100.0 - pi);
val it = 6.283203125 : real
- (d*1000.0 + pi)- (d*1000.0 - pi);
val it = 6.28125 : real
- (d*10000.0 + pi) - (d*10000.0 -pi);
val it = 6.25 : real
-
```

In each of the above cases look carefully at the approximations to 2π that have been obtained. The reason non-zero values have been obtained is simply because the underlying software and hardware have carefully isolated computations of numbers of greatly different magnitudes so as to give reasonable approximate answers. The approximations get less precise as the relative difference in magnitudes increases greatly.

Comparing reals

The nature of representations and floating point algorithms on those representations makes it impossible to check for equality of two real numbers, since they may firstly have different representations, and secondly equality checking requires both of them be converted to the same representation for comparison. But this conversion itself may involve a loss of precision and thus the equality checking procedure itself becomes one of doubtful utility.

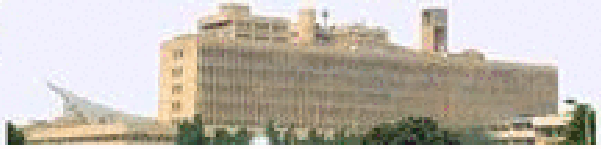
Hence the designers of SML have decided that real numbers may be compared for other relations, such as whether one real is less than another or greater than another *but not for equality*. The following continuation of the last session drives home the point. Study it carefully.

```
- (d*1000.0 + pi)- (d*1000.0 - pi) > (d*10000.0 + pi) - (d*10000.0 -pi);
val it = true : bool
- (d*1000.0 + pi)- (d*1000.0 - pi) >= (d*10000.0 + pi) - (d*10000.0 -pi);
val it = true : bool
-
```

The above statements are true simply because $6.28125 > 6.25$. But the results are clearly counter-intuitive from the stand-point of mathematics.

Now look at the following.

```
- (d*1000.0 + pi)- (d*1000.0 - pi) = (d*10000.0 + pi) - (d*10000.0 -pi);
stdIn:29.1-29.70 Error: operator and operand don't agree [equality type required
```



```
  operator domain: ''Z * ''Z
  operand:         real * real
  in expression:
    d * 1000.0 + pi - (d * 1000.0 - pi) =
      d * 10000.0 + pi - (d * 10000.0 - pi)
-
```

Study the last error message clearly prohibiting equality checking. This happens even if we actually give it identical values as in the following.

```
- 3.0 = 3.0;
stdIn:1.1-14.2 Error: operator and operand don't agree [equality type required]
  operator domain: ''Z * ''Z
  operand:         real * real
  in expression:
    3.0 = 3.0
-
```

3.2. 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`, `false`
- 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
-
```

The conditional in SML

Many functions are defined using a *conditional* construct. That is, depending upon certain conditions being satisfied the function yields one value and otherwise some other value. The simplest example of this from school mathematics perhaps is the absolute value function on the integers or the reals.

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases} \tag{17}$$

or equivalently as

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

Here "$x \geq 0$" and "$x < 0$" are the two conditions. Note that they are expressions which can only be either *true* or *false* for a given value of x.

Such conditional definitions of the form (17) may be written in SML using the conditional expression

`if` *condition* `then` *truevalue* `else` *falsevalue*

The words `if`, `then` and `else` are all SML Keywords. The three words always appear in this order with the *condition* (a boolean expression) appearing between `if` and `then`. *truevalue* and *falsevalue* are referred to as the two *arms* or *clauses* of this conditional and are always separated by `else`. *truevalue* is the `then`-clause and *falsevalue* is the `else`-clause of the conditional.

Definition (17) is written in SML as

```
fun abs x = if x >= 0 then x
            else ~x
```

Conditionals with more than two arms are also possible as in the case of the *signum* function

$$signum(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases} \qquad (18)$$

which may be rendered in SML as

```
fun signum x = if x > 0 then 1
               else if x = 0 then 0
                    else ~1
```

Here it is important to note that the SML version is obtained by *nesting* two conditionals in the form

`if` *condition1* `then` *truevalue1* `else` (`if` *condition2* `then` *truevalue2* `else` *falsevalue*)

i.e. the second conditional appears as the `else`-clause of the first conditional. Notice also how this is made evident in the SML code by using *indentation* which aligns each `else` with the corresponding `if`. When the conditions are complex, we would write the `then`-clause in a separate line with the keyword `then` aligned with the keyword `if`. Such indentation makes code clearer to read though it does not affect compilation or running of the code.

There are two important points to note about conditional definitions.

Totality. For any given values of names used in the conditions, at least one of the conditions should be true. For example, the following is a definition which is *not* total

$$|x| = \begin{cases} x & \text{if } x > 0 \\ -x & \text{if } x < 0 \end{cases}$$

since it does not specify the value of $|0|$.

Mutual Exclusivity. The different conditions are all *mutually exclusive* in the sense that for any given value of the names occurring in the conditions, *exactly* one condition would be true. In the above example for any real or integer x only one of the two conditions can be true – both cannot be true for the same value of x. However in the case of the absolute value function the following could be considered a perfectly correct definition of $|x|$.

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x \leq 0 \end{cases}$$

since both conditions yield the same unique value for each value of x including for $x = 0$ even though the conditions are not mutually exclusive. However all programming languages with this conditional construct implicitly assume exclusivity.

The conditional is one of the most frequently used constructs in most programming languages. Even all inductive definitions depend upon separating the basis of the induction from the induction hypothesis and the conditional is a most useful way of doing this.

A convenient way to think about the conditional is as a ternary operator (made up of three keywords `if`, `then` and `else`) whose operands appear between the keywords of the operator. Actually every construct in every programming language may be thought of as an operator of appropriate *arity*. The conditional is a 3-ary operator (or an operator of *arity* 3) in very much the same way as addition and subtraction are binary operators (i.e. 2-ary operators or operators of *arity* 2).

The Conditional: Example: 1

Given any integer y denoting a year in the Christian calendar, we would like to determine whether it is a *leap year*. A leap year is one which is divisible by 4. Also not all years divisible by 4 are leap years. Century years are leap only if they are divisible by 400. So we define a *boolean* function $leap$ which yields a value "true" if the year y is a leap year and false otherwise.

$$leap\ y = \begin{cases} true & \text{if } 400 \mid y \\ true & \text{if } 100 \nmid y \text{ and } 4 \mid y \\ false & \text{otherwise} \end{cases} \tag{19}$$

where $a|b$ stands for "b is divisible by a" and $a \nmid b$ for "b is *not* divisible by a".

The Conditional: Example: 2

But divisibility checking is not primitive in SML. However, $a|b$ for $a \neq 0$ if and only if $b\ mod\ a = 0$ where mod denotes the remainder obtained on division.
Definition (19) may be rewritten using mod.

$$leap\ y = \begin{cases} true & \text{if } y\ mod\ 400 = 0 \\ true & \text{if } y\ mod\ 100 \neq 0 \text{ and } y\ mod\ 4 = 0 \\ false & \text{otherwise} \end{cases} \tag{20}$$

Notice how the conditions in this definition are *mutually exclusive* and *total*.

The Conditional: Example: 3

In SML the definition may be written as

```
fun leap y = if y mod 400 = 0 then true
             else if y mod 100 = 0
                  then false
                  else if y mod 4 = 0
                       then true
                       else false
```

Notice how the code has been indented to make clear the nesting structure and and how each *else*-clause has been aligned with its matching `if`.

The Conditional: Identities

Notice that the function *leap* yields a boolean value on an integer argument. Also notice how the conditions are nested within one another.

Conditions can get pretty complicated and nesting levels would often reflect this complexity and therefore affect the readability of the code. There are many ways to tackle these problems.

Indentation Use a good indentation-style by aligning the keywords of a compound operator so that the code becomes readable. Remember that a programming language notation is a highly formalised mathematical notation. Though in general, it is to be compiled and run by a machine, it should also be readable and comprehensible so that one may easily identify errors and correct them. However good indentation alone does not solve the problem.

Simplifications using algebraic identities There are many algebraic identities that hold. The use of these identities can greatly simplify the code and shorten it and make it easier to read and understand and therefore ensure its correctness. In the case of the above example since both the conditions and the results are boolean one may effectively use some obvious identities to eliminate some occurrences of the constants `true` and `false`. We use $\equiv$ as an equality relation between different SML expressions. We interpret it as an equality relation between SML expressions. So it is possible to substitute one side of the identity by the other in any SML expression without changing the meaning of the expression.

The principle of substitutivity (viz. that equals may be substituted for equals in all contexts) is an important principle in all of mathematics. Throughout your mathematics education you have probably used it unconsciously. This is an important consequence of using the pure functional programming model.

Using Identities

We use $\equiv$ as an equality relation between different SML expressions. The simplest identity is the following:

$$\boxed{\begin{array}{lcl} & & \texttt{if}\ \mathit{condition} \\ \mathit{condition} & \equiv & \texttt{then true} \\ & & \texttt{else false} \end{array}}$$

This identity (used from right to left) may be used to simplify the code of `leap`.

```
fun leap y = if y mod 400 = 0 then true
             else if y mod 100 = 0
                  then false
                  else y mod 4 = 0
```

Some More Identities

Some simple and obvious identities are

$$\boxed{\begin{array}{lcl} & & \texttt{if true} \\ \mathit{truevalue} & \equiv & \texttt{then}\ \mathit{truevalue} \\ & & \texttt{else}\ \mathit{falsevalue} \end{array}}$$

$$\boxed{\begin{array}{lcl} & & \texttt{if false} \\ \mathit{falsevalue} & \equiv & \texttt{then}\ \mathit{truevalue} \\ & & \texttt{else}\ \mathit{falsevalue} \end{array}}$$

Other useful Identities

Another useful identity is the following.

$$\boxed{\begin{array}{lcl} & & \texttt{if}\ condition \\ \texttt{not}\ \ (condition) & \equiv & \texttt{then}\ \texttt{false} \\ & & \texttt{else}\ \texttt{true} \end{array}}$$

A fairly general identity.

$$\boxed{\begin{array}{lcl} \texttt{if}\ condition & & \texttt{if}\ \texttt{not}\ \ (condition) \\ \texttt{then}\ truevalue & \equiv & \texttt{then}\ falsevalue \\ \texttt{else}\ falsevalue & & \texttt{else}\ truevalue \end{array}}$$

where *truevalue* and *falsevalue* may themselves be conditional expressions.

Structuring As conditions become even more complex and harder to understand, simplifications and appropriate indentation may still leave definitions complicated and hard to understand. In such situations a structuring mechanism which clearly isolates *logically distinct concepts* is then necessary. For example, even in the case of the leap year example we could divide the cases of the definition into those involving century years and others. So we isolate the century years from other years by defining auxiliary functions which capture the intuitive concept we are trying to program. Here is a definition which uses an auxiliary definition and also removes some redundant uses of the constants $true$ and $false$.

Structural Simplifications

$$
\begin{aligned}
\mathit{century}\ x &= (x\ \mathit{mod}\ 100 = 0)\\
\mathit{leap}\ y &= \begin{cases} (y\ \mathit{mod}\ 400 = 0) & \text{if } \mathit{century}(y)\\ (y\ \mathit{mod}\ 4 = 0) & \text{otherwise}\end{cases}
\end{aligned}
$$

In SML the above definitions become

```
fun century x = (x mod 100 = 0);
fun leap    y = if (century y)
                then (y mod 400 = 0)
                else y mod 4 = 0
```

Another example of a conditional is the number of days in any month of the year. Try defining this as a function of month number (1 to 12).

Binary Boolean Operators in SML

The two SML operators corresponding to “and” and “or” are the keywords `andalso` and `orelse`.

Examples:

```
- val n = 10;
val n = 10 : int
- (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;
val it = false : bool
-
```

$\wedge$ vs. andalso

The meanings of the two boolean operators may be defined using truth tables as we usually do in boolean algebra. For any two boolean conditions p and q we have the following truth table for the boolean operators $\wedge$ and the SML operator 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

Similarly the meanings of $\vee$ and the SML operator `orelse` may be defined by the following truth table.

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

Boolean operators and The Conditional

The boolean operators may also be expressed in terms of the conditional and the truth values as the following simple identities show.

$$p \ \texttt{andalso} \ q \equiv \texttt{if} \ p \ \texttt{then} \ q \ \texttt{else} \ \ \texttt{false} \tag{21}$$

$$p \ \texttt{orelse} \ q \equiv \texttt{if} \ p \ \texttt{then} \ \ \texttt{true} \ \ \texttt{else} \ q \tag{22}$$

3.5. Nontermination and Boolean Operators

The operators `andalso` and `orelse` differ from their Boolean algebra counterparts $\wedge$ and $\vee$ in one important aspect. This is best explained in the presence of nontermination. An example of a non-terminating function definition was that of the alternative definition of factorial given in definition (6) We have already seen examples of $leap$ and $century$. We have also seen that such functions may be used in conditionals. It is quite possible to define boolean functions in SML which may not terminate on certain inputs.

One example is the following inductively defined function on the integers

$$gtz(n) = \begin{cases} true & \text{if } n = 1 \\ gtz(n-1) & \text{otherwise} \end{cases}$$

Notice that this function will never terminate for non-positive integer values of n. Now consider forming a complex boolean condition using such a function.

SML: orelse

```
Standard ML of New Jersey,
- val tt = true;
val tt = true : 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
- (gtz 0) orelse tt;
```

The result of the last expression is never obtained!

Computation: 1

The computation of `tt orelse (gtz 0)` goes as follows — essentially a left to right evaluation of the expression unless bracketing associates the sub-expressions differently.

```
  tt orelse (gtz 0)
≡ true orelse (gtz 0)
≡ if true then true orelse (gtz 0)
≡ true
```

Computation: 2

The computation of `(gtz 0) orelse tt` on the other hand goes as follows (again left to right simplification and evaluation unless dictated otherwise by the bracketing):

```
   (gtz 0) orelse tt
≡ (if 0=1 then true else gtz (0-1)) orelse tt
≡ (gtz ~1) orelse tt
≡ ((if ~1=1 then true else gtz (~1-1)) orelse tt
≡ (gtz ~2) orelse tt
≡ ((if ~2=1 then true else gtz (~2-1)) orelse tt
≡ (gtz ~3) orelse tt
≡ ...
```

and so on forever.

The Effect of Nontermination: 1

Since the evaluation of `(gtz 0)` does not terminate, no result will be obtained for the expression `(gtz 0) orelse tt`, whereas `tt orelse (gtz 0)` will always yield the value `true` since by the identities, the condition `(gtz 0)` is never evaluated and

```
  tt orelse (gtz 0)
≡ if tt then true else (gtz 0)
≡ true
```

Hence

$$\boxed{\texttt{tt orelse (gtz 0)} \not\equiv \texttt{(gtz 0) orelse tt}}$$

SML: andalso

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

Similarly we have

$$\boxed{\texttt{ff andalso (gtz 0)} \neq \texttt{(gtz 0) andalso ff}}$$

$\wedge$, `andalso` with $\perp$

Let us denote nontermination by $\perp$. We then have a modified truth table as follows.

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

$\wedge$ is commutative whereas `andalso` is not (from the last two lines).

or, `orelse`, $\perp$

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

$\vee$ is commutative whereas `orelse` is not (as is clear from the last two lines of the table).

Conclusions

- The boolean operators `andalso` and `orelse` differ from their corresponding analogues in mathematics.
- Under all circumstances however, `andalso` and `orelse` satisfy the identities relating them to the conditional
- The boolean operators $\wedge$ and $\vee$ are commutative whereas `andalso` and `orelse` are not.
- The boolean operators $\wedge$ and $\vee$ are also associative. Are `andalso` and `orelse` associative? Prove using the conditional identitites or find a counterexample to show that they are not.
- How does negation interact with nontermination? Write a new truth table for `not`. Does it differ in behaviour from the usual boolean $\neg$?

Exercises

1. Write down the full truth table for the `not` operator.
2. Prove that both `andalso` and `orelse` are both associative. That is
   (a) $p\ \texttt{andalso}\ (q\ \texttt{andalso}\ r) \equiv (p\ \texttt{andalso}\ q)\ \texttt{andalso}\ r$
   (b) $p\ \texttt{orelse}\ (q\ \texttt{orelse}\ r) \equiv (p\ \texttt{orelse}\ q)\ \texttt{orelse}\ r$

   Hint: Do a case analysis on the three possible values of p.
3. Prove that `andalso` and `orelse` also satisfy the DeMorgan laws.
   (a) $\texttt{not}\ (p\ \texttt{andalso}\ q) \equiv (\texttt{not}\ p)\ \texttt{orelse}\ (\texttt{not}\ q)$
   (b) $\texttt{not}\ (p\ \texttt{orelse}\ q) \equiv (\texttt{not}\ p)\ \texttt{andalso}\ (\texttt{not}\ q)$

3.6. Example: Fibonacci

A Detailed Example

As a precursor of the various aspects involved in computing, we choose a rather simple example and study it thoroughly for the various properties. We start with a simple mathematical definition and prooceed to subject it to rigorous analysis and transformations. It is our aim to introduce the tools of the trade of computing by means of this example. The tools include the following.

1. A more or less direct translation of the definition into a functional program. There is a certain sense in which it would be quite obvious that this program exactly captures the definition of the function.
2. Alternative definitions which in a certain sense generalise the first one, leading to other functional programs which implement the same function.
3. Proofs of correctness which show that the new definitions indeed entirely capture all the properties of the first definition.
4. An analysis of the two definitions for what they mean in terms of computational complexity. Most of this requires no knowledge of digital computers and should be quite intuitive.

As our example we have chosen the function which generates the fibonacci numbers. The fibonacci number sequence is the sequence given by

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots \tag{23}$$

where except for the starting two numbers, every number in the sequence may be obtained as the sum of the two preceding numbers in the sequence. Sequences expressed as functions of the position (starting from a position 0) are fairly common and the fibonacci sequence is no exception. It may be generated by the following function

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases} \tag{24}$$

The Fibonacci numbers have many interesting mathematical properties, and these properties could also be used to define algorithms for generating them. However, at this stage we use only the definition to study the relationships between mathematical definitions, algorithms obtained directly from the definitions, their correctness and their complexity.

Since we use several possible definitions for the generation of fibonacci numbers we subscript each new definition as we go along so that we can keep track of the various definitions we give. So we begin with

Fibonacci Numbers: 1

$$F_1(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_1(n-1) + F_1(n-2) & \text{if } n > 1 \end{cases}$$

A slightly shorter but equivalent definition is as follows:

$$F_2(n) = \begin{cases} n & \text{if } 0 \leq n \leq 1 \\ F_2(n-1) + F_2(n-2) & \text{if } n > 1 \end{cases}$$

It is easy to see that these definitions generate the fibonacci sequence (23).

The function F_1 (and F_2) is a typical example of a function that is inductively defined. Our previous encounter with an inductively defined function was the factorial function.

Fibonacci Numbers: 2

$$F_1(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_1(n-1) + F_1(n-2) & \text{if } n > 1 \end{cases}$$

This function may be directly translated into SML as follows:

```
fun fib1 (n) =
    if (n = 0) then 0
    else if (n = 1) then 1
    else fib1 (n-1) + fib1 (n-2);
```

Fibonacci Numbers: 3

$$F_2(n) = \begin{cases} n & \text{if } 0 \leq n \leq 1 \\ F_2(n-1) + F_2(n-2) & \text{if } n > 1 \end{cases}$$

This definition may be translated into SML as either

```
fun fib2 (n) =
    if (0 <= n) andalso (n <= 1)then n
    else fib2 (n-1) + fib2 (n-2);
```

or as

```
fun fib2' (n) =
    if (n = 0) orelse (n = 1) then n
    else fib2' (n-1) + fib2' (n-2);
```

Fibonacci Numbers: 4

One could give other alternative definitions as well. For example, consider the following function F_3 defined in terms of another function F_a. Let

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

and then define

$$F_3(n) = F_a(n, 0, 1)$$

Fibonacci Numbers: 5

The two functions F_a and F_3 may then be translated into SML as

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

fun fib3 (n) = fib_a (n, 0, 1);
```

respectively.

Is it Correct?

In the first two cases it was more or less “obvious” that the two definitions were indeed equivalent, i.e. that they defined the same function.
But as we find more and more complex ways of defining F, the question arises as to whether the new definitions are equivalent to the old ones.

Intuition. F_a looks like a generalization of F.

Question 1. What does it actually generalize to?

Question 2. Does the generalization have a “closed” form i.e. can F_a be expressed in some way other than by using F_a itself?

Question 3. Can one prove that F_3 is exactly the same function as F_1 (and F_2)?

Question 4. *How does one go about proving such a thing?*

A Direct Approach

Our intuition suggests that since

$$\begin{aligned} F_a(0,0,1) &= 0\\ F_a(1,0,1) &= 1\\ F_a(2,0,1) &= 1\\ F_a(3,0,1) &= 2 \qquad\qquad \text{and}\\ F_3(n) &= F_a(n,0,1) \end{aligned}$$

F may be obtained from F_a by simply setting $a = 0$ and $b = 1$. So we might try to prove a direct theorem relating F with this special case of F_a.

Theorem 3.1 *For all integers* $n > 1$,

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

Since the definitions are all inductive it is natural to first try a proof by mathematical induction. So let's have a go at it.

Try Proving it!

Proof: By induction on $n > 1$.

Basis For $n = 0$, $F_a(0, 0, 1) = 0 = F(0)$ and for $n = 1$, $F_a(1, 0, 1) = 1 = F(1)$

Induction hypothesis (IH) Assume $F_a(k, 0, 1) = F(k)$, for some $k \geq 1$
Induction Step

$$\begin{array}{rcl} & & F_a(k+1, 0, 1) \\ \{\text{By definition of } F_a\} & = & F_a(k, 1, 1) \\ \{\text{Induction Hypothesis}\} & = & ???\end{array}$$

STUCK!

So that does not work and we have to think of something else.

Trial & Error

Could one use trial and error to gather more intuition and understanding about the relationship between F_a and F?

$$\begin{aligned}
F_a(0,a,b) &= a & & \\
F_a(1,a,b) &= b & & \\
F_a(2,a,b) &= a+b & &= aF(1)+bF(2) \\
F_a(3,a,b) &= F_a(2,b,a+b) & & \\
&= a+2b & &= aF(2)+bF(3) \\
F_a(4,a,b) &= F_a(3,b,a+b) & & \\
&= F_a(2,a+b,a+2b) & & \\
&= 2a+3b & &= aF(3)+bF(4) \\
F_a(5,a,b) &= F_a(2,a+2b,2a+3b) & & \\
&= 3a+5b & &= aF(4)+bF(5)
\end{aligned}$$

The patterns suggest the following.

Generalization

- $F_a(0, a, b) = a$
- $F_a(1, a, b) = b$
- $F_a(n, a, b) = aF(n-1) + bF(n)$ for $n > 1$.

Theorem 3.2 *For all integers* a, b *and* $n > 1$,

$$\boxed{F_a(n, a, b) = aF(n-1) + bF(n)}$$

If this theorem holds then a corollary of this theorem is the direct one (viz. $F_a(n, 0, 1) = F(n)$) that we tried and failed to prove.
Now let's try to prove this theorem.

Proof by Induction on $n > 1$

Proof:

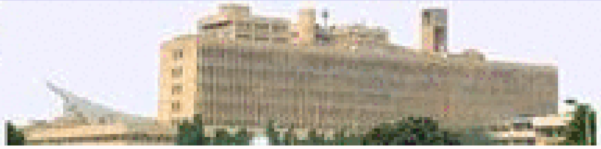
Basis For $n = 2$, $F_a(2, a, b) = a + b = aF(1) + bF(2)$

Induction hypothesis (IH) Assume $F_a(k, a, b) = aF(k-1)+bF(k)$, for some $k > 1$ and all integers a, b.

Induction Step

$$\begin{array}{lll} & & F_a(k+1, a, b) \\ \{\text{ Definition of } F_a\} & = & F_a(k, b, a+b) \\ \{\text{ Induction Hypothesis}\} & = & bF(k-1) + (a+b)F(k) \\ & = & aF(k) + b(F(k-1) + F(k)) \\ \{\text{ Definition of } F\} & = & aF(k) + bF(k+1) \end{array}$$

Aha! So that certainly worked!



Another Generalization

Intuition. If a and b are successive fibonacci numbers (say for some $j \geq 1$, $a = F(j-1)$ and $b = F(j)$) then $F_a(1, a, b) = b$ and $F_a(2, a, b) = F_a(1, F(j), F(j-1)+F(j)) = F(j+1)$ and then $F_a(n, a, b)$ is n steps ahead of a in the fibonacci sequence (23).

Here is a more direct theorem

Theorem 3.3 *For all integers* $n \geq 1$ *and* $j \geq 1$,

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

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

Again a corollary of the above theorem is the direct theorem which we failed to prove earlier.

Corollary 3.4 *For all integers* $n \geq 1$,

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

Try Proving it!

Proof:

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

Induction hypothesis (IH) For some $k > 1$ and all $j \geq 1$,

$$F_a(k, F(j-1), F(j)) = F(k+j-1)$$

Induction Step We need to prove $F_a(k+1, F(j-1), F(j)) = F(k+j)$.

$$\begin{aligned}
& & & F_a(k+1, F(j-1), F(j)) \\
& & = \; & F_a(k, F(j), F(j-1) + F(j)) \\
& & = \; & F_a(k, F(j), F(j+1)) \\
& \{\text{ Induction Hypothesis}\} & = \; & F_a(k+j)
\end{aligned}$$

Well that also worked satisfactorily!

Lessons learned

We have shown the first non-obvious proof of correctness of an algorithm viz. that F_3 does generate the fibonacci sequence (23).

Our attempts to prove the correctness show that

1. Certain obvious and direct approaches may not always work,
2. An algorithm may have subtle generalizations (or even properties) which may need to be established before we attempt to prove correctness.
3. Coming up with such a generalised theorem is crucial to justify the correctness of the algorithm. Getting the right generalizations requires hard work and intuition about the algorithm.

In certain cases designing a new algorithm often requires studying the properties and proving fresh theorems based on properties. We will encounter this in several examples. In each case the only way to convince ourselves about the correctness of our algorithm is to prove it as we do in mathematics.

The process of coming up with alternative algorithms and also proving their correctness requires a degree of creativity higher than that in the standard school mathematics curriculum in India. Most exercises in school mathematics which involve proofs are such that the student is merely asked to prove a given statement. The student is seldom called upon to come up with a new (non-obvious) statement and also prove it right. Herein lies the creative element in correct algorithm design.

Time Complexity

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

Space Complexity

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

Time Complexity: $\mathcal{R}$

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

Time Complexity

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

So we concentrate on $\mathcal{R}$.

Time Complexity: $\mathcal{R}$

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

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

Theorem 3.5 $\mathcal{R}_F(n) \leq 2^{n-1}$ *for all* $n > 2$

Proof: By induction on $n > 2$.

Bound on $\mathcal{R}$

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

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

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

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

Other Bounds: $\mathcal{C}_F$

One comparison for each call.

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

Theorem 3.6 $\mathcal{C}_F(n) \leq 2^n$ *for all* $n \geq 0$.

Other Bounds: $\mathcal{A}_F$

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

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

Theorem 3.7 $\mathcal{A}_F(n) \leq 2^{n-1}$ *for all* $n > 0$.

3.7. Fibonacci Numbers and the Golden Ratio

The Fibonacci numbers are related to the golden ratio $\phi = (1+\sqrt{5})/2 = 1.61803\ldots$ which is one of the solutions of the quadratic equation

$$x^2 = 1 + x \tag{25}$$

The other solution $\psi = (1-\sqrt{5})/2 = -0.61803\ldots$ is the conjugate of ϕ. Since ϕ and ψ are both solutions of equation (25) it follows that

$$\phi^2 = \phi + 1 \tag{26}$$

$$\psi^2 = \psi + 1 \tag{27}$$

It is easy to prove by induction on n that

$$F_n = \frac{\phi^n - \psi^n}{\sqrt{5}} \tag{28}$$

Proof: It is clear that $F_0 = 0$. We then proceed by induction on n.

Basis.($n = 1$) Trivial.

Induction hypothesis. For all $k \geq 1$, $k < n$, $F_k = \dfrac{\phi^k - \psi^k}{\sqrt{5}}$.

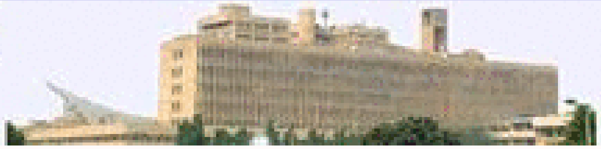
Induction Step. Assume $n \geq 2$. We have

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ &= \frac{\phi^{n-1} - \psi^{n-1}}{\sqrt{5}} + \frac{\phi^{n-2} - \psi^{n-2}}{\sqrt{5}} && \textit{By induction hypothesis} \\ &= \frac{\phi^{n-2}(\phi + 1) - \psi^{n-2}(\psi + 1)}{\sqrt{5}} \\ &= \frac{\phi^n - \psi^n}{\sqrt{5}} && \textit{By the identities (26) and (27)} \end{aligned}$$

In fact notice that since $|\psi| < 1$ and $\sqrt{5} > 2$ we have that for all $n > 1$, $|\psi^n|/\sqrt{5} < 1/\sqrt{5} < 1/2$. Hence $F_n = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$.

4. Algorithms: Design & Refinement

4.1. 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

The following is a faster powering method. See if you can figure it out!

```
fun power2 (x, n) =
   if n < 0
  then 1.0/power2 (x, ~n)
  else if n = 0
  then 1.0
  else
     let fun even m = (m mod 2 = 0);
         fun square y = y * y;
         val pwr_n_by_2 = power2 (x, n div 2);
         val sq_pwr_n_by_2 = square (pwr_n_by_2)
      in if even (n)
         then sq_pwr_n_by_2
         else x * sq_pwr_n_by_2
     end
```

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:89.32-89.46
-
```

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

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

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

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

4.2. 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 Completeness
    - 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

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

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

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

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`

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

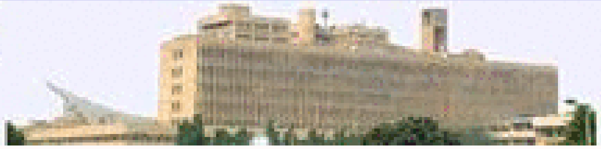
```
fun ifdivisor (k) =
    if n mod k = 0
    then k
    else 0
```

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

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

Hence

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

Perfect Numbers . . . *contd.*

Hence

$$
\begin{aligned}
& perfect(n) \\
\iff & n = \textstyle\sum_{k=1}^{n-1} ifdivisor(k) \\
\iff & n = \textstyle\sum_{k-1}^{n \texttt{ div } 2} ifdivisor(k)
\end{aligned}
$$

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!

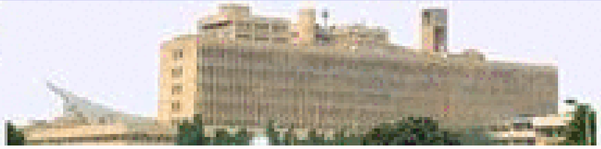
Summation: Generalizations

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

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

Algorithmic Improvements:

1. $perfect2$
2. $shrink2$



Algorithmic Variations

1. For any $k|n$, $m = n$ div k is also a divisor of n
2. 1 is a divisor of every positive number
3. For $n > 4$, $\lfloor\sqrt{n}\rfloor < n \;\texttt{div}\; 2$
4. It is easy to see by inspection that the first perfect number has to be greater than 4.
5. Hence $\sum_{k=1}^{n \,\texttt{div}\, 2} ifdivisor(k) =$

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

Algorithmic Variations

$$perfect(n)$$

$$\iff (n > 4) \wedge (n = 1 + \sum_{k=2}^{\lfloor\sqrt{n}\rfloor} ifdivisor2(k))$$

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?

Glitch

Question. What if for some value of k, $k = n \texttt{ div } k$?

Question. Can a perfect number be a perfect square?

skip

4.3. Variations: Algorithms & Code

Recap

- Combinations
- Perfect Numbers
- Code Variations
- Algorithmic Variations

forward

Recap: Combinations

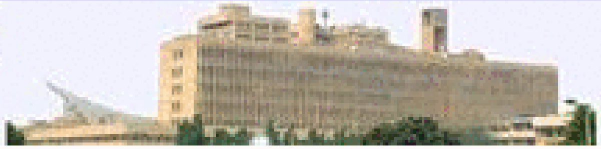
$$
\begin{aligned}
{}^{n}C_{k} &= \frac{n!}{(n-k)!k!} \\
&= \frac{n(n-1)\cdots(n-k+1)}{k!} \\
&= \frac{n(n-1)\cdots(k+1)}{(n-k)!} \\
&= {}^{n-1}C_{k-1} + {}^{n-1}C_{k}
\end{aligned}
$$

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

$$perfect(n)$$

$$\iff (n > 4) \wedge (n = 1 + \sum_{k=2}^{\lfloor\sqrt{n}\rfloor} ifdivisor2(k))$$

$$\iff (n > 4) \wedge (2n = \sum_{k=1}^{\lfloor\sqrt{n}\rfloor} ifdivisor2(k))$$

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 $m = (n \texttt{ div } k)$

```
local
  exception invalidArg;

  fun ifdivisor2 (n, k) =
    if n <= 0 orelse
       k <= 0 orelse
       n < k
    then raise invalidArg
    else if n mod k = 0
    then if k = n div k
         then k
         else k + (n div k)
    else 0;

  fun sum_div2 (n, l, u) =
    if n <= 0 orelse
       l <= 0 orelse
       l > n  orelse
       u <= 0 orelse
       u > n
    then raise invalidArg
    else if l > u
    then 0
    else ifdivisor2 (n, l) + sum_div2 (n, l+1, u)

in
  fun perfect n =
    if n <= 0
    then raise invalidArg
    else if n <= 4 then false
    else let val sqrt_n = Real.trunc(Real.Math.sqrt(real n))
         in 2*n = sum_div2 (n, 1, sqrt_n)
         end
end


exception LessThan5
fun genAllPerfectUpto (m) =
    if  m < 5 then raise LessThan5
    else
        let fun genFromTo (l, u, P) =
                if l > u then P
```

```
                 else if (perfect l)
                 then (print ((Int.toString l)^"\n");
                       genFromTo (l+1, u, l::P)
                      )
                 else genFromTo (l+1, u, P)
        in  genFromTo (5, m, [])
        end

(* 6, 28, 496, 8128,  33550336, 8589869056, 137438691328
   are the first seven perfect numbers according to Wikipedia
*)
```

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 differently

A Faster Power

$$x^n = \underbrace{(x \times x \times \cdots \times x)}_{n/2 \text{ times}} \times \underbrace{(x \times x \times \cdots \times x)}_{n/2 \text{ times}}$$

when n is even and

$$x^n = \underbrace{(x \times x \times \cdots \times x)}_{\lfloor n/2 \rfloor \text{ times}} \times \underbrace{(x \times x \times \cdots \times x)}_{\lfloor n/2 \rfloor \text{ times}} \times x$$

when n is odd

Power2: Complete

$power2(x, n) =$

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

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`

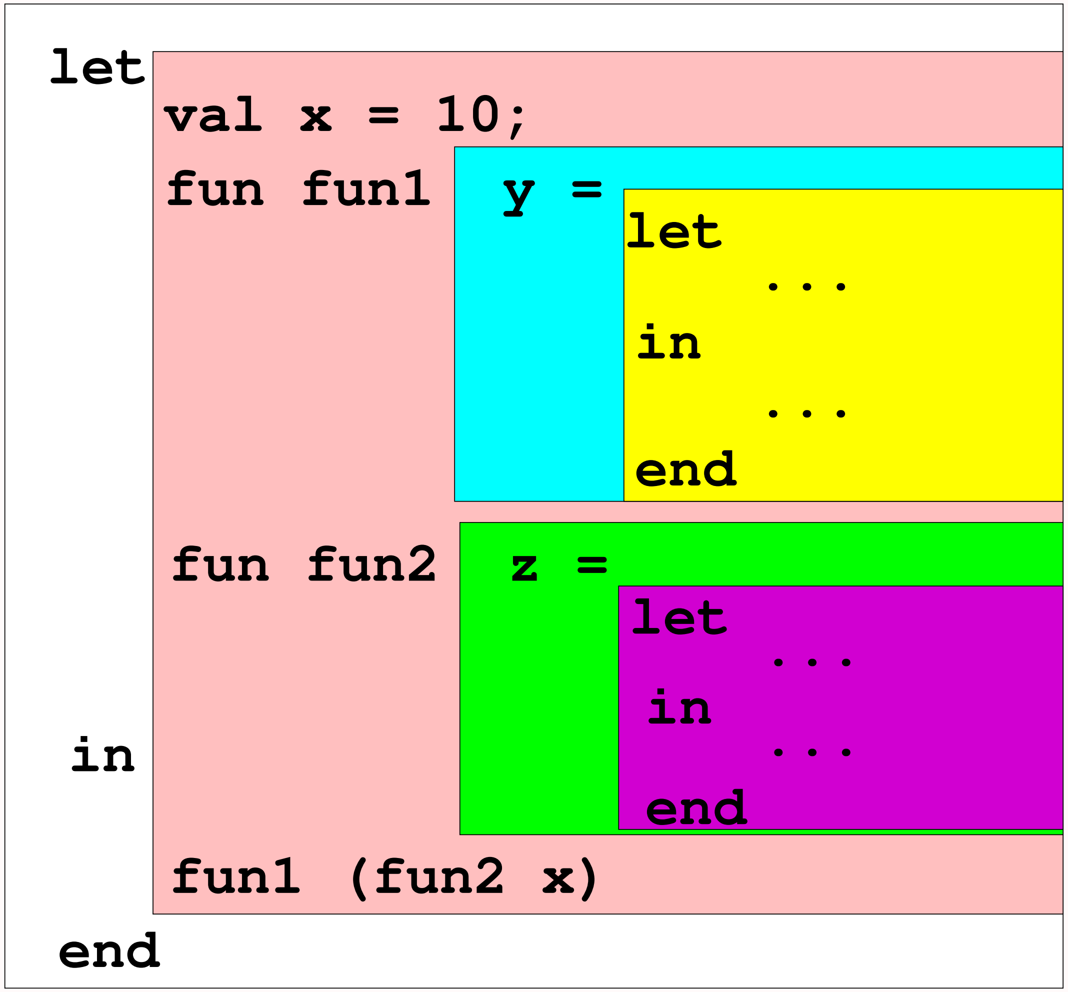
Scope Rules

- 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

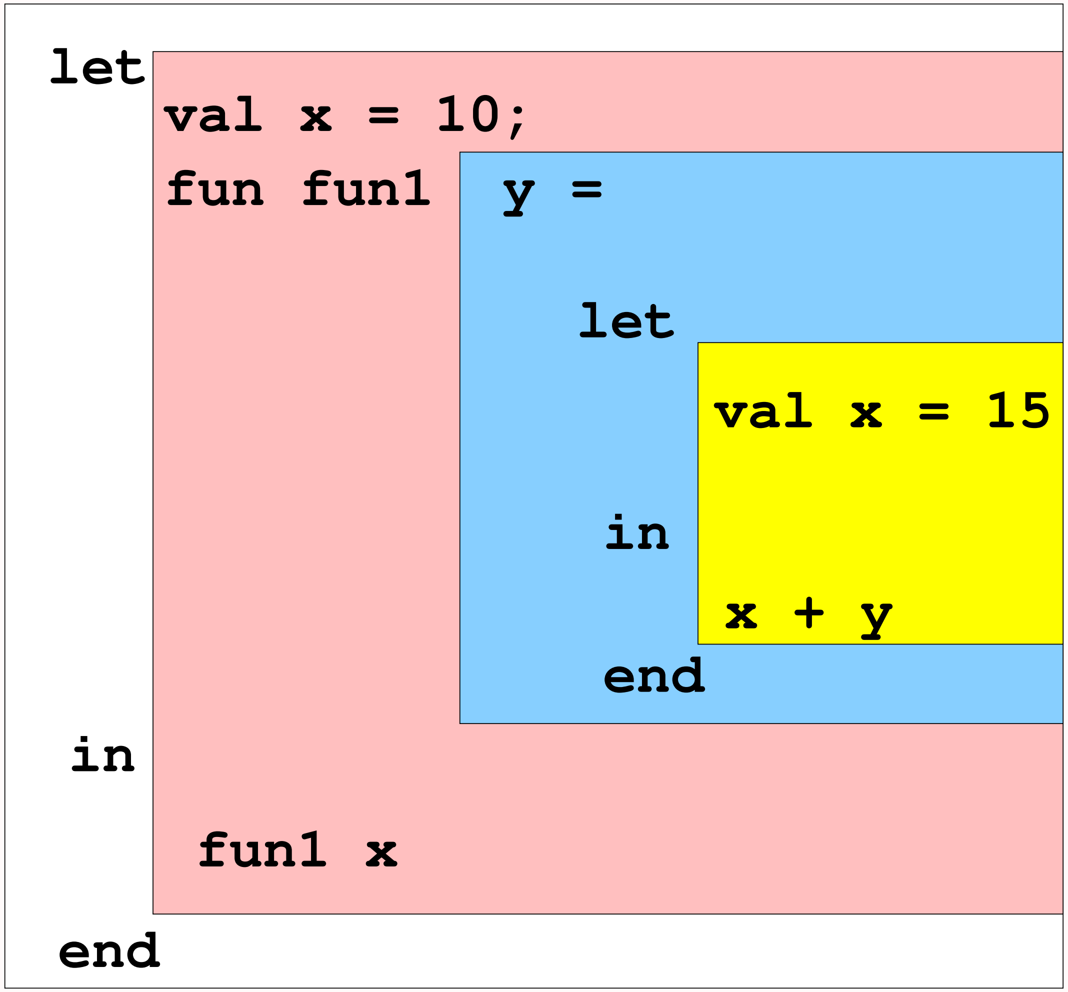
4.4. Names, Scopes & Recursion

Disjoint Scopes



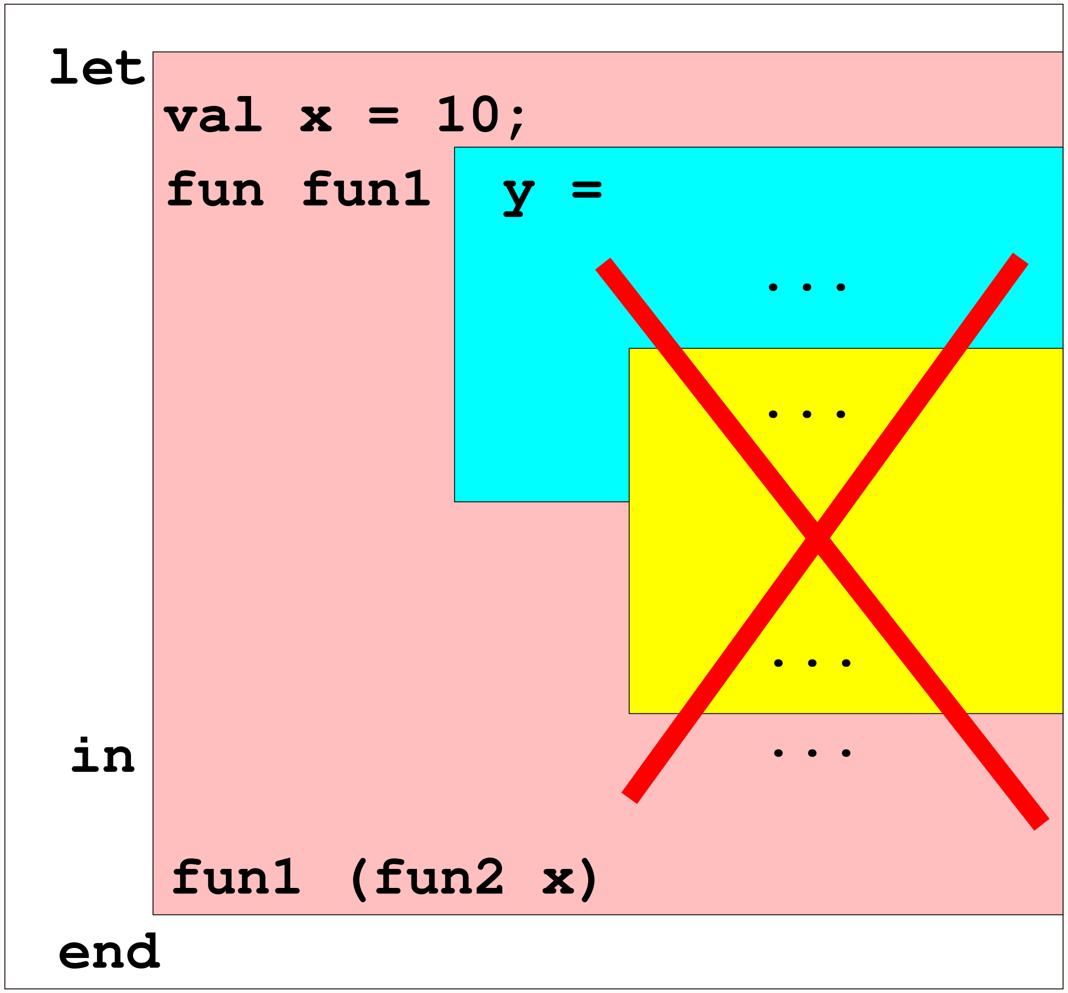
```
let
   val x = 10;
   fun fun1  y =
                 let
                       ...
                  in
                       ...
                  end
   fun fun2  z =
                 let
                       ...
                  in
                       ...
                  end
 in
   fun1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

Nested Scopes



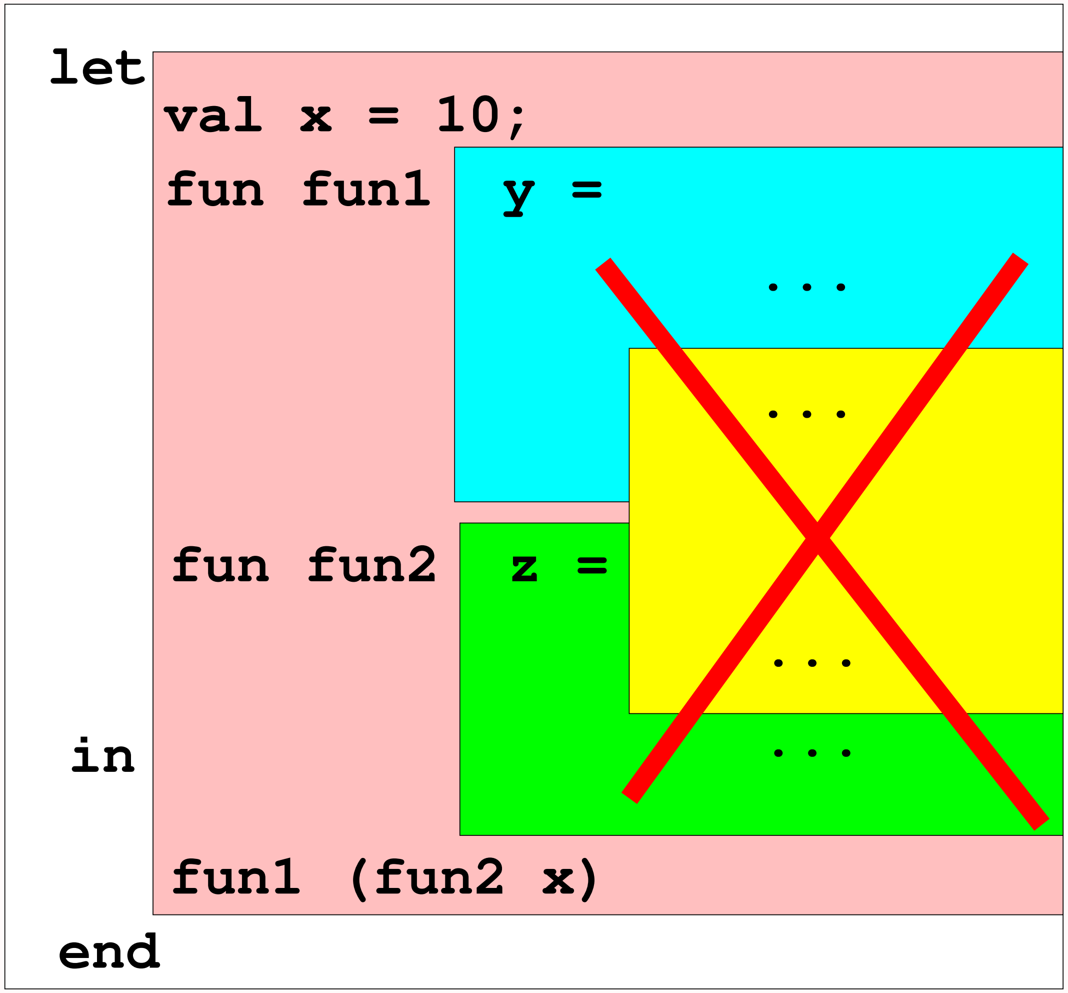
```
let
    val x = 10;
    fun fun1 y =
        let
            val x = 15
        in
            x + y
        end
in
    fun1 x
end
```

Overlapping Scopes



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

Spannning



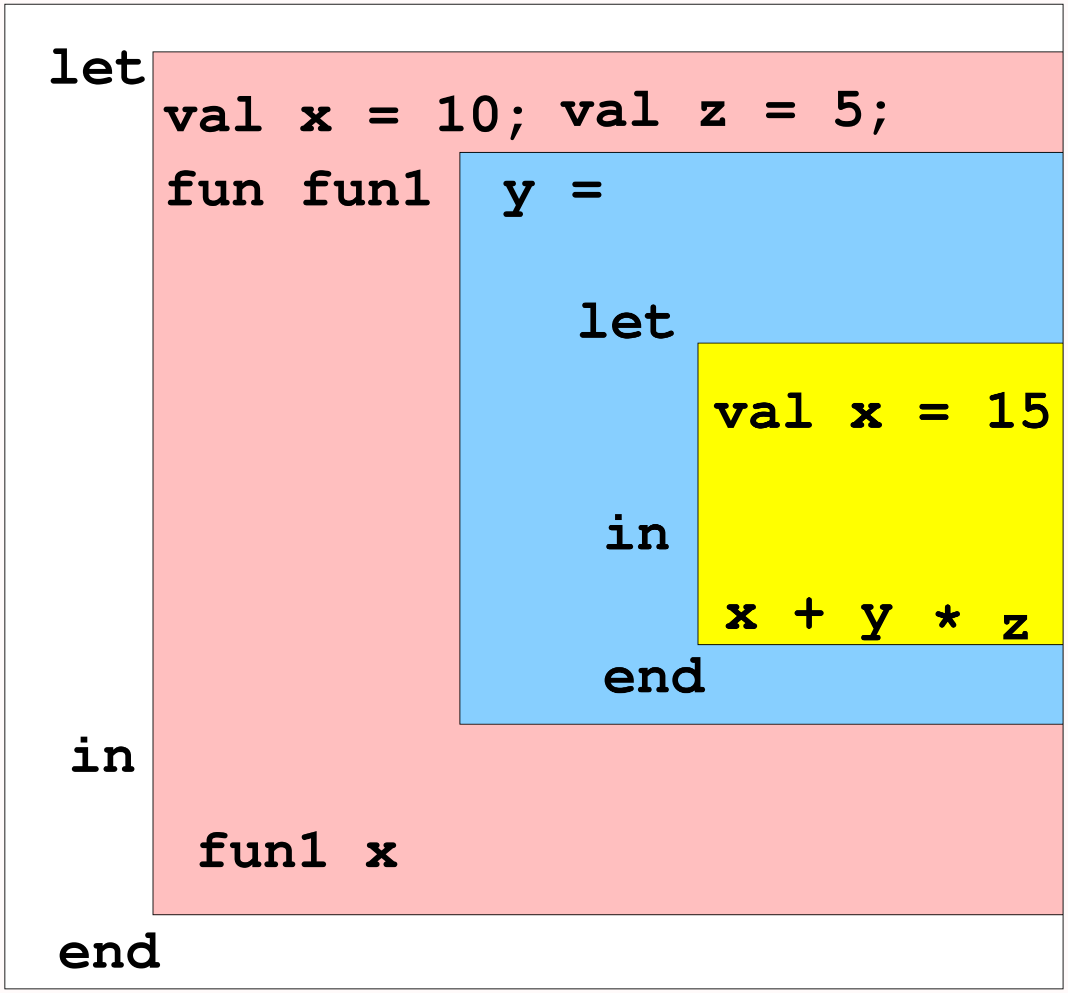
```
let
  val x = 10;
  fun fun1  y =
                    . . .
                    . . .
  fun fun2  z =
                    . . .
in                  . . .
  fun1 (fun2 x)
end
```

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

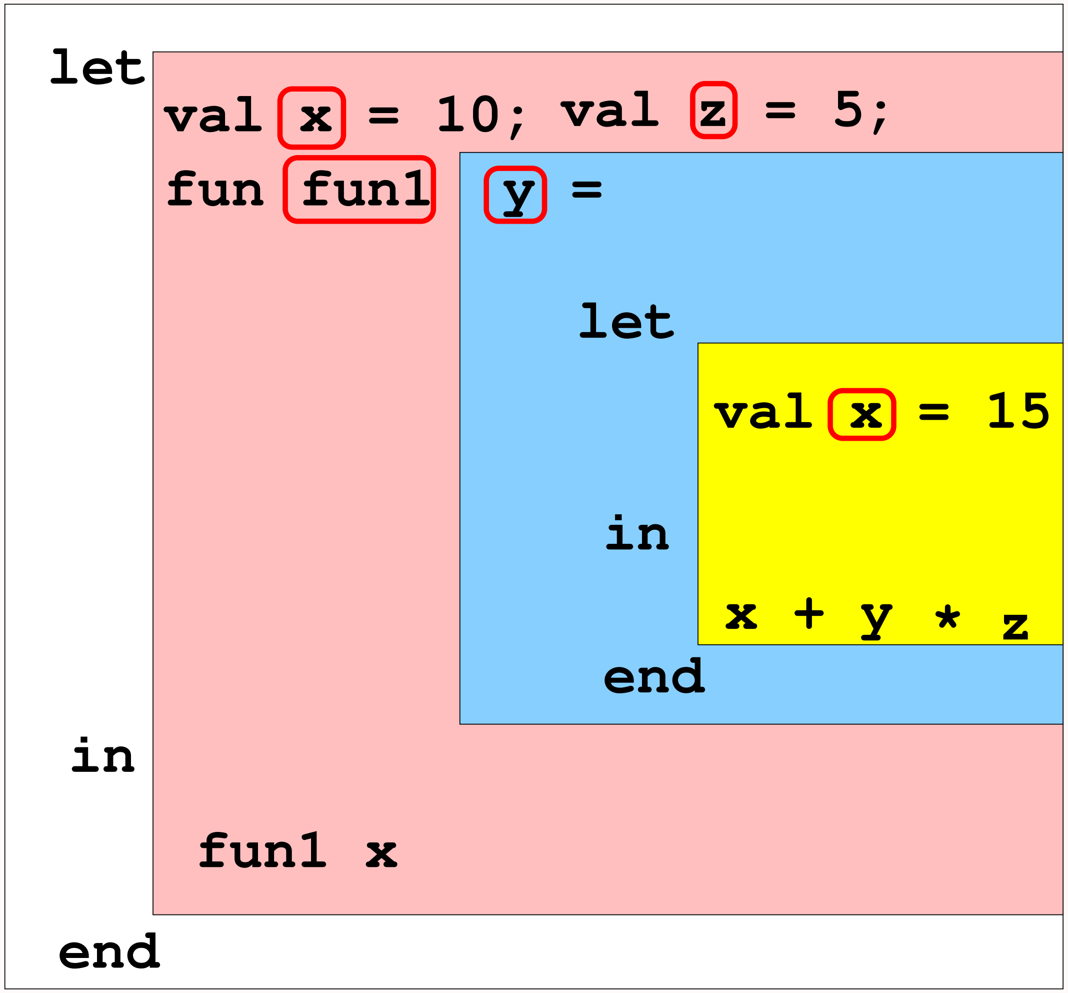
Names & References



```
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

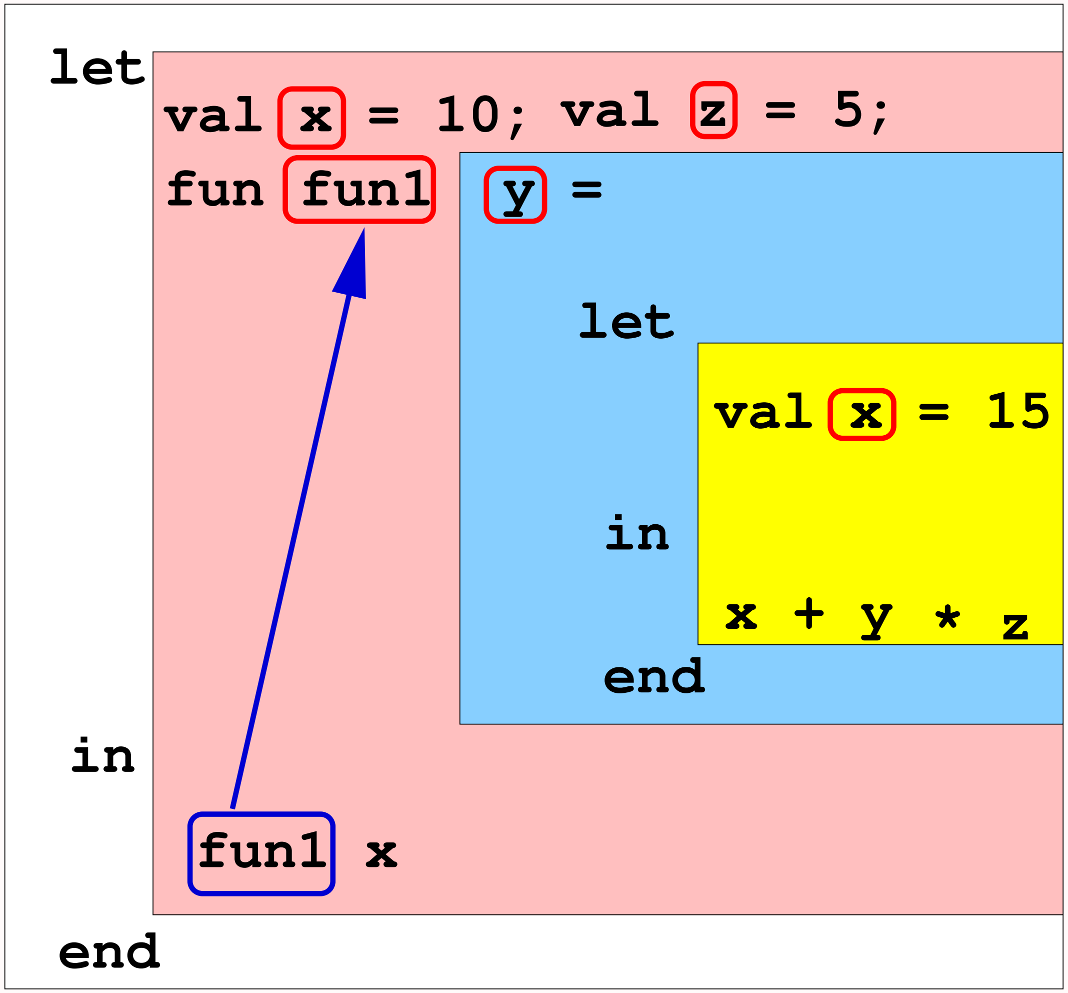
Names & References



```
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

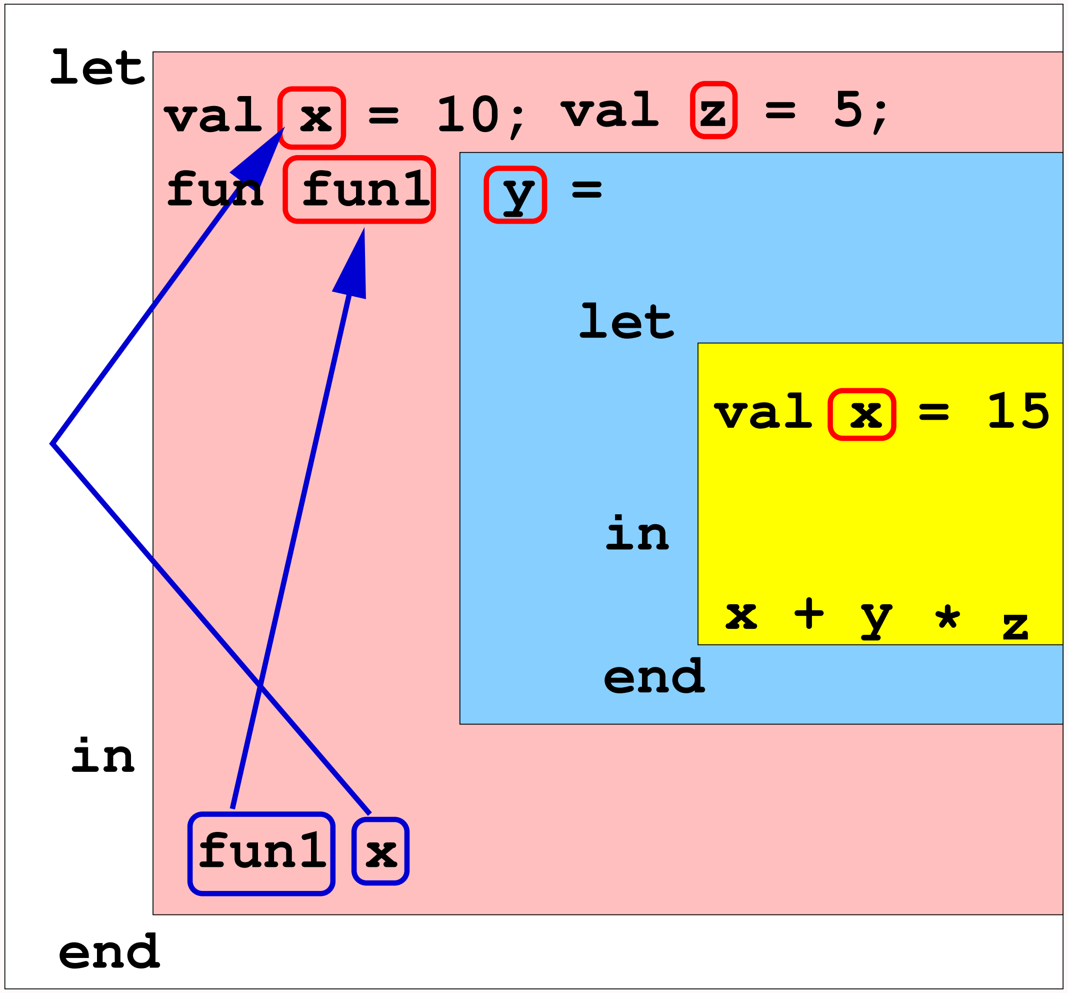
Names & References



```
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

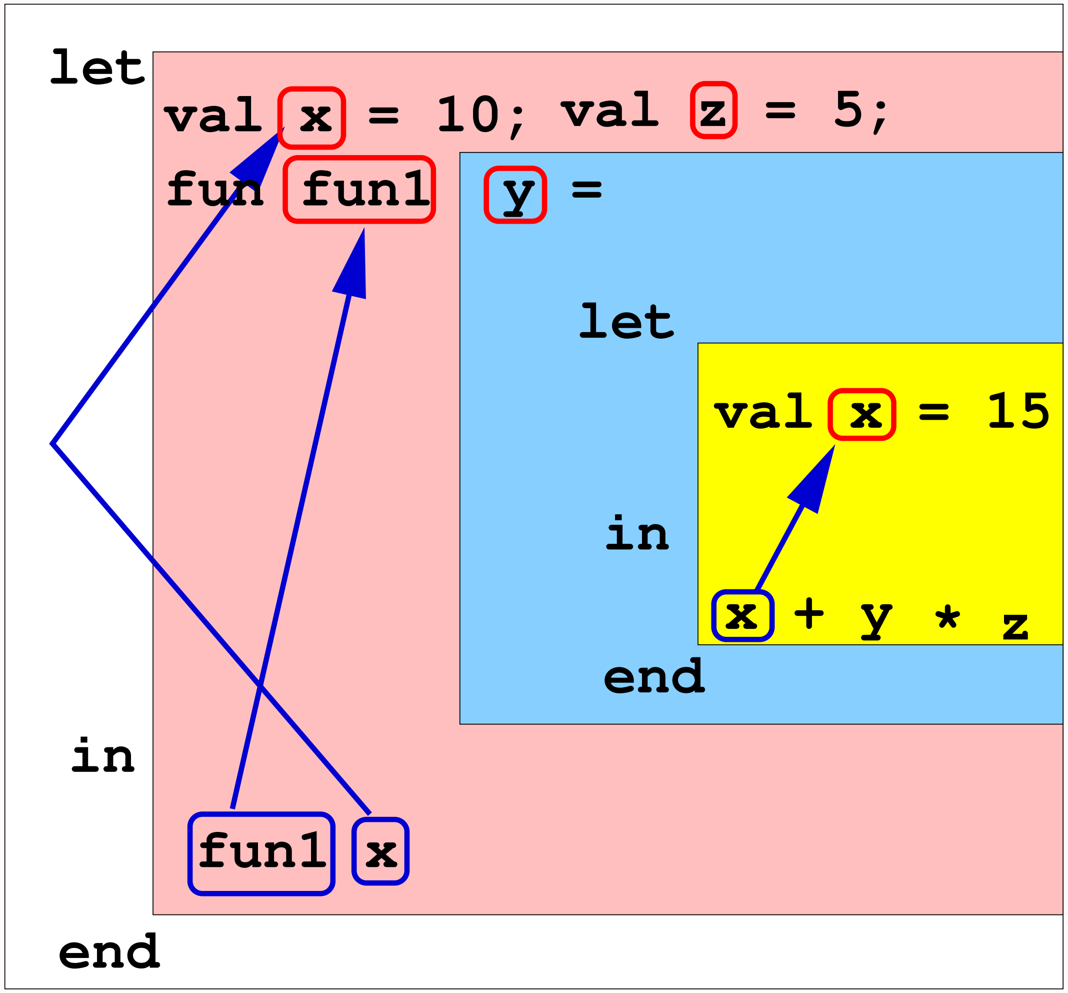
Names & References



```
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

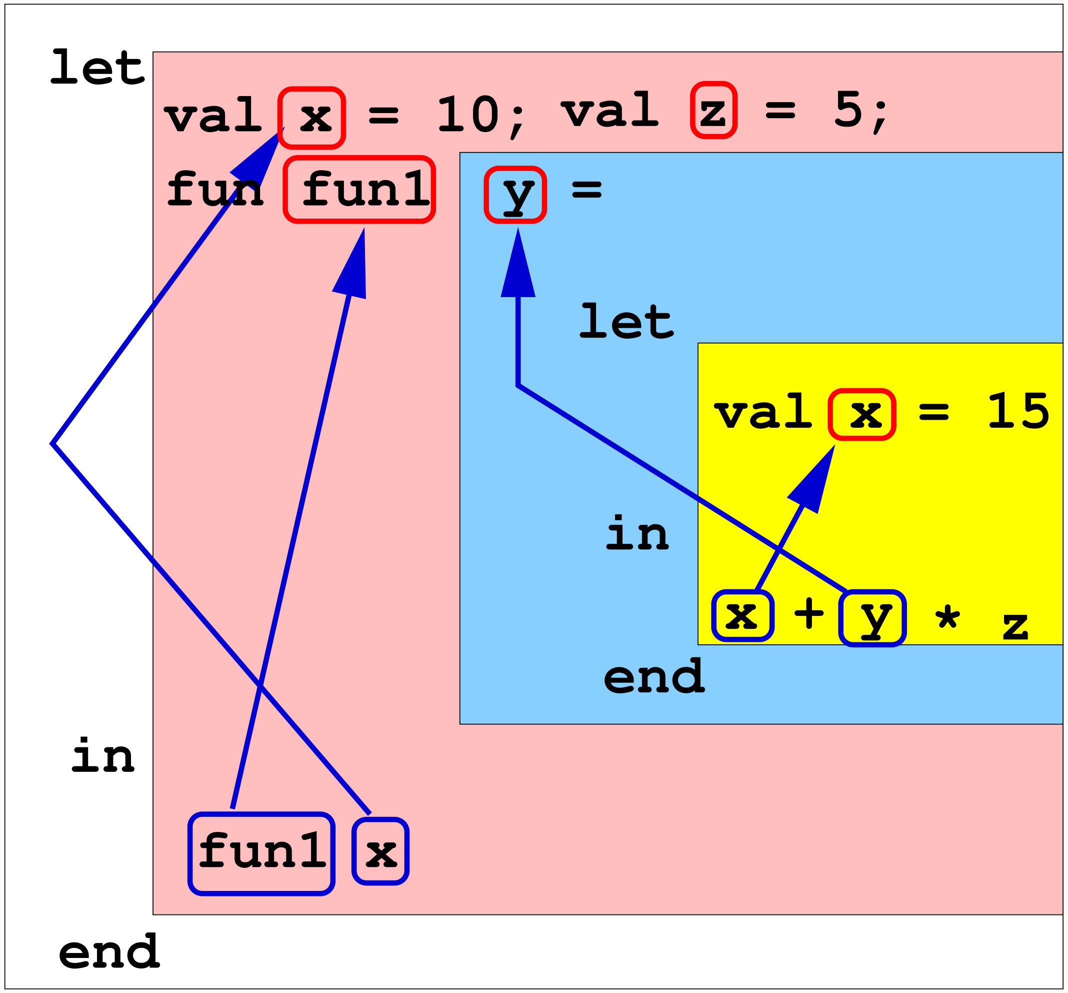
Names & References



```
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

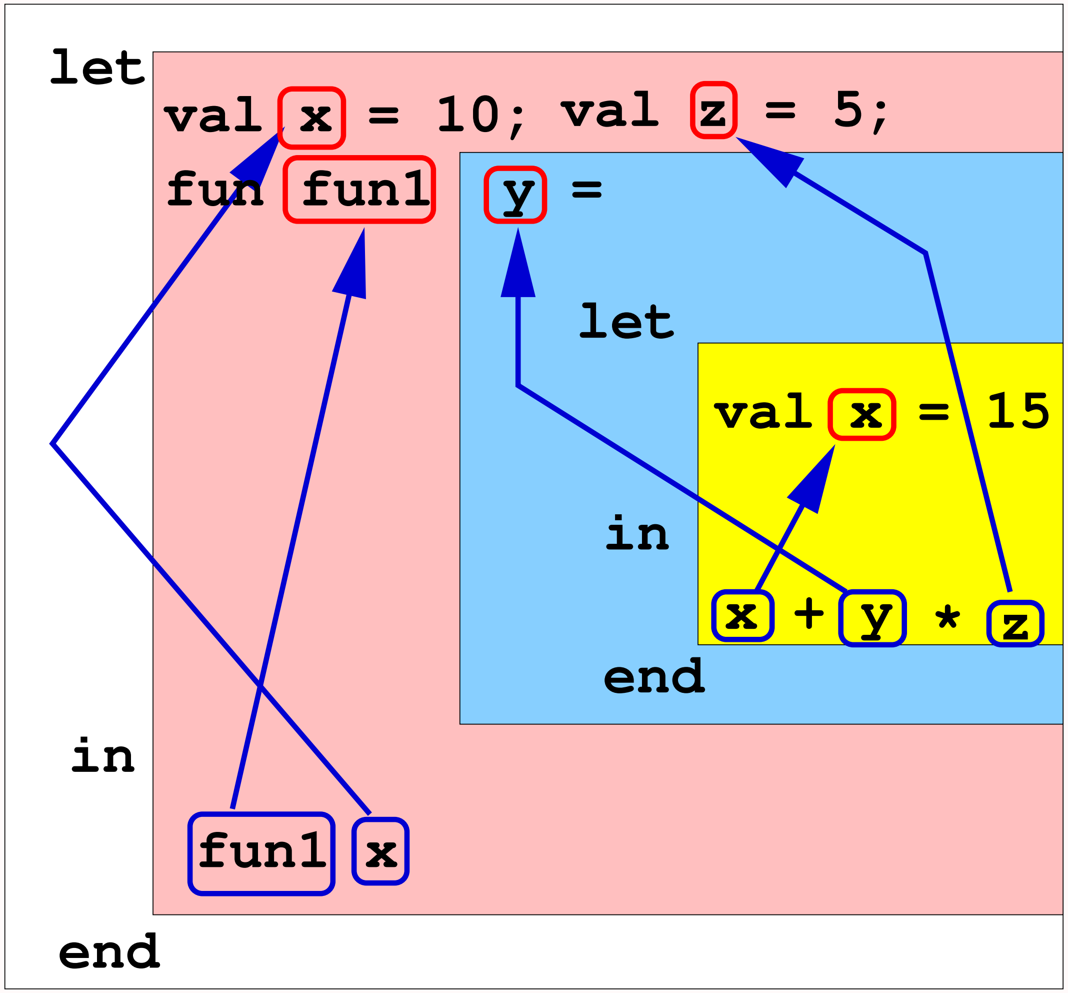
Names & References



```
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

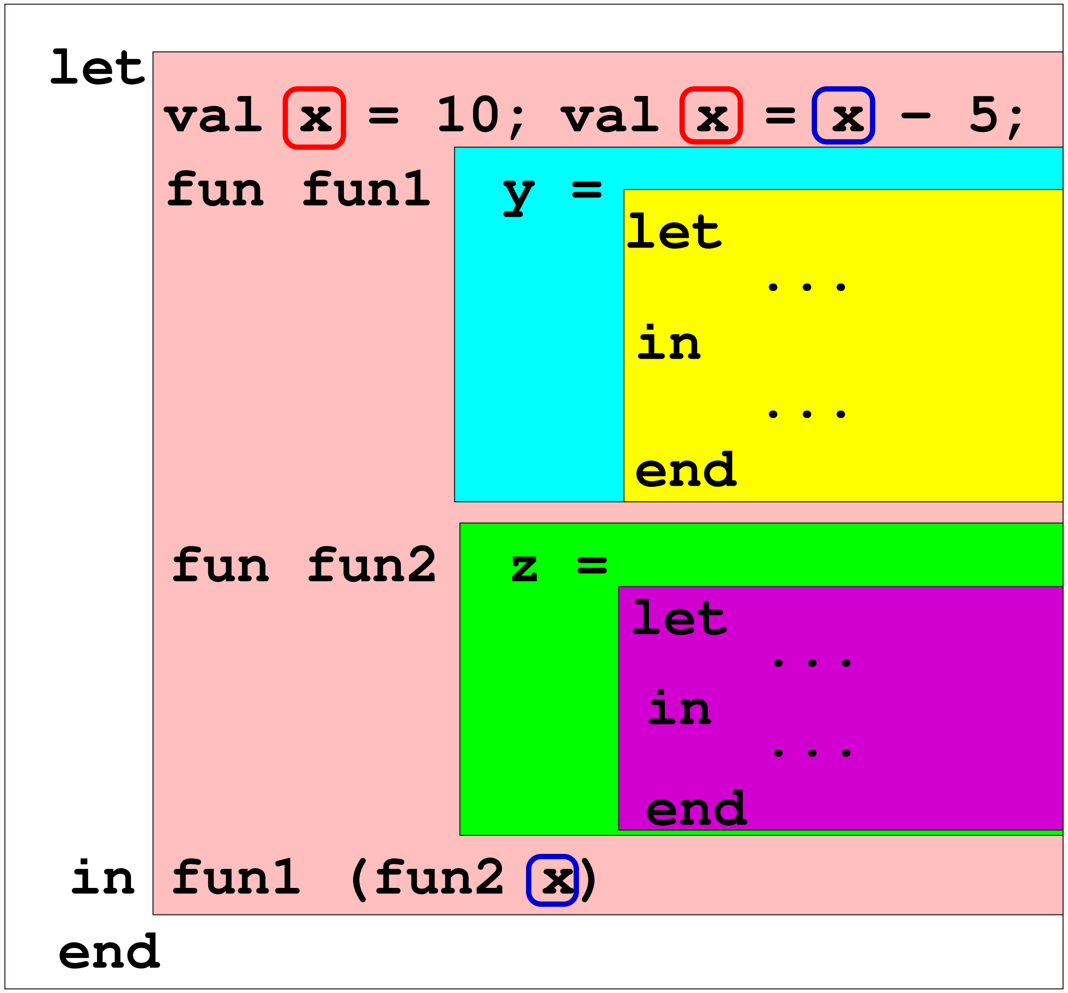
Names & References



```
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

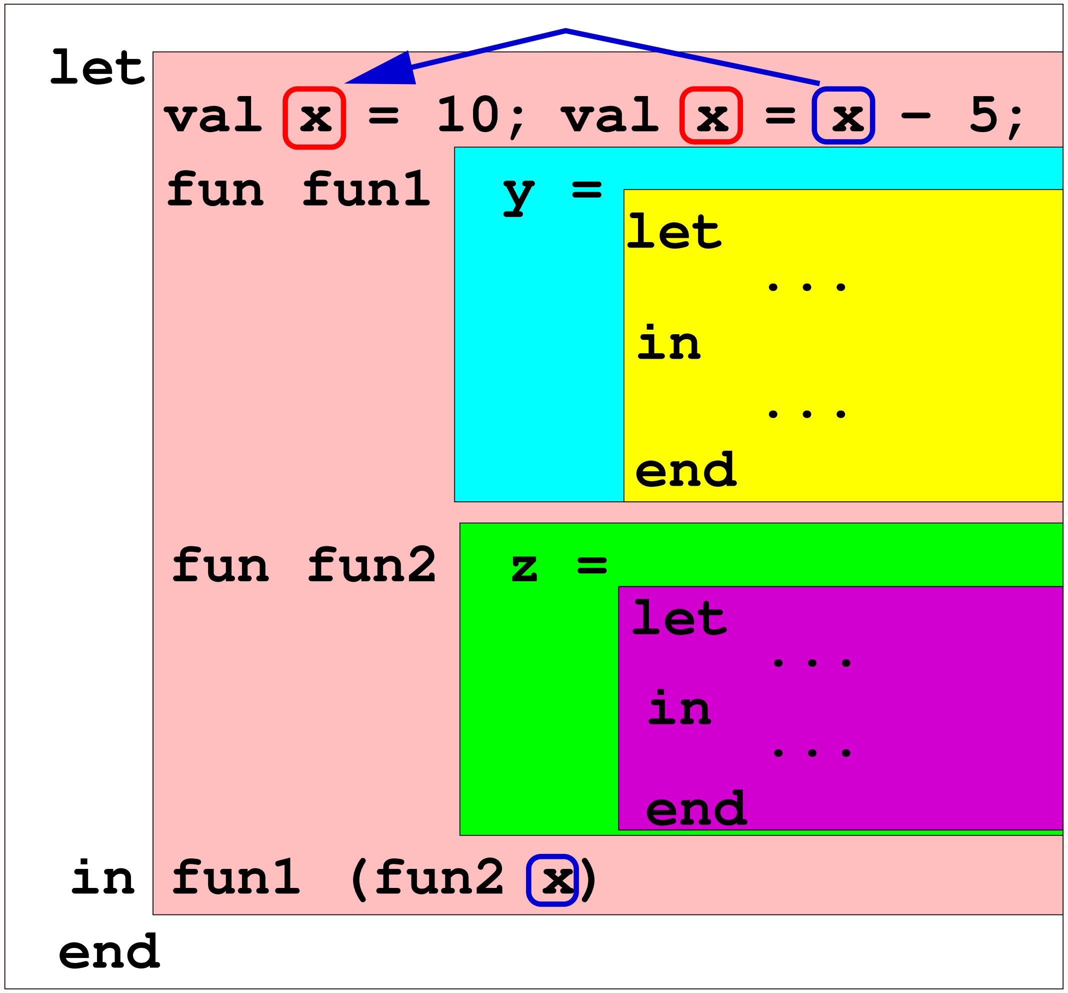
Names & References



```
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

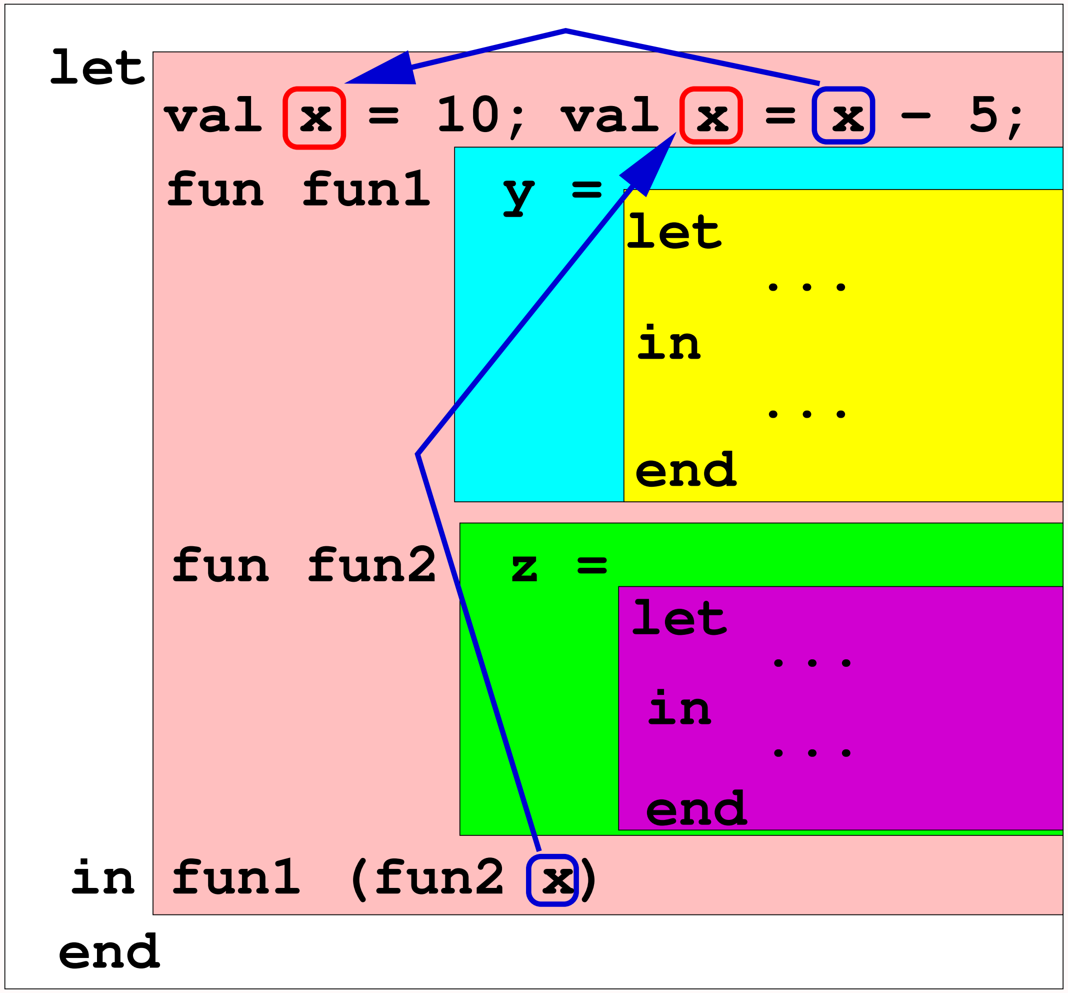
Names & References



```
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

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

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

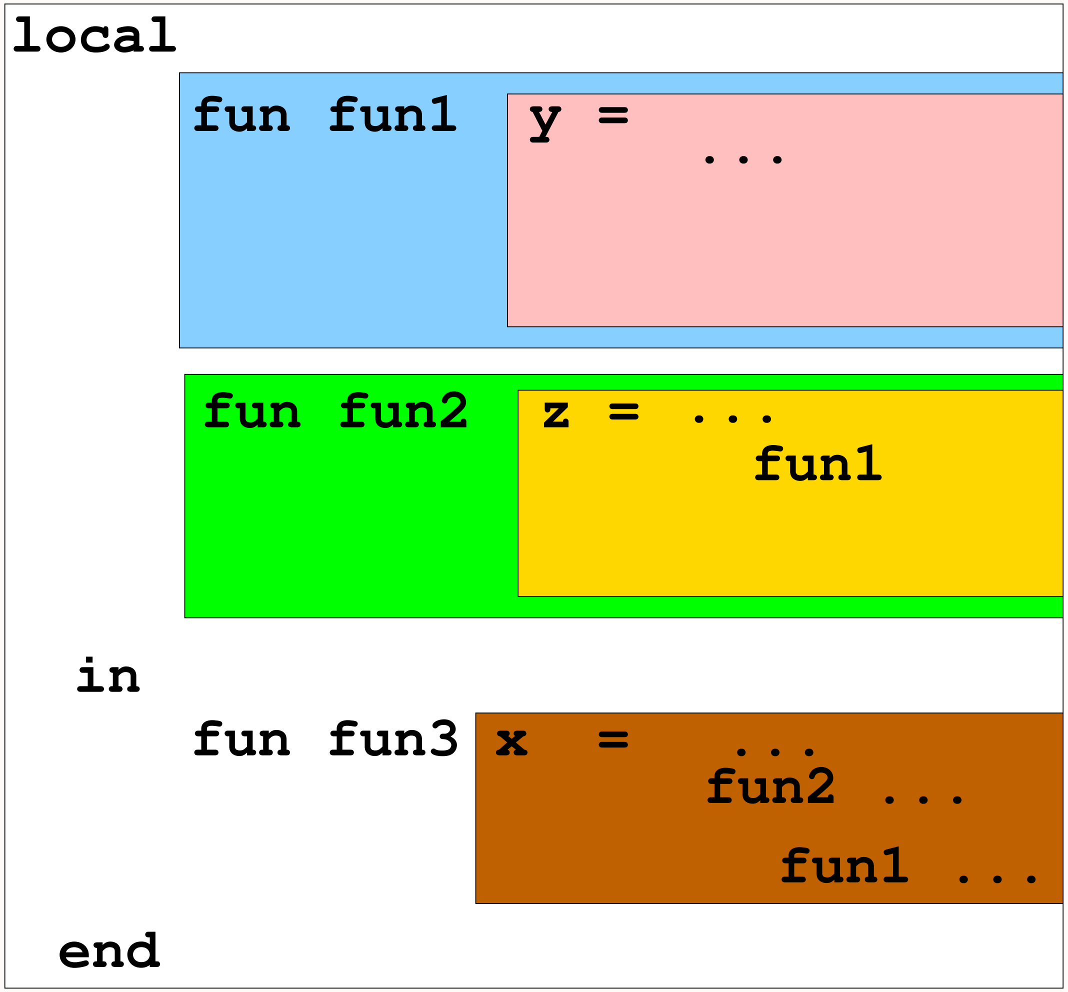
```
in
   fun perfect n =
      if n <= 0
      then raise invalidArg
      else
         let
            val nby2 = n div 2
         in
         n = sum_div2 (n, 1, nby2)
         end
end
```

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
```

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

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

Fermat Factorization

A beautiful method devised by Fermat for factoring a positive integer n into a *pair of factors*. The main virtue of the method is that

- it does not require knowing all the primes less than $\sqrt{n}$ to find a factor or to discover that n is a prime.
- the algorithm may be recursively applied to each of the pair of factors to obtain all the *prime factors*.

Fermat: Initial Refinement

Given any integer $n > 0$, Fermat's method

1. yields the pair $(2, n/2)$ if n is even,
2. otherwise (if n is odd) it attempts to find two factors m and j where $n = m.j$, $\sqrt{n} \leq m \leq n$ and $1 \leq j \leq \sqrt{n}$.

$$fermat0(n) = \begin{cases} (2, n/2) & \text{if } n \;\bar{\cdot}\; 2 = 0 \\ fermat1(n) & \text{otherwise} \end{cases} \tag{29}$$

Fermat: Factors of n

Fact 5.1 *The positive factors of* $n > 0$ *may be divided into two equal classes* $D(n)$ *and* $E(n)$ *such that*

$$D(n) = \{d \mid 1 \leq d \leq \sqrt{n}, d|n\} \tag{30}$$
$$E(n) = \{e \mid \sqrt{n} \leq e \leq n, e|n\} \tag{31}$$

and

1. *For every* $d \in D(n)$ *there is a unique* $e = n/d \in E(n)$*,*
2. *For every* $e \in E(n)$ *there is a unique* $d = n/e \in D(n)$*,*
3. n *is a perfect square if and only if* $\sqrt{n} \in D(n) \cap E(n)$ *and*
4. $1 \leq \min D(n) \leq \max D(n) \leq \sqrt{n} \leq \min E(n) \leq \max E(n) = n$*.*

Fermat: n is odd

Fact 5.2 *Assume n is odd. Now consider the smallest factor $m = \min E(n)$.*

Case n is a prime. *Then $m = n$.*

Case n is a perfect square. *Then $m = \sqrt{n} = \lceil \sqrt{n} \rceil = \lfloor \sqrt{n} \rfloor$.*

Case n is composite but not a perfect square. *Then $\lceil \sqrt{n} \rceil \leq m < \max E(n) \leq (n+1)/2$.*

Fermat: Refinement 1

fermat0

1. If n is a perfect square then it yields $(\sqrt{n}, \sqrt{n})$.
2. If n is an odd composite but not a perfect square, then it tries to find the *smallest* factor m such that $\lceil\sqrt{n}\rceil \leq m < n$ and yields the pair $(m, n/m)$. Notice n cannot have any proper factor greater than $(n+1)/2$. Hence $\lceil\sqrt{n}\rceil \leq m < (n+1)/2$.
3. If n is an odd prime then n is not a perfect square and the smallest factor m such that $\sqrt{n} \leq m \leq n$ is n itself and it yields the pair $(n, 1)$.

The above steps may be formalized by the following functional pseudo-code

$$fermat1(n) = \begin{cases} (k, k) & \text{if } odd(n) \wedge n = k^2 \\ findfactors(n, k, (n+1)/2) & \text{if } odd(n) \wedge k = \lceil\sqrt{n}\rceil \end{cases} \tag{32}$$

Fermat: Finding Factors

Since we *know* 3 is a prime, 4 is a perfect square and 5 is a prime, we assume $n > 5$.

Fact 5.3 *Assume $n > 5$ is odd and is not a perfect square. Then*

1. $\sqrt{n} < \lfloor n/2 \rfloor$.

2. If $n = a.b$, with $a \neq b$, $2 < a, b < n$, then

(a) Both a and b are odd.

(b) Without loss of generality, $1 \leq b < \sqrt{n} < a \leq n$.

(c) Both $a + b$ and $a - b$ are even.

(d) Every odd composite number is expressible as a difference of perfect squares.

$$n = [(a+b)/2]^2 - [(a-b)/2]^2 = x^2 - y^2 = (x+y).(x-y) \qquad (33)$$

(e) If n is an odd prime then $a = n$ and $b = 1$ and hence $x = (n+1)/2$ and $y = (n-1)/2$.

Fermat: Finding Factors 2

1. Hence one tries to solve the equation

$$x^2 - n = y^2 \tag{34}$$

   and takes the factors to be $(x + y)$ and $(x - y)$.

2. Notice that $x = (n + 1)/2$ and $y = (n - 1)/2$ satisfies the above identity for all odd n. However we would then have $x + y = n$ and $x - y = 1$. If $x = (n + 1)/2$ is the least value of x which solves equation (34) then there would be no divisors other than n and 1 which implies that n is a prime.

Fermat: Finding Factors 3

We have for any positive x, y which solve the equation (34)

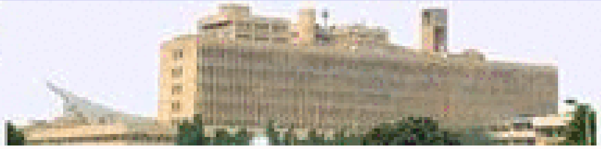
- $x^2 \geq n$
- $x > y > 0$ since $x^2 - y^2 = n > 0$,
- $x > \sqrt{n}$ since $x^2 - y^2 = n > 0$,
- $x - y < \sqrt{n} < x + y < n$ since both $(x + y)$ and $(x - y)$ are divisors of n,

Fermat: Finding Factors 4

fermat1

Assume n is odd and not a perfect square. Then let $k = \lceil\sqrt{n}\rceil$. Find the smallest m in the range $[k, (n+1)/2]$ such that $p = m^2 - n$ is a perfect square. Then the two factors are $m + \sqrt{p}$ and $m - \sqrt{p}$.

$$findfactors(n,m,q) = \begin{cases} (m + \lfloor\sqrt{p}\rfloor, m - \lfloor\sqrt{p}\rfloor) & \text{if } p = \lfloor\sqrt{p}\rfloor^2 \\ findfactors(n, m+1, q) & \text{if } p \neq \lfloor\sqrt{p}\rfloor^2 \wedge m < q \\ (n, 1) & \text{otherwise} \end{cases} \tag{35}$$



Fermat: The Complete Algorithm

fermat0 fermat1 findfactors

$$fermat0(n) = \begin{cases} (2, n/2) & \text{if } n \mathbin{\bar{\cdot}} 2 = 0 \\ fermat1(n) & \text{otherwise} \end{cases}$$

where

$$fermat1(n) = \begin{cases} (k, k) & \text{if } odd(n) \wedge n = k^2 \\ findfactors(n, k, (n+1)/2) & \text{if } odd(n) \wedge k = \lceil \sqrt{n} \rceil \end{cases}$$

where

$$findfactors(n, m, q) = \begin{cases} (m + \lfloor \sqrt{p} \rfloor, m - \lfloor \sqrt{p} \rfloor) & \text{if } p = \lfloor \sqrt{p} \rfloor^2 \\ findfactors(n, m+1, q) & \text{if } p \neq \lfloor \sqrt{p} \rfloor^2 \wedge m < q \\ (n, 1) & \text{otherwise} \end{cases}$$

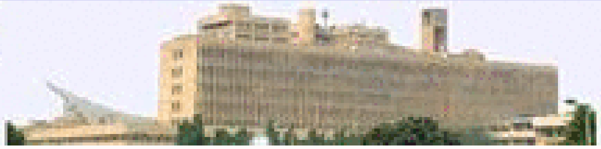
```
fun odd n = (n mod 2 = 1)
fun even n = not (odd n)
fun sqr n = n*n
fun isqrtceil n = ceil (Math.sqrt (real n))
local
    fun findfactors (n, m, q) = (* fermat's method *)
        let val p = sqr (m) - n
            val isqrtfloorp = floor(Math.sqrt (real p))
        in  if p = sqr(isqrtfloorp)
            then (m + isqrtfloorp , m - isqrtfloorp)
            else if m < q
            then findfactors (n, m+1, q)
            else (n, 1)
        end
    fun fermat1 n =
        let val k = isqrtceil n
        in  if n = k*k then (k, k)
            else findfactors (n, k, (n+1) div 2)
        end
in fun fermat0 n = if (even n) then (2, n div 2) else fermat1 n
end
```

6. Introducing Reals

6.1. 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

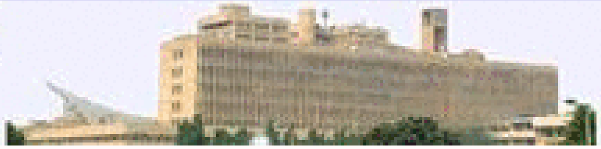
So far-5: Caveats

- 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 $\ldots$
  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 ∞

$$
\begin{aligned}
e^x &= \sum_{m=0}^{\infty} \frac{x^m}{m!} \\
\cos x &= \sum_{m=0}^{\infty} \frac{(-1)^m x^{2m}}{(2m)!} \\
\sin x &= \sum_{m=0}^{\infty} \frac{(-1)^m x^{2m+1}}{(2m+1)!}
\end{aligned}
$$

Truncation

Truncation Errors

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

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

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

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

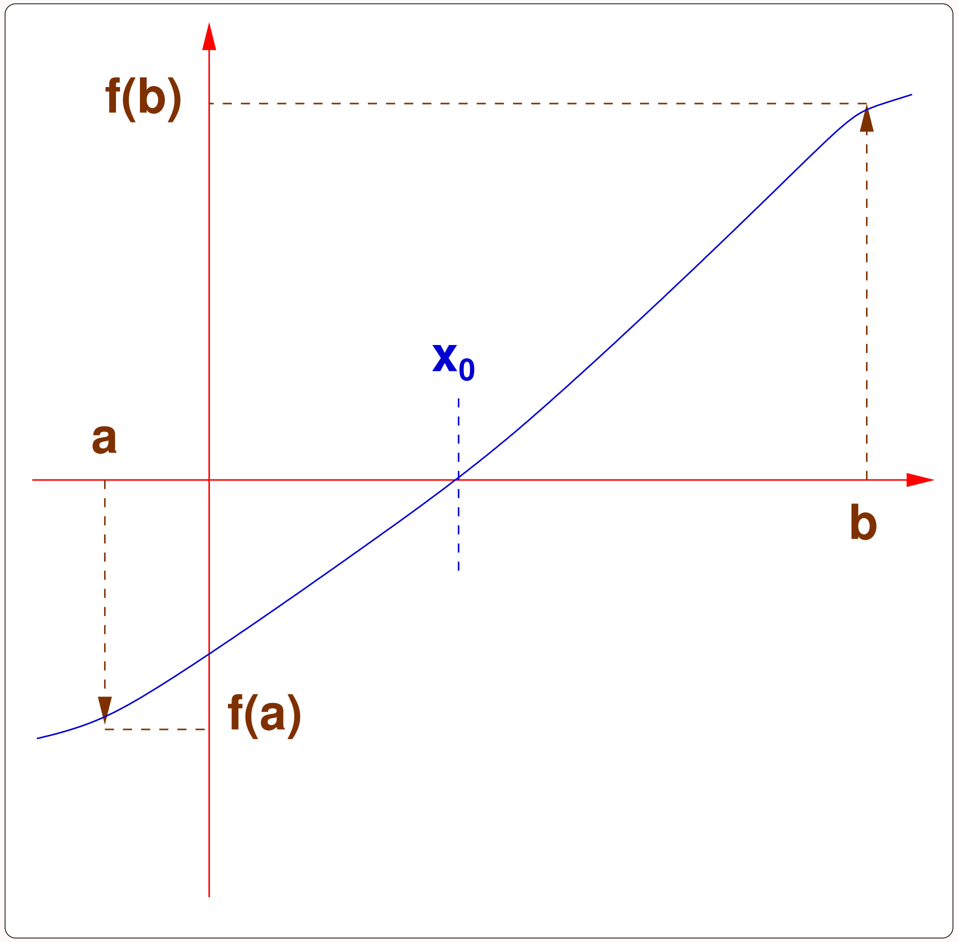
back to Errors

Equation Solving

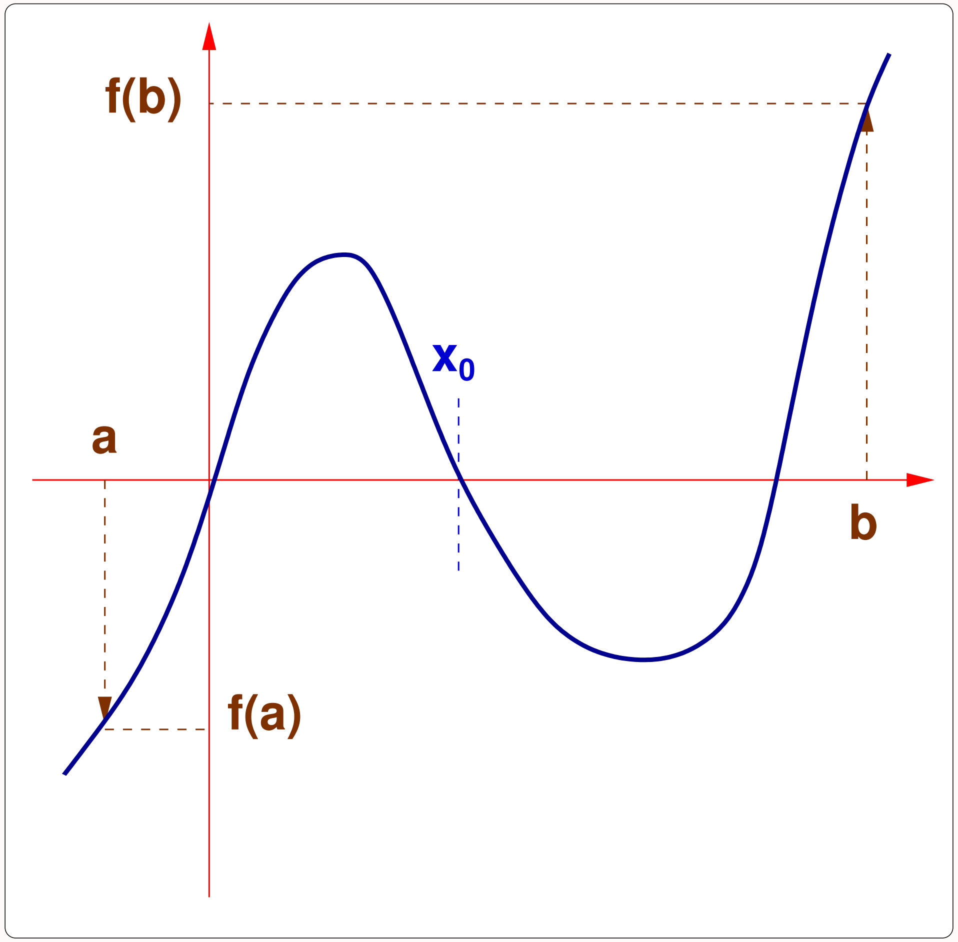
- The fifth most basic operation
- Root finding: A particular form of equation solving

$$f(x) = 0$$

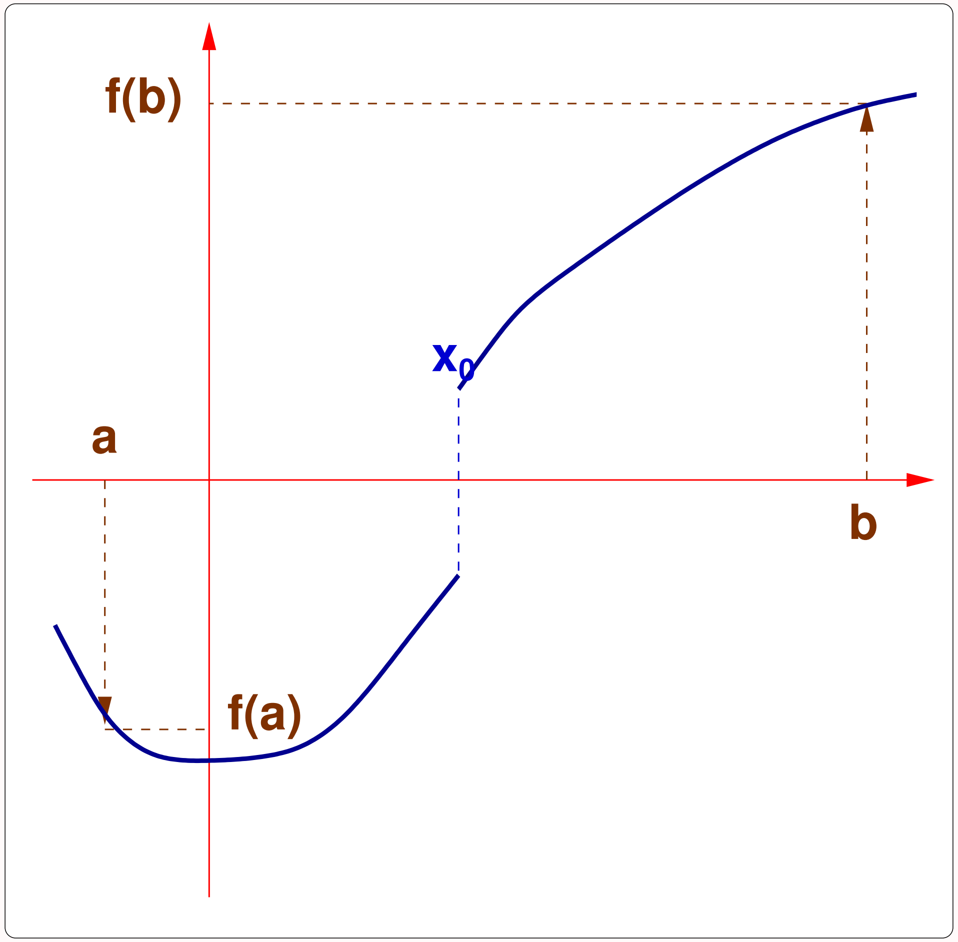
Root Finding-1



Root Finding-2

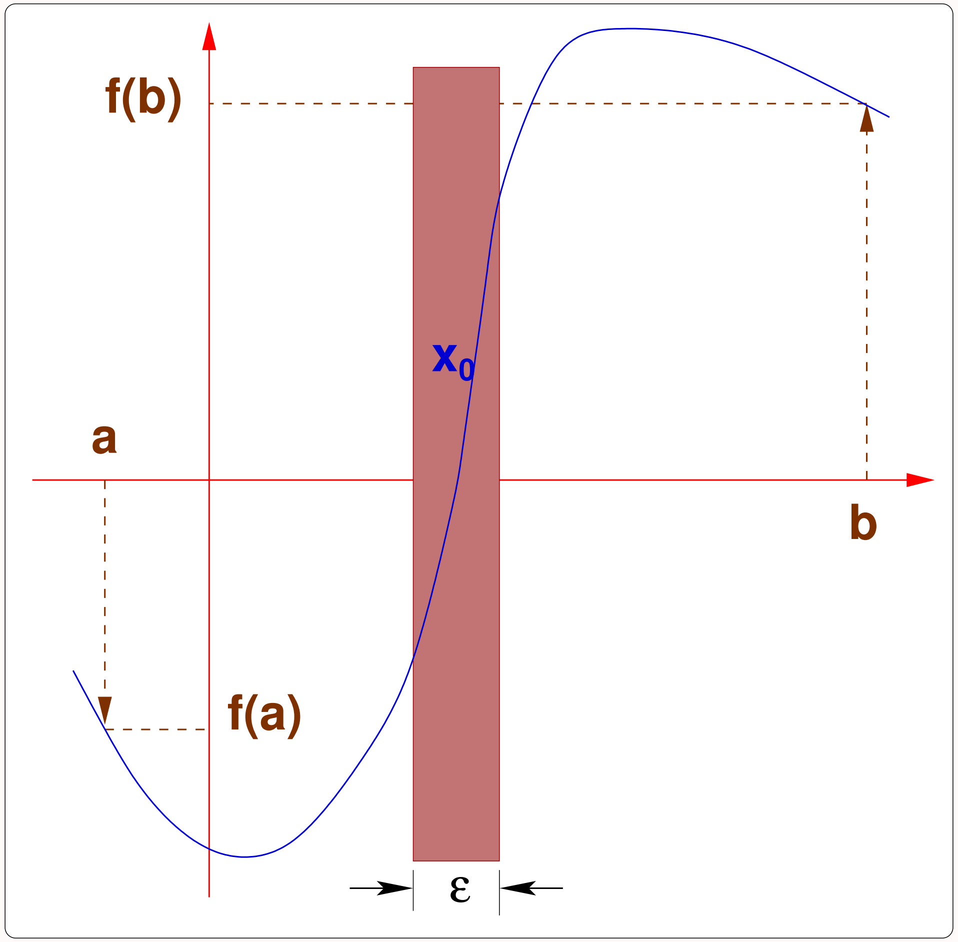


Root Finding-3



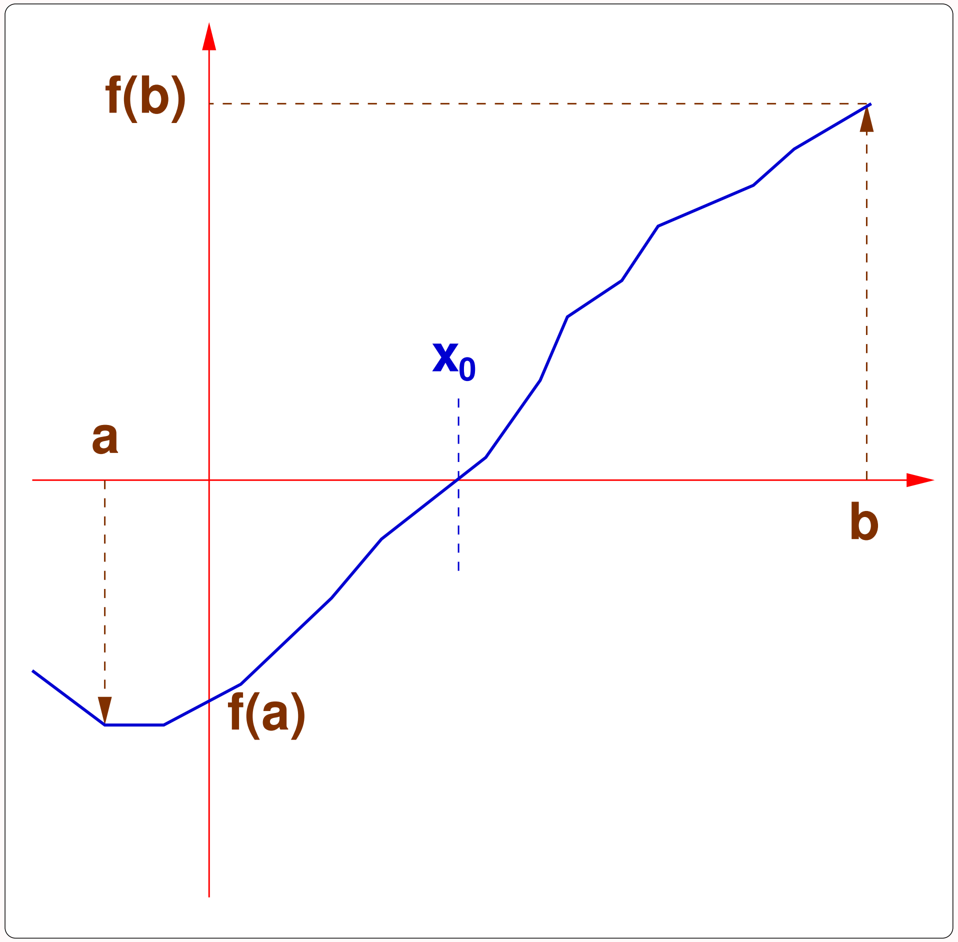
Root Finding-4

Rather steep isn't it?



Root Finding-5

Continuous functions which are not smooth?



The Bisection Method

- Given that
  - $f : \mathbb{R} \longrightarrow \mathbb{R}$ is continuous
  - a and b ($a < b$) are real values such that $f(a)$ and $f(b)$ have different signs
- $f(x) = 0$ does have one (or more) solution(s) in the interval $[a, b]$.

One may take inspiration from $shrink2$ to find a root of the equation $f(x) = 0$, by bisecting the interval $[a, b]$.

Root Finding: Bisection

Choose a sufficiently small value $\varepsilon > 0$. We try to find an approximation to the root as follows:

$$root\_bisect(a,b) = \begin{cases} a & \text{if } |f(a)| \leq \varepsilon \\ b & \text{if } |f(b)| \leq \varepsilon \\ root\_bisect(a,m) & \text{else if } sgn(f(m)) = sgn(f(b)) \\ root\_bisect(m,b) & \text{else if } sgn(f(m)) = sgn(f(a)) \end{cases}$$

where $m = (a+b)/2.0$.

Bisection Method: Cases

1. Eventually obtain an approximation in case 1
2. Eventually obtain an approximation to one of the roots in case 2
3. May never find one if it is not continuous as in case 3
4. May go on forever if the curve is too steep as in case 4
5. But works even with continuous functions that are not everywhere differentiable within the interval

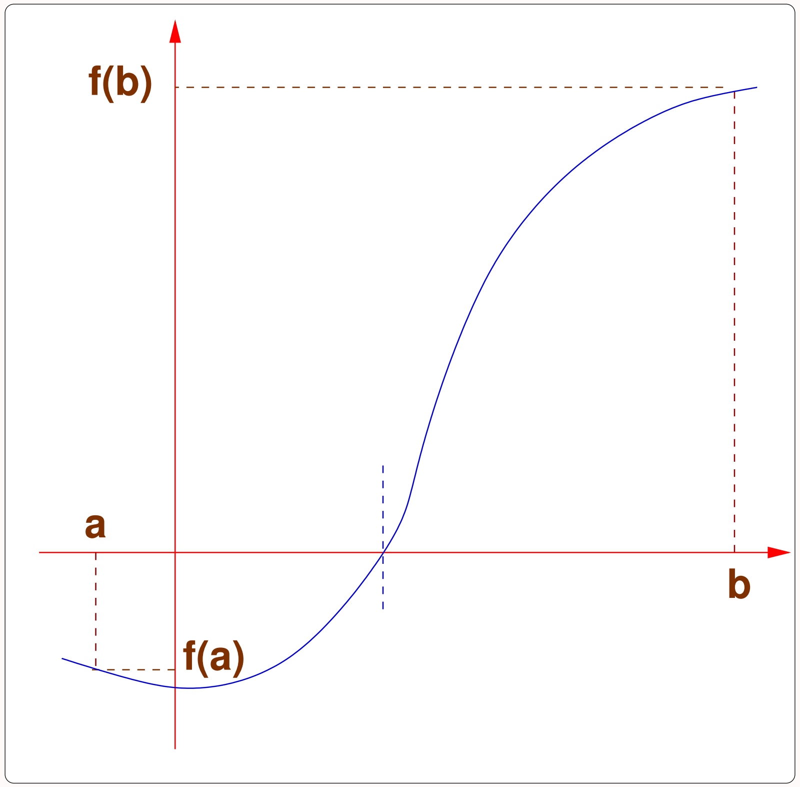
6.2. Root Finding, Composition and Recursion

Root Finding: Newton’s Method

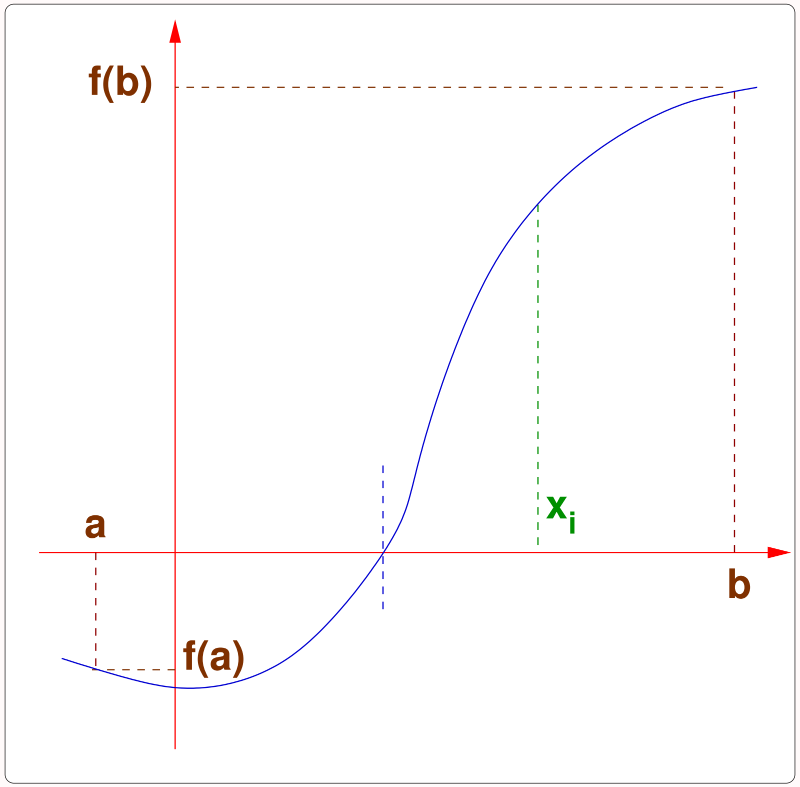
Consider a function $f(x)$

- 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

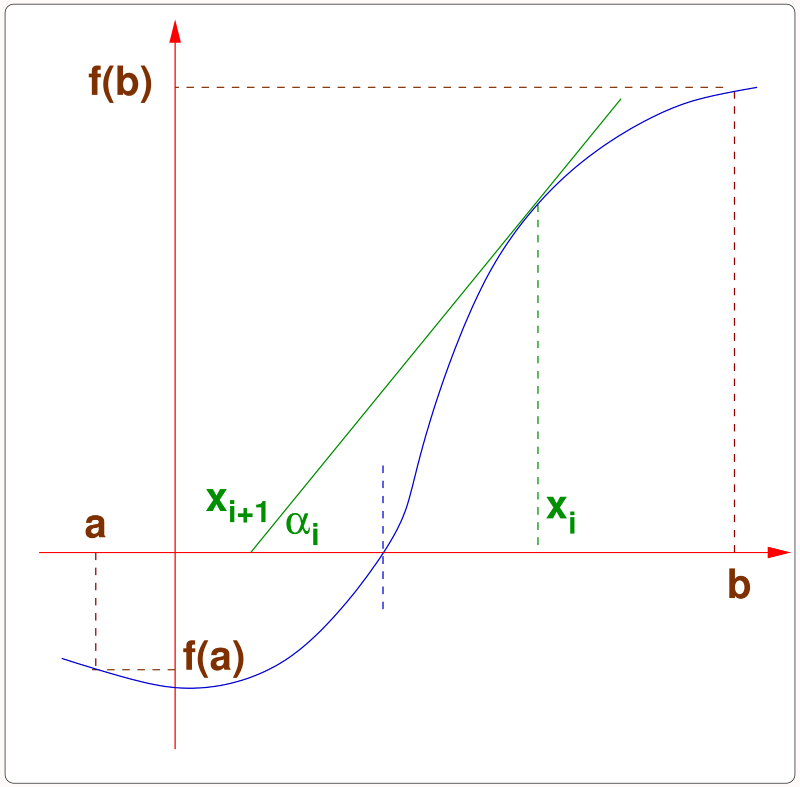
Root Finding: Newton’s Method



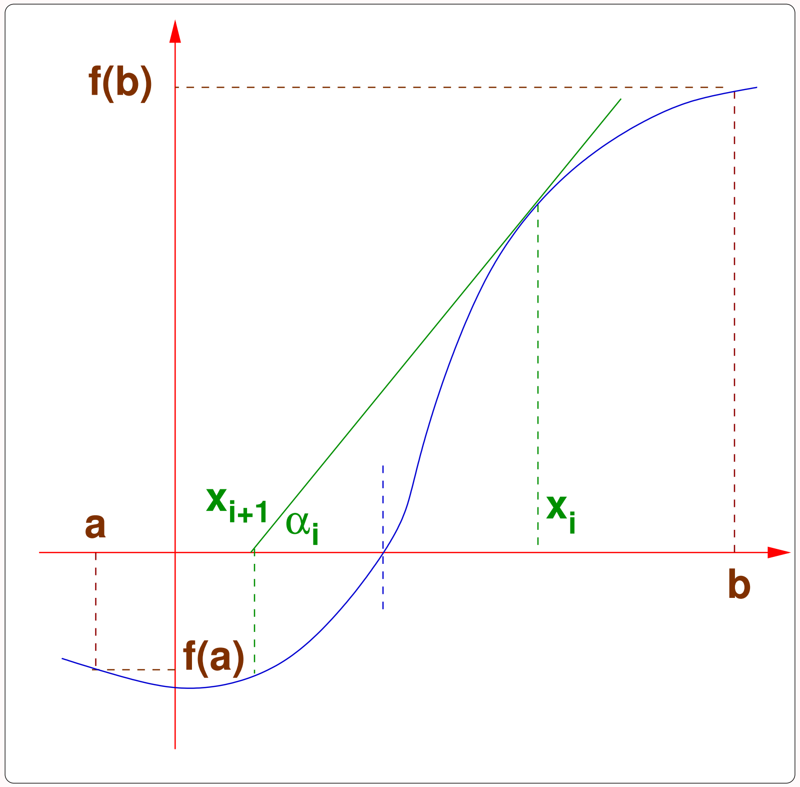
Root Finding: Newton’s Method



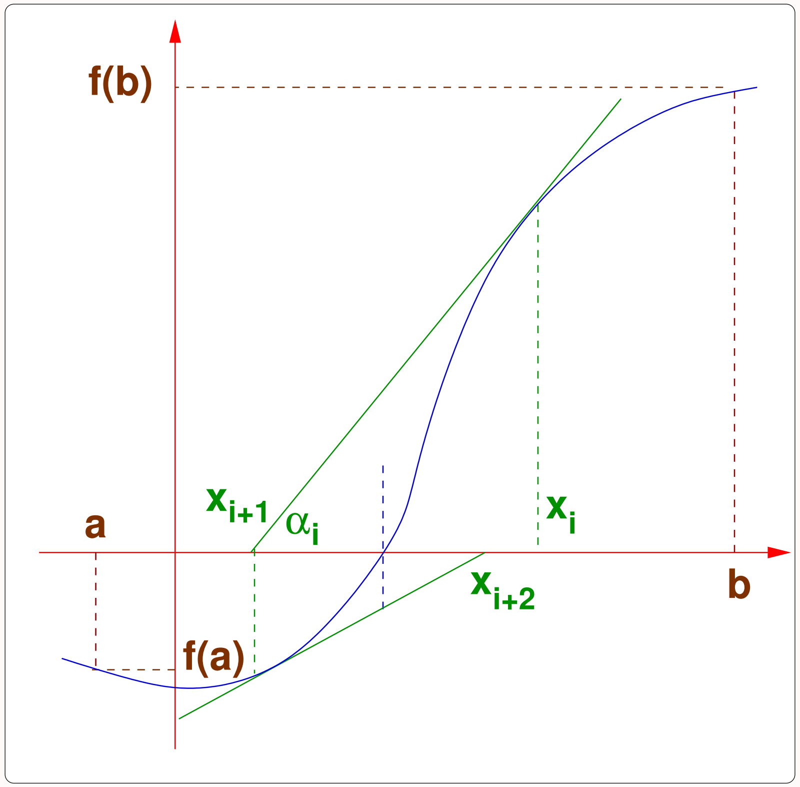
Root Finding: Newton’s Method



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

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

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's 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

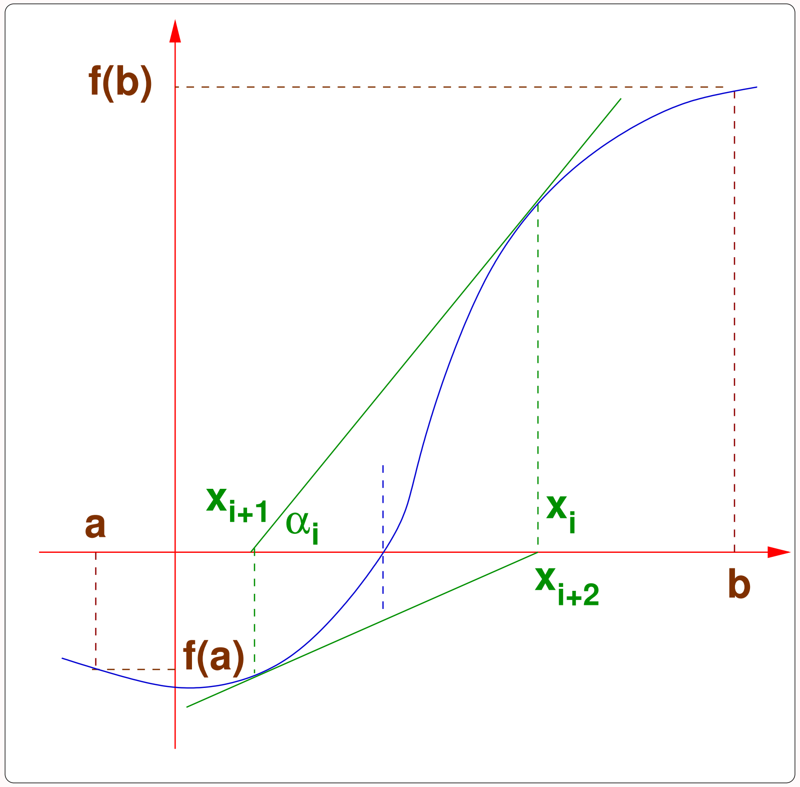
$$x_0 \in [a, b]$$

and

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

What can go wrong!-1

Oscillations!

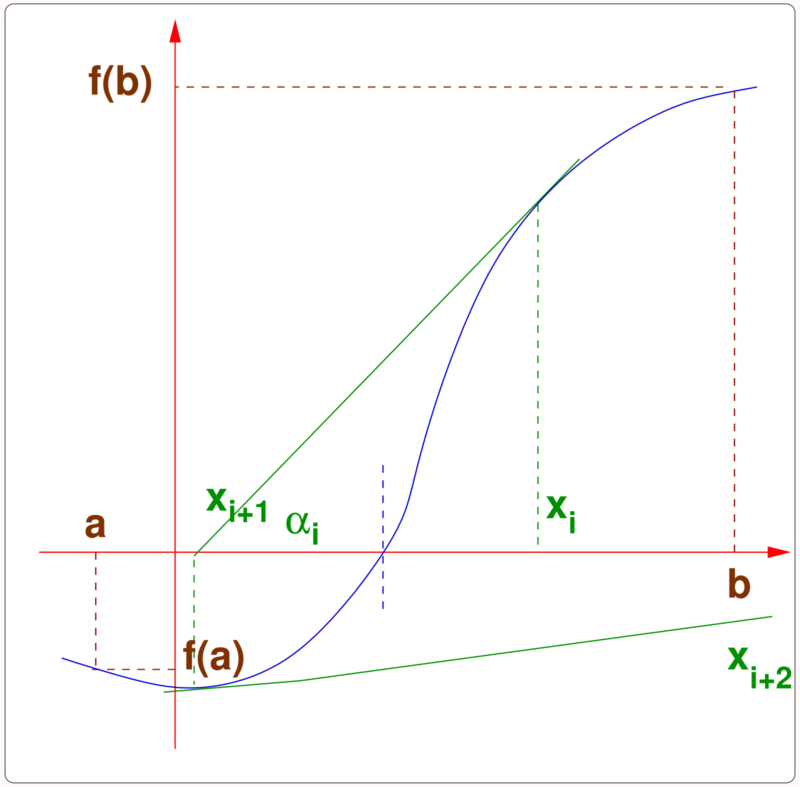


What can go wrong!-2

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.

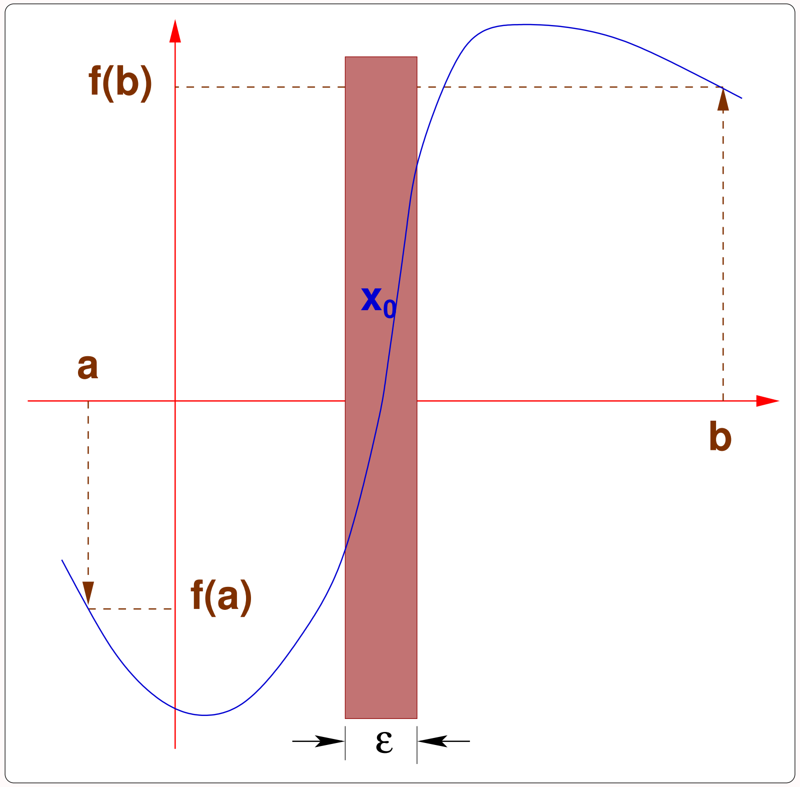
What can go wrong!-2

Interval bounds error!



What can go wrong!-3

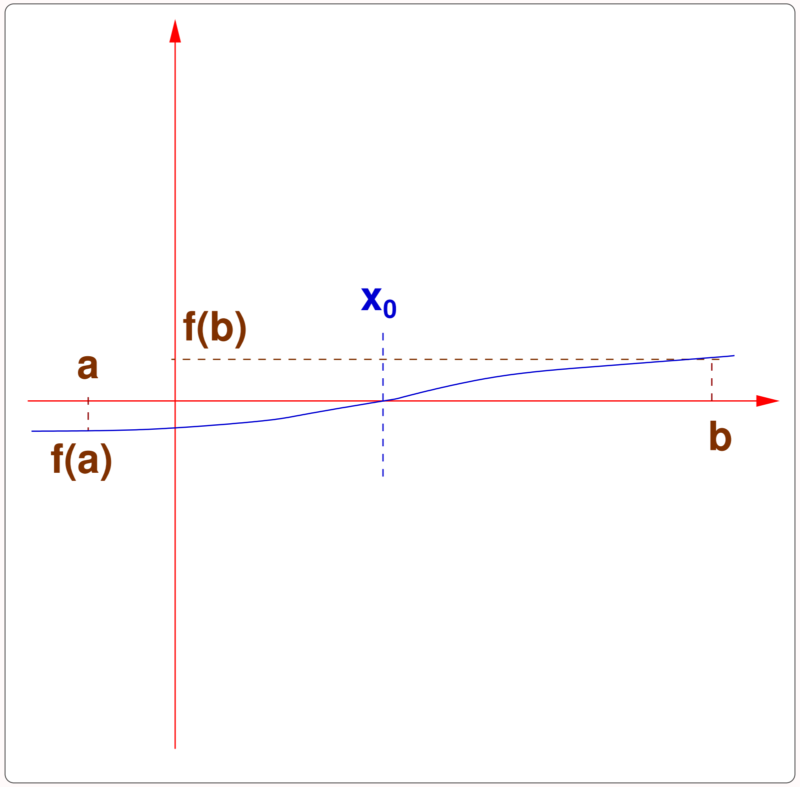
The function may be too steep



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

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

```
exception Negative_tolerance
exception Empty_interval
exception Out_of_bounds
fun newton (f, f', a,b, e, x:real) =
    let val y = f (x);
        val y' = f' (x);
     in if e < 0.0 then raise Negative_tolerance
        else if a >= b then raise Empty_interval
        else if x < a orelse x > b then raise  Out_of_bounds
        else if Real.abs(y) < e
        then x
        else newton (f, f', a,b,e, x - y/y')
     end;



(* Try it out *)

use "ics/fastpower.sml";

fun f1(x) = power2(x, 6) - x -1.0
fun f1'(x) = 6.0*power2(x, 5) -1.0

(*
 newton (f1, f1', 1.0, 2.0, 0.00005, 1.5);
 val it = 1.13472414532 : real
*)

(* We could also count the number of iterations *)

fun newton2 (f, f', a,b, e, x:real, c) =
    let val y = f (x);
        val y' = f' (x);
     in ( print (Int.toString(c)^". "^Real.toString(x)^"\n");
          if e < 0.0 then raise Negative_tolerance
          else if a >= b then raise Empty_interval
          else if x < a orelse x > b then raise  Out_of_bounds
          else if Real.abs(y) < e then x
          else newton2 (f, f', a,b,e, x - y/y', c+1)
        )
     end
```

```
(* Try it out *)
(*
- newton2 (f1, f1', 1.0, 2.0, 0.00005, 1.5, 0);
0. 1.5
1. 1.30049088359
2. 1.1814804164
3. 1.13945559028
4. 1.13477762524
5. 1.13472414532
val it = 1.13472414532 : real

We may verify the result

- f1(it);
val it = 7.11358523198E~08 : real

*)

fun f2 x = 2.0*Math.sin(x) - x

fun f2' x = 2.0*Math.cos(x) -1.0
```

```
(* We get

-  f2(0.5*Math.pi);
val it = 0.429203673205 : real
- f2(Math.pi);
val it = ~3.14159265359 : real

Therefore there is root in the interval [0.5*Math.pi, Math.pi] besides the
obvious root at 0.0

*)
(*
- newton2 (f2, f2', 0.5*Math.pi, Math.pi, 0.00001, 0.75*Math.pi, 0);
0. 2.35619449019
1. 1.96601321439
2. 1.89811156854
3. 1.8954982162
val it = 1.8954982162 : real
*)
```

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 $h(1)$

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

Two Computations of $h(-1)$

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

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

$$
\begin{array}{ll}
 & factR(4) \\
\leadsto & (4 * factR(4-1)) \\
\leadsto & (4 * factR(3)) \\
\leadsto & (4 * (3 * factR(3-1))) \\
\leadsto & (4 * (3 * factR(2))) \\
\leadsto & (4 * (3 * (2 * factR(2-1)))) \\
\leadsto & \cdots
\end{array}
$$

Recursion: Nonlinear

Fibonacci

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

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 $factR$?

7. Correctness, Termination & Complexity

7.1. 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

$$
\begin{aligned}
&\leadsto ((((1*1)*2)*3)*4)\\
&\leadsto (((1*2)*3)*4)\\
&\leadsto ((2*3)*4)\\
&\leadsto (6*4)\\
&\leadsto 24
\end{aligned}
$$

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*

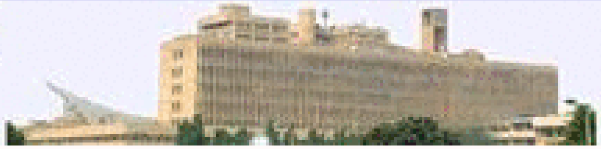
$$\leadsto (((1+1)+F_2(1))+F_2(2))+F_2(3)$$
$$\leadsto (2+F_2(1))+F_2(2))+F_2(3)$$
$$\leadsto ((2+1)+F_2(2))+F_2(3)$$
$$\leadsto \cdots$$

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

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

Fibonacci

The proof is similar to that of $factL$.

Basis For $n = 0$ or $n = 1$ $F_2(n) = 1$.

Induction hypothesis For some $n > 0$, $\forall k : 0 \leq k \leq n : F_2(k)$ terminates in $\propto f(k)$ steps

Induction Step Then since each of $F_2(n)$ and $F_2(n-1)$ is guaranteed to terminate in $\propto f(n)$ and $\propto f(n-1)$ steps $F_2(n+1) = F_2(n) + F_2(n-1)$ is also guaranteed to terminate in $f(n+1) \propto f(n) + f(n-1)$ steps.

GCD: Definition

Assume a and b are non-negative integers.

$$gcd(a,b) = \begin{cases} m & \text{if } a = 0 \vee b = 0 \vee m = n \\ gcd(m-n, n) & \text{otherwise} \end{cases}$$

where $m = max(a,b)$ and $n = min(a,b)$.

GCD computations

GCD

$$\begin{array}{ll} & gcd(12, 64) \\ \leadsto & gcd(12, 52) \\ \leadsto & gcd(12, 40) \\ \leadsto & gcd(12, 28) \\ \leadsto & gcd(12, 16) \\ \leadsto & gcd(12, 4) \\ \leadsto & gcd(8, 4) \\ \leadsto & gcd(4, 4) \\ \leadsto & 4 \end{array}$$

Well-foundedness: GCD

GCD

The sequence of maximum of the arguments obtained in the computation is well-founded i.e.

- it is a sequence of *non-negative integers*, and
- it is *strictly decreasing* sequence bounded below by 0.

$$m_1 > m_2 > \cdots > m_{k-1} > m_k \geq 0$$

where $m_1 = max(a, b)$ and if $n_1 = min(a, b)$ then for each i, $k > i > 1$ we have $m_i > m_{i+1} = max(m_i - n_i, n_i)$.

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 F_2 are defined only for non-negative integers
- The argument to $factL$ and F_2 in each recursive unfolding is strictly decreasing.

Euclidean GCD

The original algorithm by Euclid went as follows:

$$eugcd(a,b) = \begin{cases} m & \text{if } a = 0 \vee b = 0 \vee m = n \\ eugcd(m\%n, n) & \text{otherwise} \end{cases}$$

where $m = max(a,b)$, $n = min(a,b)$ and $m\%n$ is the remainder obtained on dividing m by n. Notice that in this case

$$m \geq n > m\%n \geq 0$$

is guaranteed. We may argue that the sequence of remainders obtained is always well-founded.

Where it doesn't work

Such proofs do not work when

- $fact$ 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

Well-founded function for gcd

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

Newton: Well-foundedness

Newton's Method

Convergence condition

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

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

Is the following definition an algorithm?

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

Unproven Claim. $collatz(m) \rightsquigarrow 1$ 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

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

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

$$\boxed{\mathcal{S}_{gcd(a,b)} \propto 1}$$

A similar analysis and result holds for $newton$

Factorial: Space

$factL$

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

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

$$\boxed{\mathcal{S}_{factL(n)} \propto n}.$$

A similar analysis and result holds for $factR$.

Fibonacci: Space

Fibonacci

When $n > 1$ there are two evaluations of F_2 with arguments $n-1$ and $n-2$ respectively. However assuming the second evaluation proceeds only after the first evaluation is done we may write the space requirement as

$$\mathcal{S}_{F_2(n)} = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ 1 + max(\mathcal{S}_{F_2(n-1)}, \mathcal{S}_{F_2(n-2)}) & \text{if } n > 1 \end{cases}$$

For large values of n and since we are only interested in orders of magnitude rather than exact space calculations we conclude that

$$\mathcal{S}_{F_2(n)} \leq 1 + \mathcal{S}_{F_2(n-1)} \propto n$$

7.2. 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

$\mathcal{S}_{isqrt(n)} = \mathcal{S}_{shrink(n,0,n)}$ for large n.

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

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

$$\boxed{\mathcal{S}_{isqrt(n)} = \mathcal{S}_{shrink(n,0,n)} \propto 1}$$

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 $\mathcal{T}_{shrink(n,l,u)} =$

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

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

$$\boxed{\mathcal{T}_{isqrt(n)} = \mathcal{T}_{shrink(n,0,n)} \propto n}$$

isqrt2: Time

shrink2

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

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

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

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

$$\boxed{\mathcal{T}_{isqrt2(n)} \propto \lceil \log_2 n \rceil}$$

$shrink$ 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.

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

Then

$$\boxed{\mathcal{T}_{factL(n)} \propto n}$$

Fibonacci: Time Complexity

Fibonacci

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

$$\mathcal{T}_{F_1(n)} = \begin{cases} 0 & \text{if } n \leq 1 \\ \mathcal{T}_{F_1(n-1)} + \mathcal{T}_{F_1(n-2)} & \text{if } n > 1 \end{cases}$$

It follows that

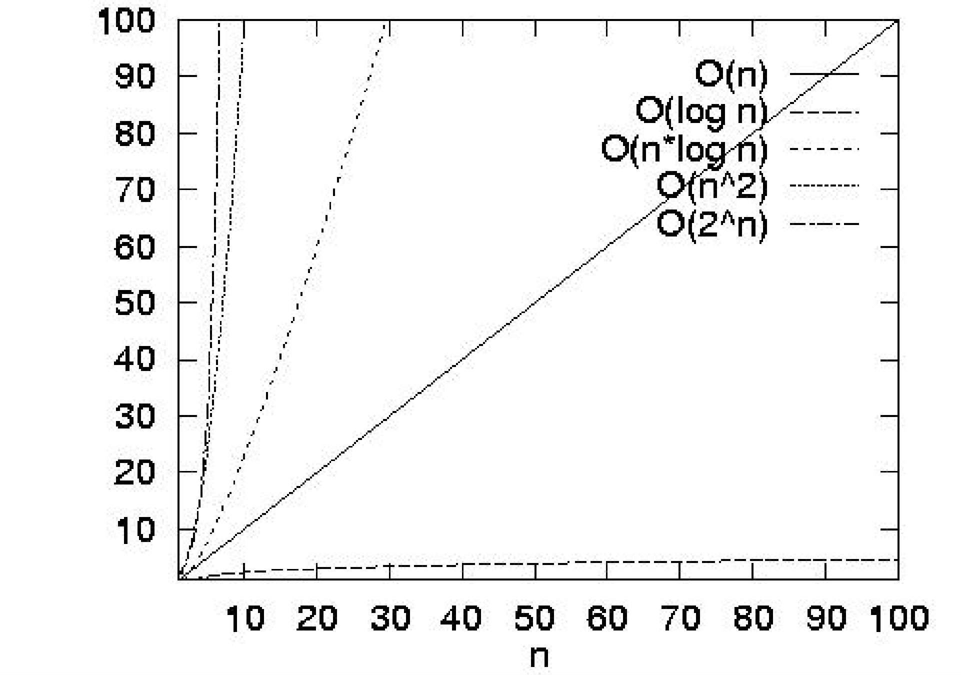
$$\boxed{2^{n-2} < \mathcal{T}_{F_1(n)} \leq 2^{n-1}}$$

Comparative Complexity

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

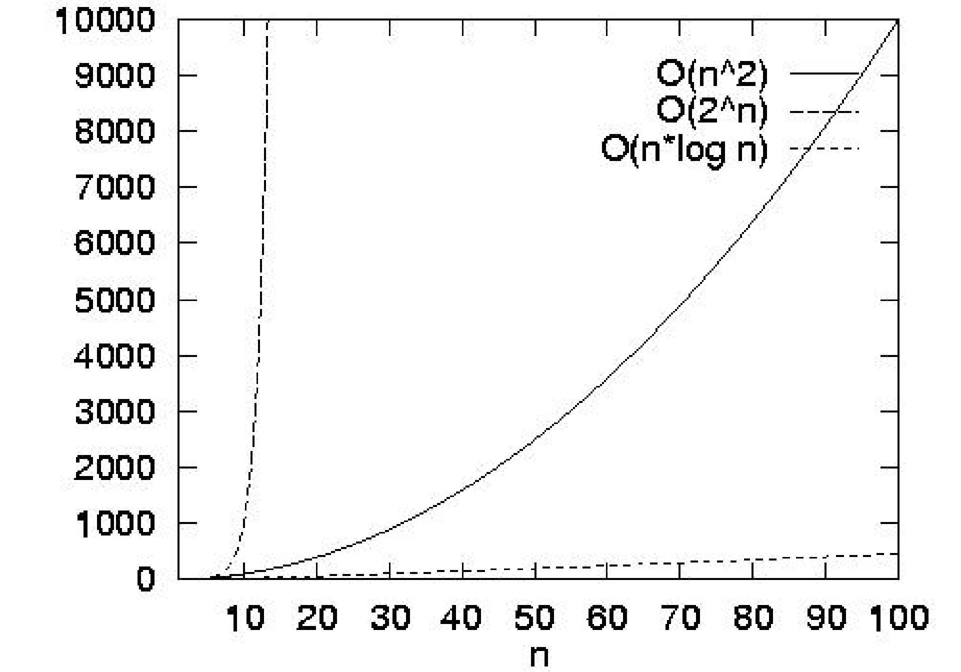
Comparisons

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, $newton$, $shrink2$

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

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

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 $\mathcal{S}_{factL2(n)} = O(1)$ (as against $\mathcal{S}_{factL(n)} \propto n$).
- Time complexity: Prove that $\mathcal{T}_{factL2(n)} = O(n)$

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 negativeArgument
    else if (n <= 1) then 1
    else fib_iter (n, 1, 1, 1)
end;
```

State in Tail Recursion

- The variables that make up the *state* bear a definite relation to each other.
- INVARIANCE. That relationship between the state variables does not change throughout the computation of the function.

Invariance

- The invariant property of a tail-recursive function must hold

  Initially when it is first invoked, and

  Continues to hold before every successive invocation
- The invariant property characterizes the entire computation and the algorithm and is crucial to the proof of correctness

7.3. 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 7.1 *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 7.2 *For all* $n \geq 0$ *and* p

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

Proof: By induction on n.

Back to theorem

Invariance: Example

$factL2$

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

Contrast with a $factL2(4)$ computation

Invariance: Example

$factL2$

So what exactly *is* invariant?

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

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

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

Back to lemma

Invariance Lemma: fib_iter

fib_iter $\mathbf{F}$

Lemma 7.3 *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 $k = n - m$

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

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

Correctness: Fibonacci

$fibTR$ Fibonacci

Theorem 7.4 *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 $b = 1 = \mathbf{F}(m)$.

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

Newton's Method Algorithm

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 $c = (a+b)/2$

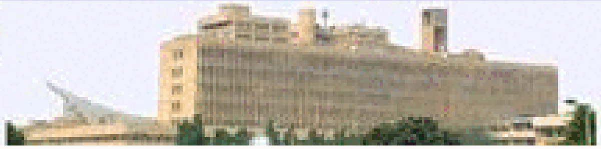
Advantage Bisection

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.

8. Compound Data

8.1. Tuples & Lists



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

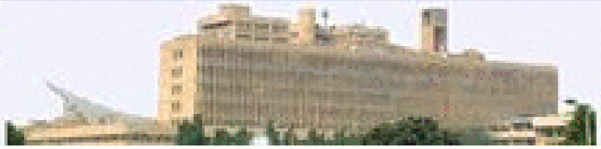
Every tuple that has been broken up into its components using its destructors can be put together back again using its constructors.
Given a tuple $\mathbf{a} \in A_1 \times A_2 \times \ldots \times A_n$, we have

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

Lists

An $\alpha\ list$ 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

- An empty list $[]$ can always be created. It is of type $\alpha\ list$ for any type α.
- A nonempty list of type $\alpha\ list$ may be created using the $cons$ (infix $::$) operation on any element of type α and a (nonempty or empty) list of type $\alpha\ list$.

Given a (nonempty) list L,

- A new list M may be created from an existing list L by the tl operation.
- 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

$$poly0(L,x) = \begin{cases} 0 & \text{if } L = nil \\ hx^n + poly0(T,x) & \text{if } L = h :: T \end{cases} \qquad (36)$$

where

$$n = |L| - 1$$

```
fun poly0 ([], x) = 0.0
  | poly0 ((h::T), x) =
    let val n = length L
    in h*power(x, n)+poly0(T, x)
    end
```

Complexity of $poly0$

Space. $O(n)$ to store both the list and the intermediate computations (unless a fresh list is created for each invocation of $poly0$ – in such a case it could be $O(n^2)$).

Additions. $O(n)$ additions. Also the computation of the length of the list in each invocation of $poly0$.

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

Multiplications. $O(\log_2(n)+O(\log_2(n-1)+\cdots+O(\log_2(1)) = O(\log_2(n!))$ by the fast powering algorithm. However by Stirling's approximation[a] we have

$$\ln(n!) = n\ln(n) - n + O(\ln(n)) = O(n\log_2(n))$$

We could save on some of these computations using Horner's rule.

[a] see Wikipedia http://en.wikipedia.org/wiki/Stirling_approximation

Horner'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.

$$poly1(L, x) = poly\_tr(0, L, x) \tag{37}$$

where

$$poly\_tr(p, L, x) = \begin{cases} p & \text{if } L = nil \\ poly\_tr(px + h, T, x) & \text{if } L = h :: T \end{cases} \tag{38}$$

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. $poly\_tr(p, L, x) = px^{n+1} + \sum_{i=0}^{n} a_i x^i$

Complexity of $poly1$: Savings from $poly0$

Space. $O(n)$ to store both the list and the intermediate computations (unless a fresh list is created for each invocation of $poly0$ – in such a case it could be $O(n^2)$).

Additions. $O(n)$ additions. Also the length of the list need not be computed.

Multiplications. $O(n)$ multiplications. This is the biggest saving in this algorithm.

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.

*revpoly*0

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

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

Tail Recursive

We transform $revpoly0$ in a completely standard fashion to obtain a tail-recursive version.

$$revpoly1(L, x) = revpoly1\_tr(L, x, 1, 0) \tag{39}$$

where

$$revpoly1\_tr(L, x, p, s) = \begin{cases} s & \text{if } L = nil \\ revpoly1(T, x, px, s + ph) & \text{if } L = h :: T \end{cases} \tag{40}$$

Here p represents the value of x^i for each i, $0 \leq i \leq n$.
Question 2. What is the invariant for this tail-recursive version?

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 8.1 *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

- Every number may be expressed uniquely (upto order) as a product of prime factors.
- No divisor of a positive integer can be greater than itself.
- For each divisor $d|n$ such that $d \leq \lfloor\sqrt{n}\rfloor$, $n/d \geq \lfloor\sqrt{n}\rfloor$ is also a divisor.

Composites

- If a number n is composite, then it has a proper divisor d, $2 \leq d \leq \lfloor\sqrt{n}\rfloor$.
- If a number n is composite, then it has a prime divisor p, $2 \leq p \leq \lfloor\sqrt{n}\rfloor$.
- An odd number cannot have any even divisors.
- 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 $\leq \lfloor\sqrt{n}\rfloor$.

$$primesUpto(n)$$

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

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

- P is a list of ordered pairs (i, p_i) containing the first $k-1$ primes,
- k denotes the index of the next prime to be found,
- $m > p_{k-1}$ is the next number to be tried as a possible candidate for the k-th prime,
- $m \leq n$ is odd and
- there are no primes in the interval $[p_{k-1}+2, \ldots, m-2]$

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

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 8.2 *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

$$primeWRT$$

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

where

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

for some $i > 0$.

```
(* Program for generating primes upto some number *)
local
    fun primeWRT (m, []) = true
      | primeWRT (m, (_, h)::t) =
        if m mod h = 0 then false
        else primeWRT (m, t)

    fun generateFrom (P, m, n, k) =
        if m > n then P
        else if primeWRT (m, P)
        then ( print (Int.toString (m)^" is a prime\n");
               generateFrom (((k, m)::P), m+2, n, k+1)
             )
        else generateFrom (P, m+2, n, k)
in
fun primesUpto n =
    if n < 2 then []
    else if n=2 then [(1,2)]
    else if (n mod 2 = 0) then primesUpto (n-1)
    else generateFrom ([(1,2)], 3, n, 2);
end
```

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

8.4. 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

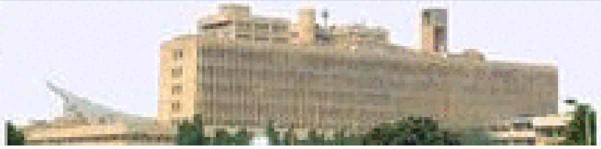
```
- #2 a;
val it = true : bool
- #1 a;
val it = "arun" : string
- #3 a;
val it = 2 : int
- #4 a;
stdIn:1.1-1.5 Error: operator and operand don’t agree [r
  operator domain: {4:’Y; ’Z}
  operand:         string * bool * int
  in expression:
    (fn {4=4,...} => 4) a
```

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 operand don't agree
  operator domain: (string * (int * bool)) * (string * (
  operand:         (string * (int * bool)) * (string * i
  in expression:
    ("arun",(1,true))
  = ("arun",1,true)
- ("arun", (1, true))
= (("arun", 1), true);
stdIn:1.1-29.39 Error: operator and operand don't agree
```

Lists: Recap

formation : Sequence $\alpha\ List$

selection : Selection of individual components

new lists : Making new lists from old

The Append Operation:

Unlike $cons$ which prepends an element to a list, the infix append binary operation, @, on lists appends a list to another list. i.e. if $L = [a_0, \cdots, a_{l-1}]$ and $M = [b_0, \cdots, b_{m-1}]$ then $L@M = [a_0, \cdots, a_{l-1}, b_0 \cdots, b_{m-1}]$.
The append operation satisfies the following properties:

$$\begin{array}{lcl} []@M & = & M \\ L@[] & = & L \\ (L@M)@N & = & L@(M@N) \end{array}$$

where L, M and N are three lists of the same type of elements. Further if $|L|$ denotes the length of a list then

$$|L@M| = |L| + |M|$$

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
```

$::$ vs. $@$

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

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

$$\mathcal{T}_{L@M} = \begin{cases} 0 & \text{if } L = nil \\ 1 + \mathcal{T}_{T@M} & \text{if } L = h :: T \end{cases}$$

Lists of Functions

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

Lists of Functions

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

Arithmetic Sequences

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

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

where $L_{n-1} = AS1(a,d,n-1)$

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

Tail Recursion

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

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

$$\boxed{n-k}$$

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

Tail Recursion: Complexity

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

So this tail recursion simply doesn't help!

Another Tail Recursion : $AS3$

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

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

$$\boxed{INV3 : L = (a+(n-1)*d) :: \cdots :: (a+(n_0-1)*d) :: L_0}$$

Another Tail Recursion: $AS3\_tr$

$$\boxed{INV3 : L = (a + (n - 1) * d) :: \cdots :: (a + (n_0 - 1) * d) :: L_0}$$

and bound function n,

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

AS3: Complexity

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

Generating Primes: Recap

- $primesUpto$
- $generateFrom$ and its invariant
- $primeWRT(m, P)$

Note that

1. $primeWRT(m, P)$ is evaluated for every candidate against the currently generated primes
2. $primeWRT(m, P)$ checks divisibility against every prime generated.
3. It suffices to check only against the primes that are smaller than $\sqrt{m}$ for any odd number m, by properties of composites.

Generating Primes: Version 2

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

where the only difference between $primes2Upto(n)$ and $primesUpto(n)$ is the evalution of $generate2From$ instead of $generateFrom$.

Invariant & Bound Function: Version 2

invariant

bound function $\boxed{n - m}$

Invariant2

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

implies

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

and

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

i.e. the list P of primes is in ascending order.

$$generate2From$$

$generateFrom$

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

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 $p_j \leq \lfloor\sqrt{m}\rfloor < p_{j+1}$

$$prime2WRT$$

$$prime2WRT(m, P) = \begin{cases} true & \text{if } P = nil \\ true & \text{if } h > m \texttt{ div } h \\ false & \text{elseif } h|m \\ prime2WRT(m, tl(P)) & \text{else} \end{cases}$$

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 $AS3$)?

```
(* This is really the same as primes.sml but uses the fact that it suffices
   to check divisibility only with respect to all the primes less than the
   square-root of the candidate number. As a result,

   1. the primes have to be placed in ascending order in the list P
   2. But this implies that the new prime generated is appended to the list
      of primes rather than prefixed to it.

   See the slide ”Generating Primes: Recap”
*)

local
     fun prime2WRT (m, []) = true
       | prime2WRT (m, (_, h)::t) =
          if h > m div h (* h*h > m *) then true
          else if (m mod h = 0) then false
          else prime2WRT (m, t)
     fun generate2From (P, m, n, k) =
          if m > n then P
          else if prime2WRT (m, P)
          then ( print (Int.toString (m)^" is a prime\n");
                 generate2From ((P@[(k, m)]), m+2, n, k+1)
```

```
             )
        else generate2From (P, m+2, n, k)
in
fun primes2Upto n =
    if n < 2 then []
    else if n=2 then [(1,2)]
    else if (n mod 2 = 0) then primes2Upto (n-1)
    else generate2From ([(1,2)], 3, n, 2);

end
```

8.5. Compound Data & List Algorithms

Compound Data: Summary

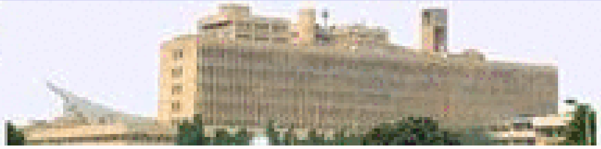
- Compound Data:

  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



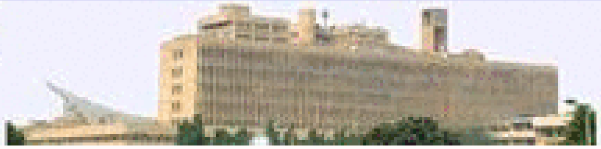
Records: Constructors

- 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

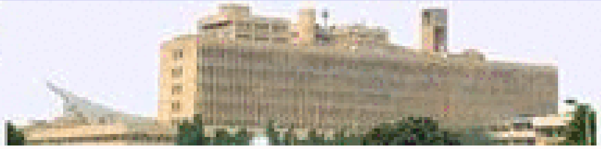
  types of fieldnames

  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:string}
- 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:string}
- pinky = pinky2;
val it = true : bool
```

Tuples & Records

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

back

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 ($\mathscr{P}$) are proved by induction on the length of the list.

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

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

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

Property $\mathscr{P}$ is proved by <u>case analysis</u>.

Basis Prove

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

Induction hypothesis (IH) Assume

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

Induction Step Prove

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

List-functions: $length$

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

List Functions: $reverse$

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

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

Theorem 8.3 *For any list* $L = [a_0, \cdots, a_{l-1}]$*,* $l \geq 0$*,*

$$reverse(L) = [a_{l-1}, \ldots, a_0]$$

Proof: By induction on l.
Time Complexity?? $O(n^2)$

List Functions: $reverse2$

$$\begin{cases} reverse2\ [] & = [] \\ reverse2\ (h :: T) & = rev\ ((h :: T), []) \end{cases}$$

where

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

Lemma 8.4 *For any list* N *and list* $L = [a_0, \cdots, a_{l-1}], l \geq 0,$

$$rev(L, N) = a_{l-1} :: \cdots :: a_0 :: N$$

Proof: By induction on l.

Corollary 8.5 *For any list* $L = [a_0, \cdots, a_{l-1}],$

$$reverse2(L) = [a_{l-1}, \ldots, a_0]$$

Proof: Follows from lemma **??** with $N = []$.
Time Complexity: $O(n)$

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

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

Time Complexity??

Total Orderings

Definition 8.6 *Given a set S and a binary relation $\mathcal{R}$ on S, i.e. $\mathcal{R} \subseteq S \times S$,*

Irreflexivity. *$\mathcal{R}$ is* **irreflexive** *if $(x, x) \notin \mathcal{R}$ for any $x \in S$.*

Transitivity. *$\mathcal{R}$ is* **transitive** *if for all $x, y, z \in S$, $(x, y) \in \mathcal{R}$ and $(y, z) \in \mathcal{R}$ implies $(x, z) \in \mathcal{R}$.*

Total Ordering. *$\mathcal{R}$ is a* **total ordering** *on S if it is an irreflexive and transitive relation such that for any two elements $x, y \in S$, $x \neq y$ implies either $(x, y) \in \mathcal{R}$ or $(y, x) \in \mathcal{R}$.*

- Often binary relations are used in *infix* form i.e. $(x, y) \in \mathcal{R}$ is usually written as $x\mathcal{R}y$.

The simplest examples of irreflexive total orderings are the usual “less-than ($<$)” and the “greater-than” ($>$) relations on numbers. The ordering of words in a dictionary is another example of an irreflexive total ordering. Certain other irreflexive relations such as the “proper subset ($\subset$)” ordering on sets are not total since they do not obey the trichotomy law.

A basic fact underlying these irreflexive total orders is the following “trichotomy law”.

Fact 8.7 *If $<$ is an irreflexive total ordering on S then for any $x, y \in S$ exactly one of the following holds.*

$$x < y \qquad x = y \qquad y < x$$

Definition 8.8 *Let $\mathcal{R} \subseteq S \times S$ be any binary relation on S. The* **converse** *of $\mathcal{R}$ denoted $\mathcal{R}^{-1}$ is defined as*

$$\mathcal{R}^{-1} = \{(y, x) \mid (x, y) \in \mathcal{R}\}$$

From the fact below it easily follows that the converse of any irreflexive total order is also an irreflexive total order. The “$>$” relation on numbers is the converse of the “$<$” relation and vice-versa.

Fact 8.9

1. *For any relation $\mathcal{R} \subseteq S \times S$, $\mathcal{R} = \mathcal{R}^{-1^{-1}}$.*
2. *The converse of an irreflexive relation is also irreflexive.*
3. *If an irreflexive relation is a total order then so is its converse.*

In the above examples we have emphasised the irreflexive nature of these orderings. This is to distinguish them from the reflexive orderings such as the “less-than-or-equal-to ($\leq$)” and the “greater-than-or-equal-to ($\geq$)” orderings which are reflexive. In fact they are obtained from the corresponding irreflexive relations by a set union with the smallest reflexive relation on sets. See the definition below.

Definition 8.10 *Let $\mathcal{I}_S = \{(x, x) \mid x \in S\}$ be the* **identity relation** *on S. Then for any irreflexive total ordering $<$ on S we define its* **reflexive closure** *as the relation $\leq \; = \; < \cup \mathcal{I}_S$.*

List Functions: $ordered_{\leq}$

Definition 8.11 *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 $0 \leq i < n-1$, where $<$ is a total ordering on the type of elements that make up the list.*

We may define this property of lists *inductively* on the structure of lists as follows:

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

Time Complexity??

List Functions:$insert$

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

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

Theorem 8.12 *For any list L of elements drawn from a set with a total ordering $<$ and an element x from the same set, $ordered_{\leq}(L)$ implies $ordered_{\leq}(insert(x, L))$.*

Proof: By induction on the structure of the ordered list L.
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:$merge$

$merge$ may be defined inductively as

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

where $L = a :: S$ and $M = b :: T$.

Theorem 8.13 *Let L, M, N be lists consisting of elements from a set S totally ordered by a relation $<$ such that $ordered_{\leq}(L)$, $ordered_{\leq}(M)$ and $N = merge(L, M)$. Then*

1. *$ordered_{\leq}(N)$ and*
2. *for each $a \in S$, $\#a(N) = \#a(L) + \#a(M)$, where $\#a(L)$ denotes the number of occurrences of a in L.*

Proof: ??

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)
```

Time Complexity??

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.

9. Higher Order Functions & Structured Data

9.1. 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 :

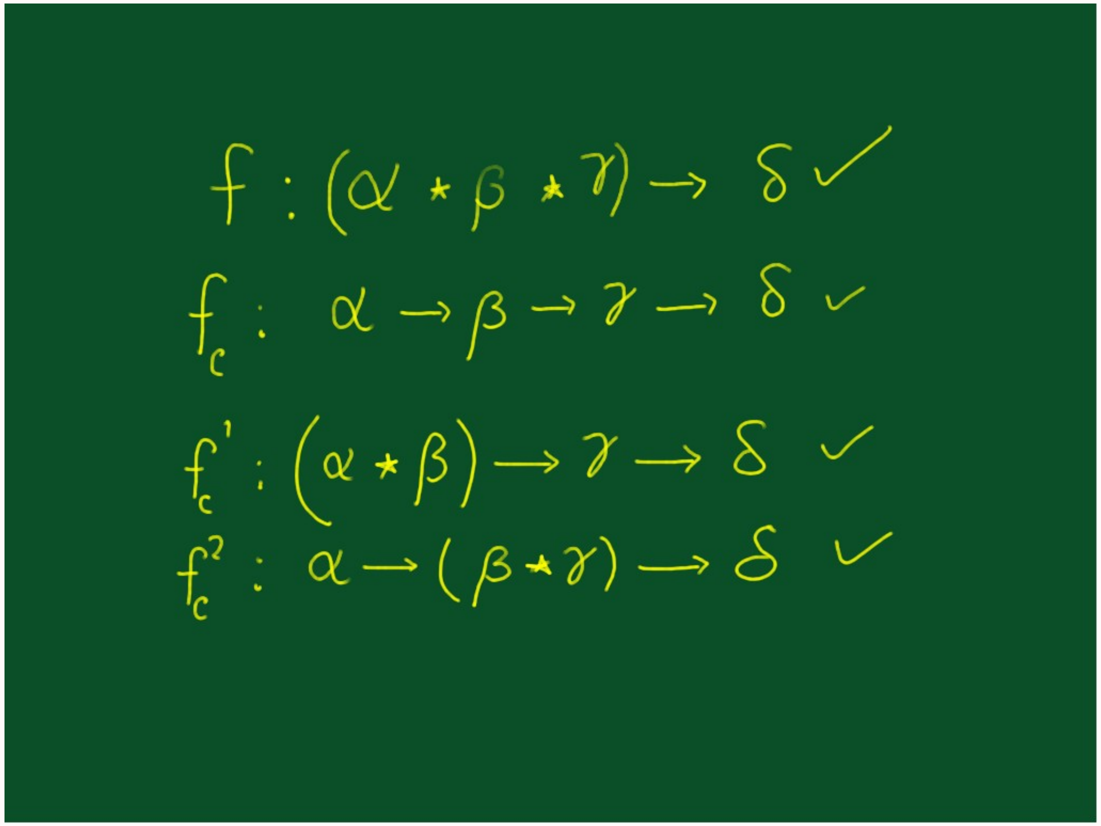
```
val addc = fn :
       int -> (int -> int)
```

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

Apply a list Transpose of a matrix

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

```
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

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

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

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 $\bigotimes$

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

- $\otimes : (\alpha * \alpha) \to \alpha$ 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 $f, succ : \alpha \to \alpha$ $ser(\otimes, f, succ, l, u) =$

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

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

9.2. 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 ]       |    [11   21    31   41 ]
  [        22       23 ]       |
  [        32       33 ]       |
  [        42       43 ]       |
]                              |  ]
```

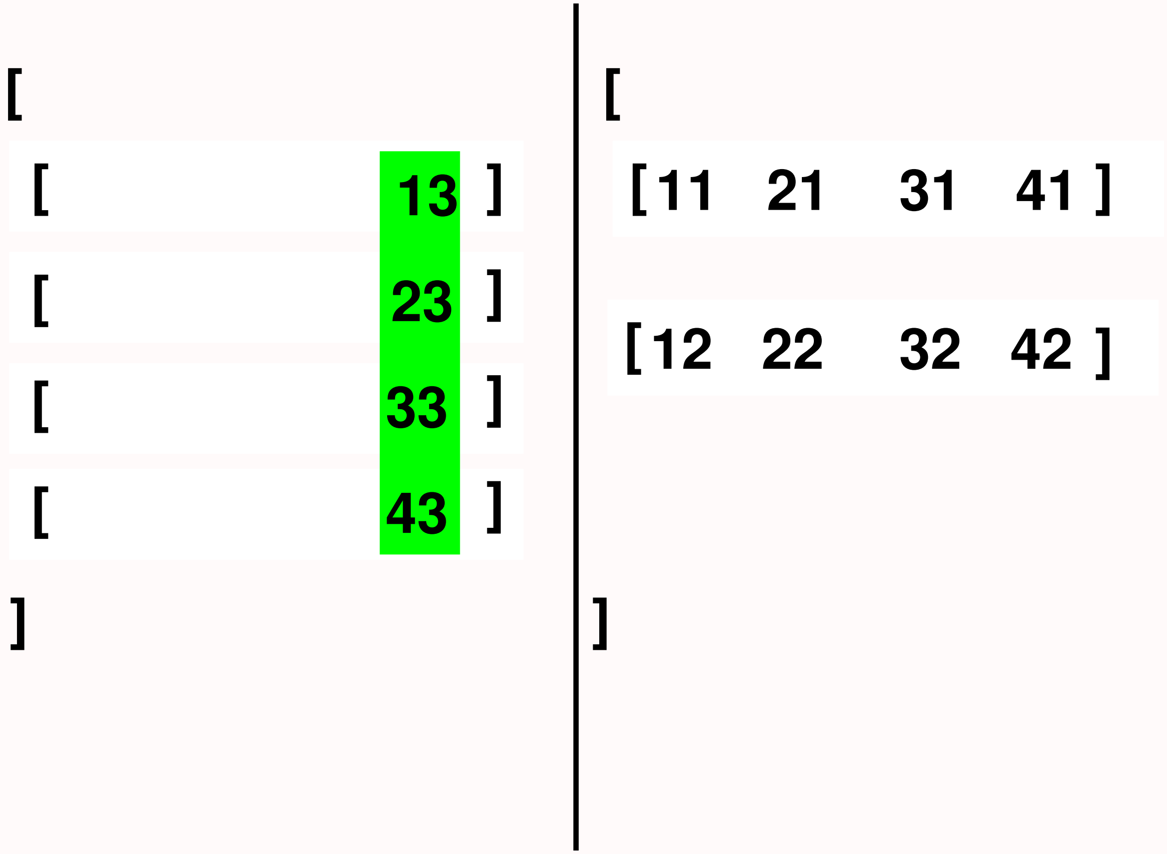
Transpose: 20

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

Transpose: 02

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

Transpose: 30



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

Transpose: 03

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

$$trans$$

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

and

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

$$is2DMatrix$$

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

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

Sorting

We have already seen two sorting algorithms on integers

- insertion sort
- merge sort

Generalized Sorting

It is possible to generalize these algorithms so that they are both

higher order . The ordering relation is an argument to the sort function.

polymorphic . They work on *any* data type on which a *reflexive total order* may be defined and

In particular *ascending* order and *descending* order may be obtained as the total order and its converse (e.g. the "$\geq$" relation is the converse of the "$\leq$" relation).

```
(* It is assumed that R is a binary total ordering. Otherwise there is no
   guarantee that these programs will work correctly.
*)

(*———————————————— INSERTION SORT ————————————————*)
fun insertSort R [] = []
  | insertSort R (h::t) =
        let fun insert R [] x = [x]
              | insert R (h::t) x = if R (x, h) then x::(h::t)
                                    else h::(insert R t x)
            val rest = insertSort R t
        in  insert R rest h
        end;

(* Test
val i = insertSort;
i (op <) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
i (op <=) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
i (op >) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
i (op >=) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
*)
```

```
(* ———————————————— MERGE SORT ———————————————————————— *)
fun mergeSort R [] = []
  | mergeSort R [h] = [h]
  | mergeSort R L = (* can't split a list unless it has > 1 element *)
       let fun split [] = ([], [])
             | split [h] = ([h], [])
             | split (h1::h2::t) =
                   let val (left, right) = split t;
                    in (h1::left, h2::right)
                   end;
           val (left, right) = split L;
           fun merge (R, [], []) = []
             | merge (R, [], (L2 as h2::t2)) = L2
             | merge (R, (L1 as h1::t1), []) = L1
             | merge (R, (L1 as h1::t1), (L2 as h2::t2)) =
                      if R(h1, h2) then h1::(merge (R, t1, L2))
                      else h2::(merge(R, L1, t2));
           val sortedLeft = mergeSort R left;
           val sortedRight = mergeSort R right;
       in  merge (R, sortedLeft, sortedRight)
       end;
```

```
(* Test
val m = mergeSort;
m (op <) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
m (op <=) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
m (op >) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
m (op >=) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
*)
```

Sorting: Polymorphism

The same sort functions may be used to sort strings in "*ASCIIbetical*" order under a lexicographic ordering. The function which defines the lexicographic ordering on strings is shown.

```
(* Lexicographic ordering on strings *)
fun lexlt (s, t) =
    let val Ls = explode (s);
        val Lt = explode (t);
        fun lstlexlt (_, []) = false
          | lstlexlt ([], (b:char)::M) = true
          | lstlexlt (a::L, b::M) =
                    if (a < b) then true
                    else if (a = b) then lstlexlt (L, M)
                         else false
     in lstlexlt (Ls, Lt)
     end
fun lexleq (s, t) = (s = t) orelse lexlt (s, t)
fun lexgt (s, t) = lexlt(t, s)
fun lexgeq (s, t) = (s = t) orelse lexgt (s, t)
(*
- val m = mergeSort;
val m = fn : ('a * 'a -> bool) -> 'a list -> 'a list
- m lexlt ["katrina", "SONAKSHI", "Katrina", "sonakshi", "Sonakshi", "KATRINA"];
val it = ["KATRINA","Katrina","SONAKSHI","Sonakshi","katrina","sonakshi"]
  : string list
- *)
```

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 not the same 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 $\alpha\ list$ 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

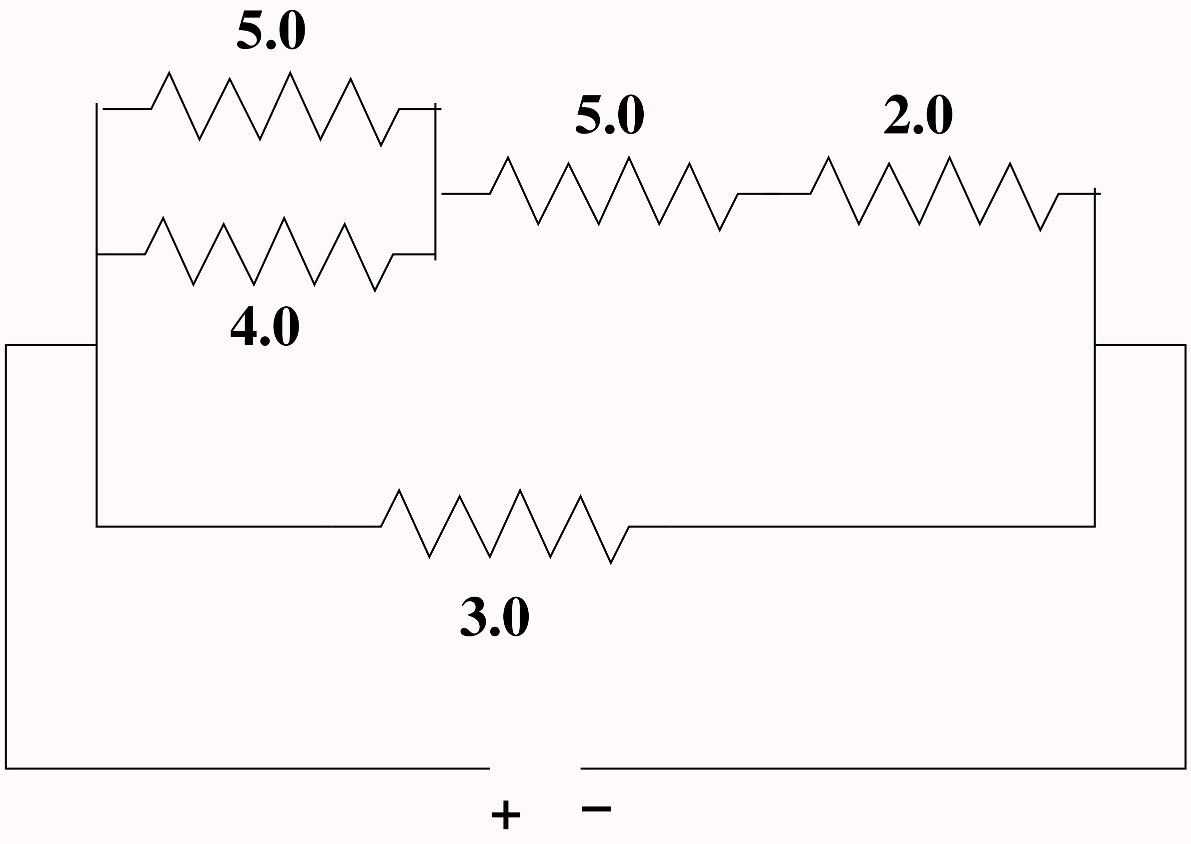
Resistors: Datatype

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

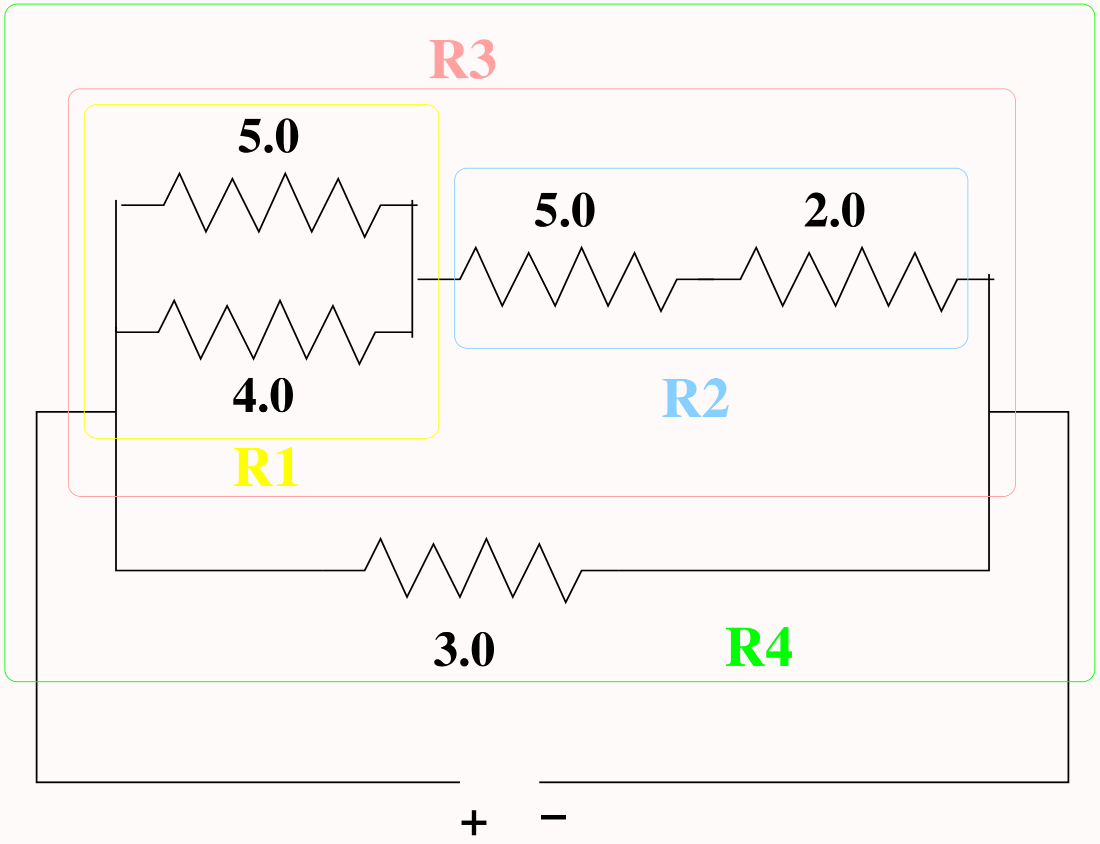
Resistance Expressions

9.4. 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

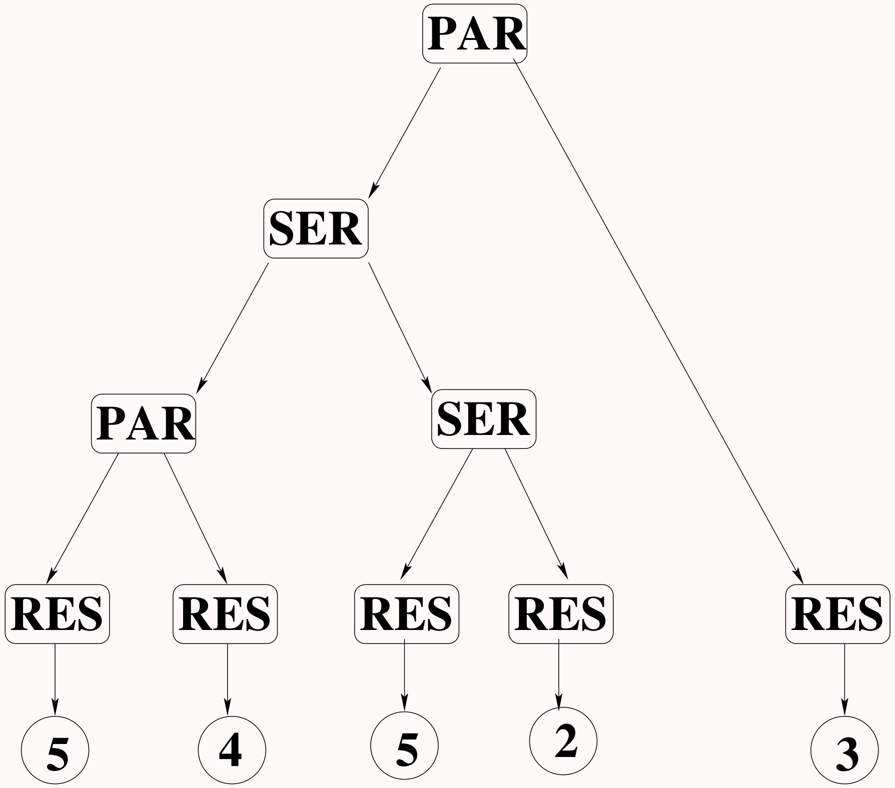
```
val R1 = PAR (RES 5.0,RES 4.0) : resist
val R2 = SER (RES 5.0,RES 2.0) : resist
val R3 = SER (R1, R2);
val R4 = PAR (R3, RES(3.0));
```

Resistor: Values

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

Resistance Expressions

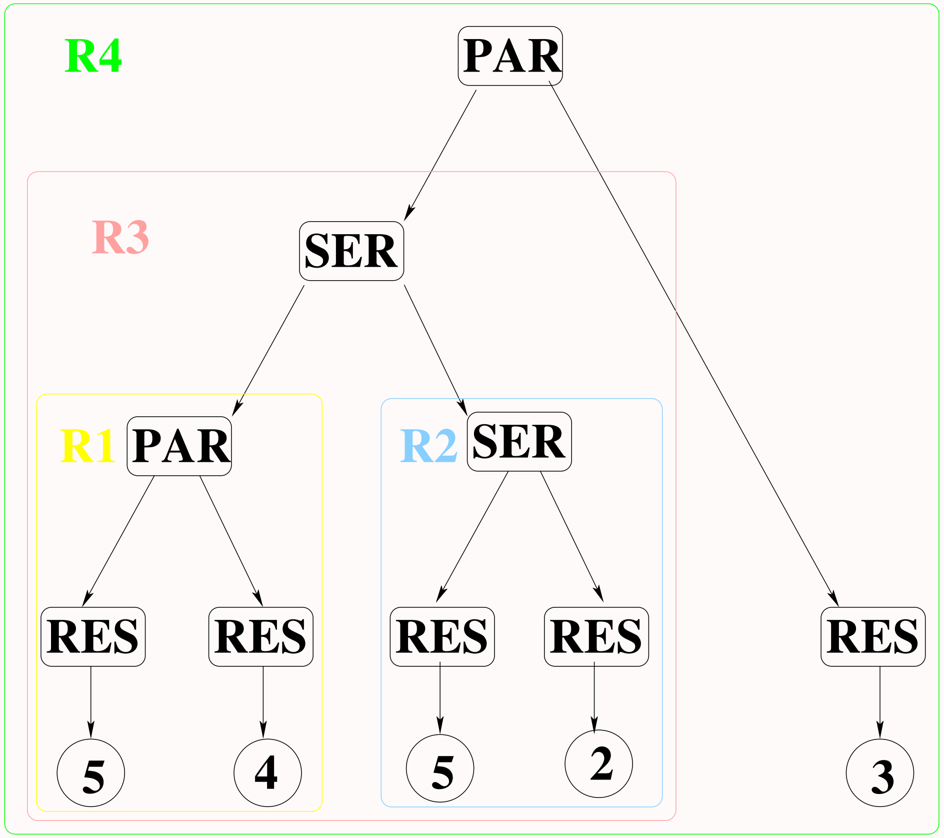
A resistance expression



Circuit Diagram

Resistance Expressions

A resistance expression



Circuit Diagram

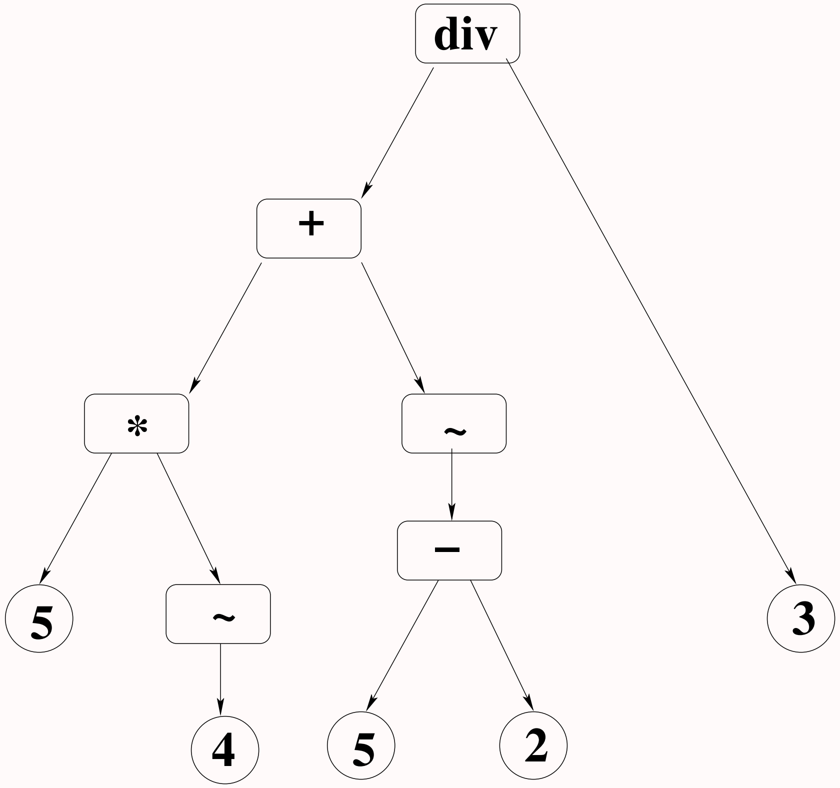
Arithmetic Expressions

ML arithmetic expressions:

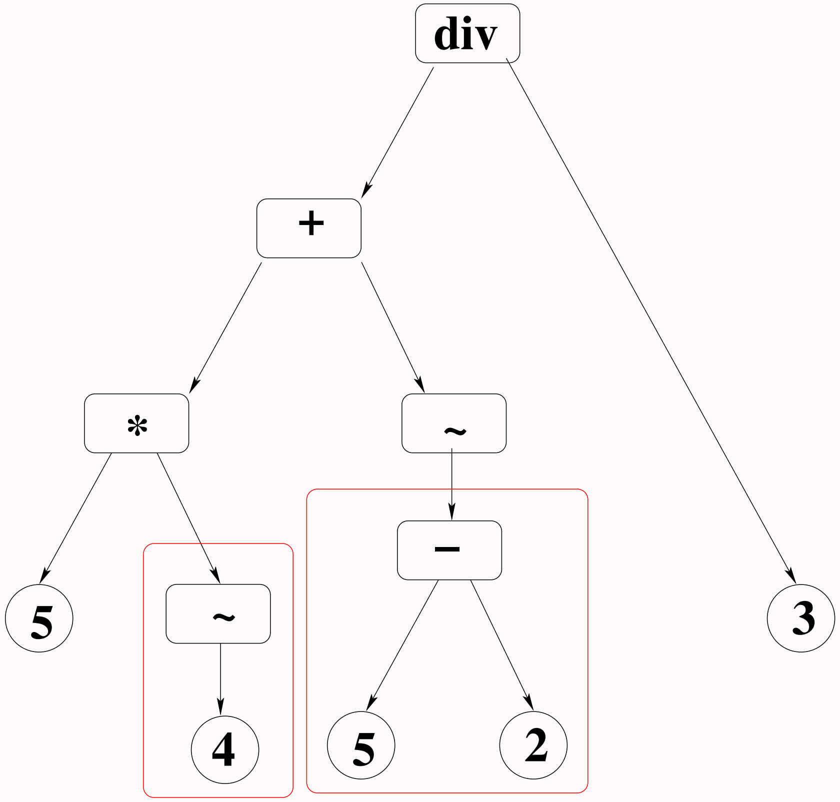
```
  ((5 * ~4) + ~(5 - 2)) div 3
```

are represented as trees

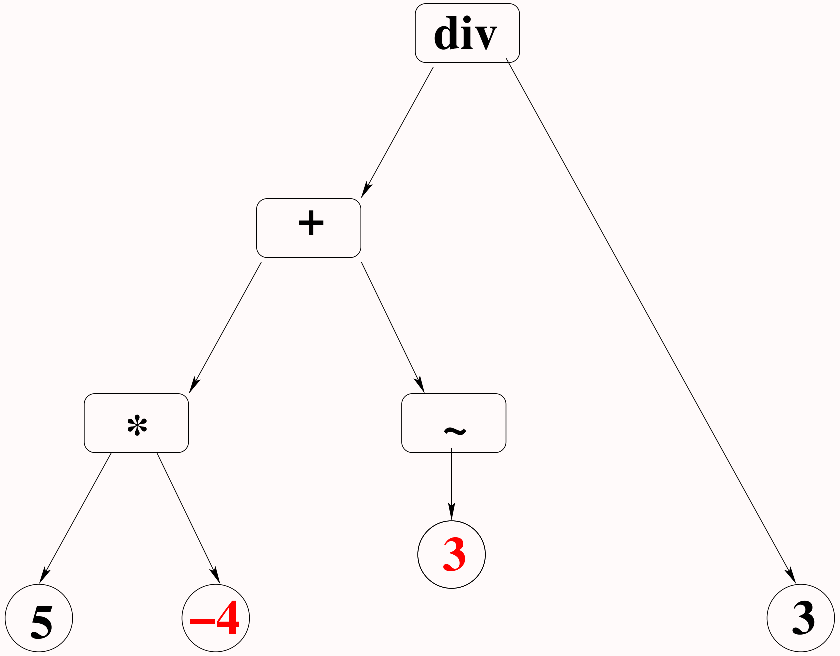
Arithmetic Expressions: 0



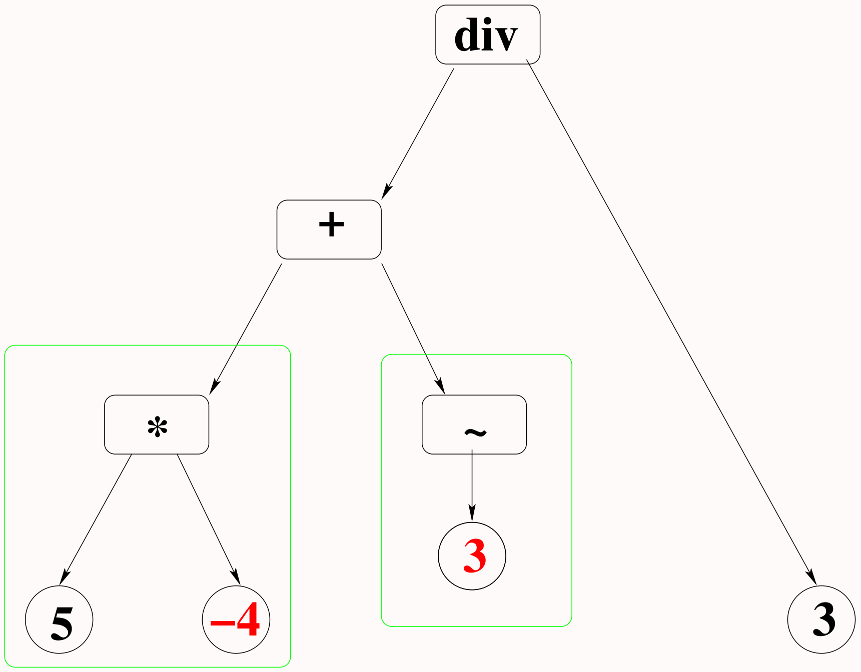
Arithmetic Expressions: 1



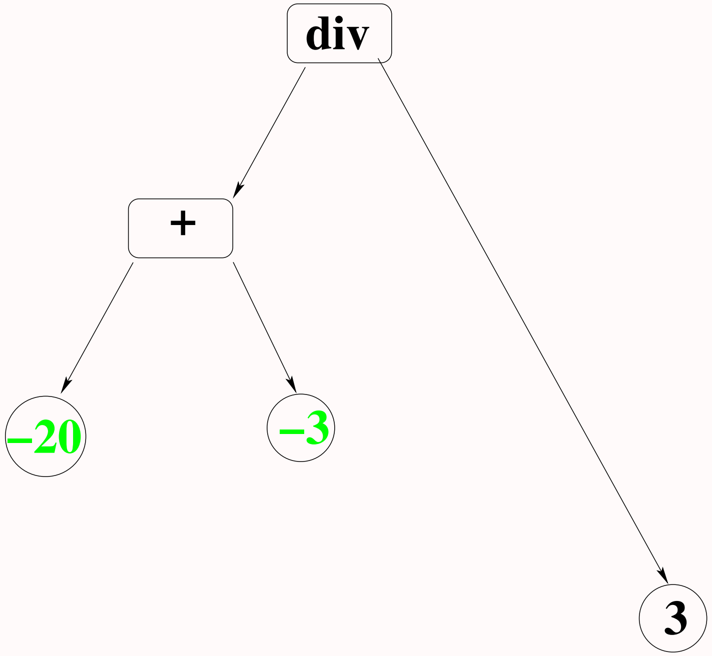
Arithmetic Expressions: 2



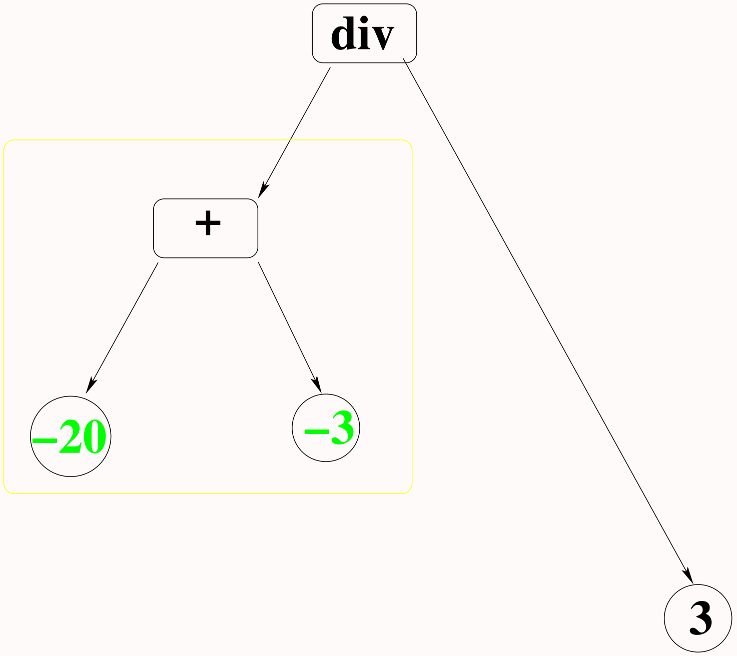
Arithmetic Expressions: 3



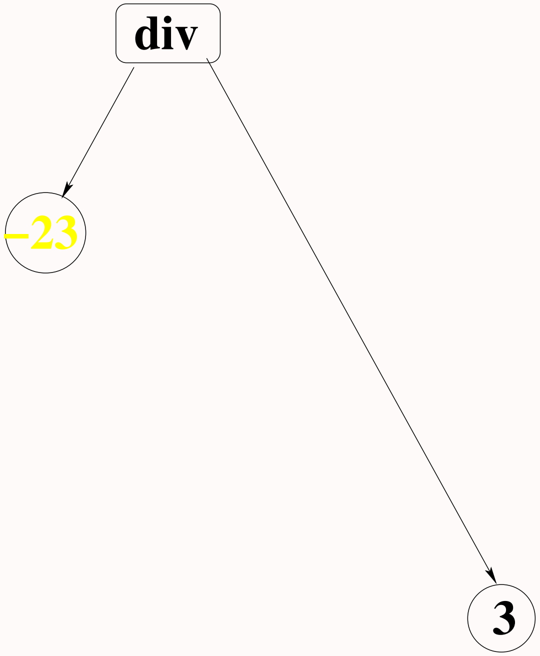
Arithmetic Expressions: 4



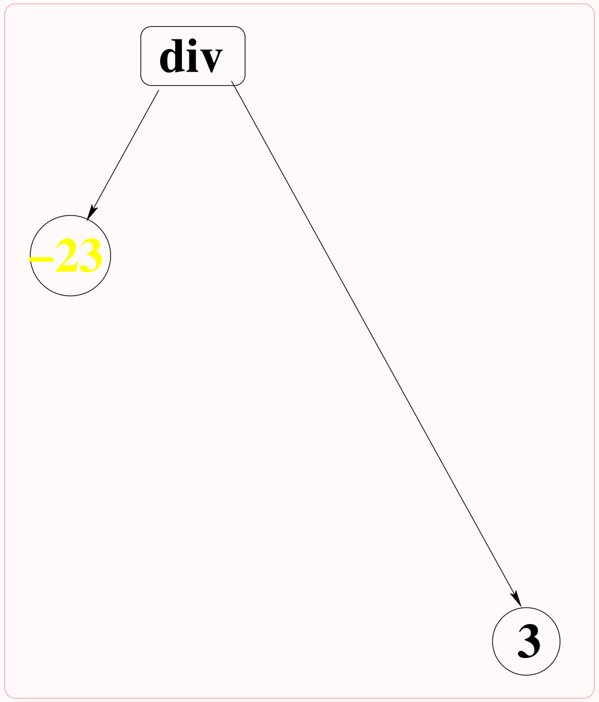
Arithmetic Expressions: 5



Arithmetic Expressions: 6



Arithmetic Expressions: 7



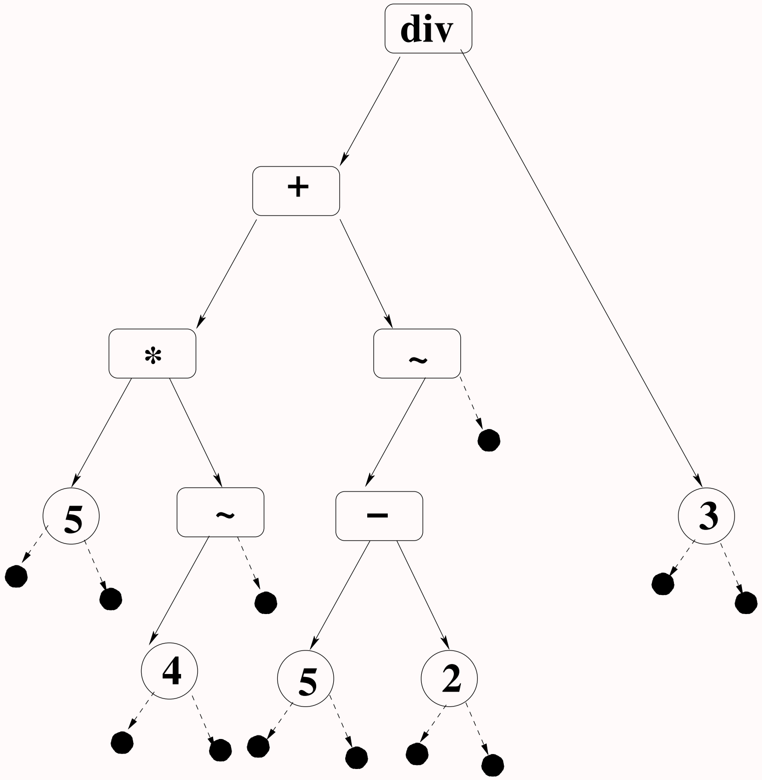
Arithmetic Expressions: 8

−8

Binary Trees

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

Arithmetic Expressions: 0



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

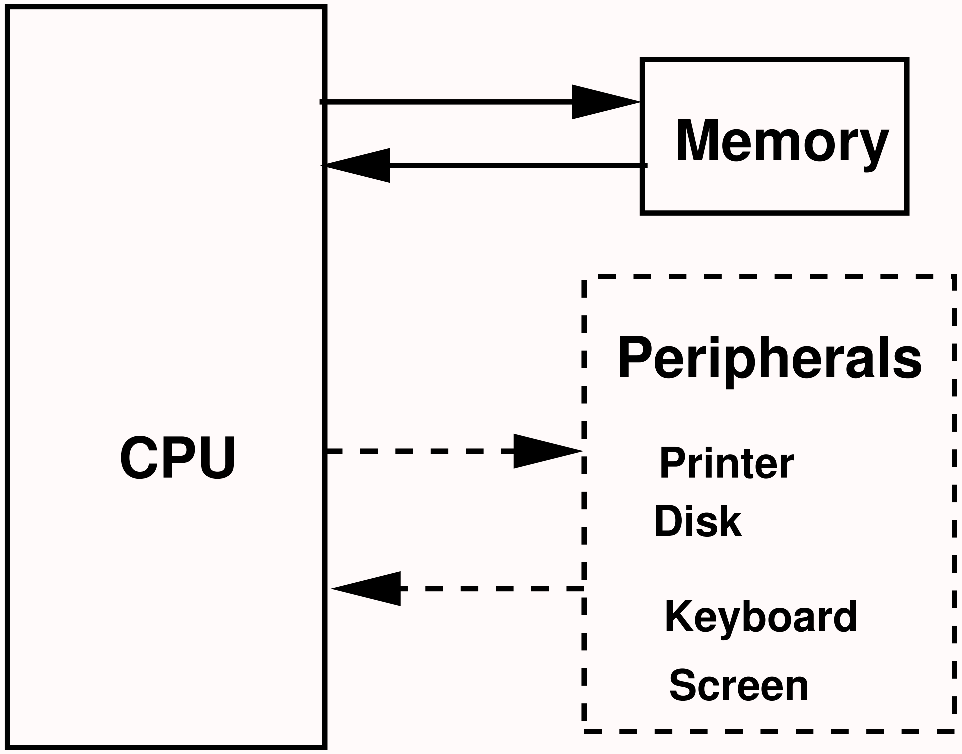
10. Imperative Programming: An Introduction

10.1. Introducing a Memory Model

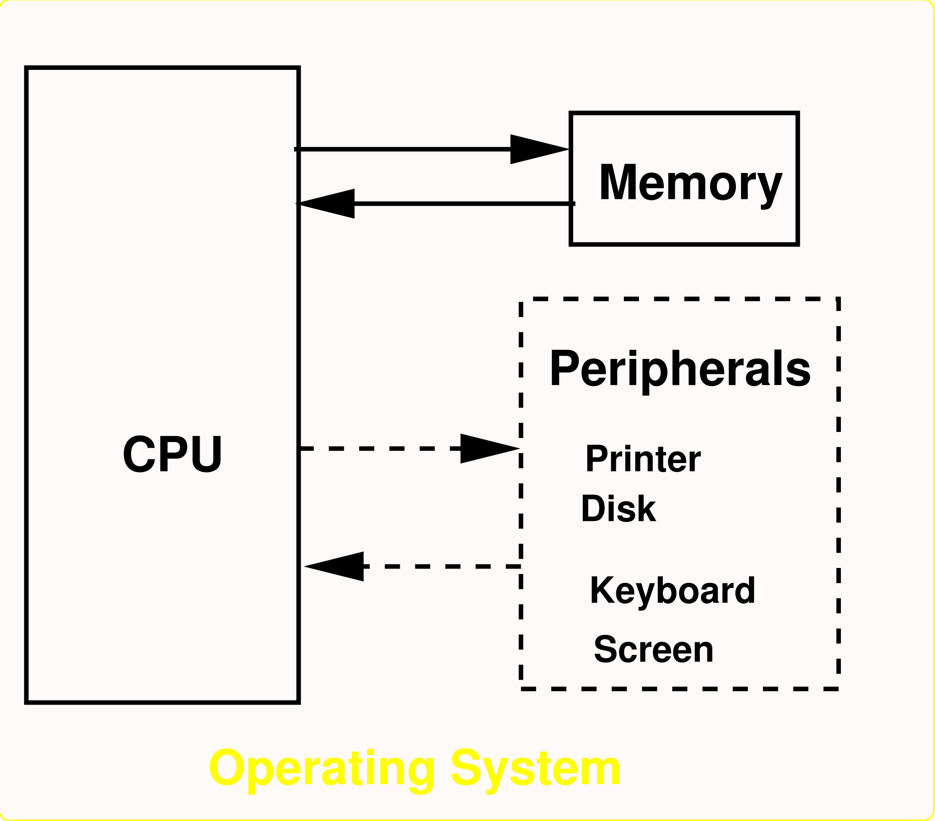
Summary: Functional Model

- 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



CPU

Memory

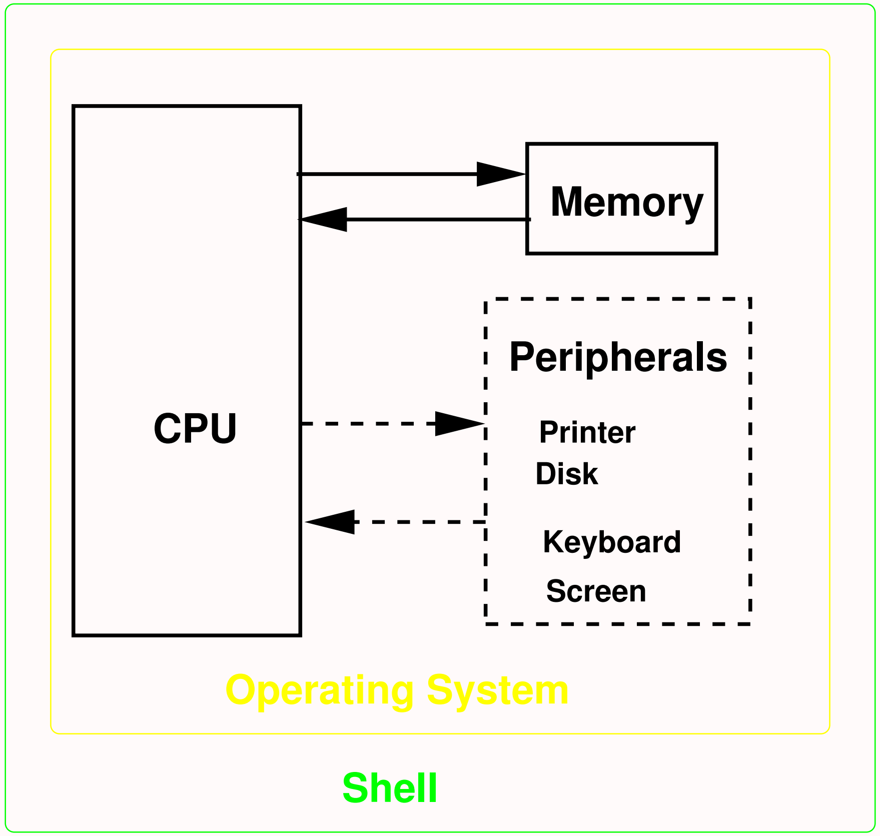
Peripherals

Printer
Disk

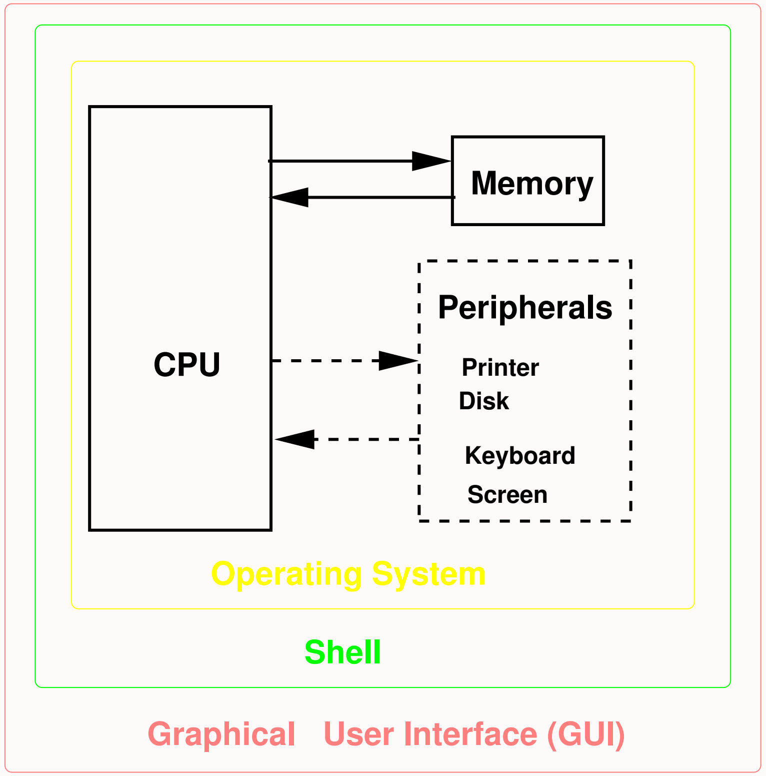
Keyboard
Screen

Operating System

Shell: User Interface



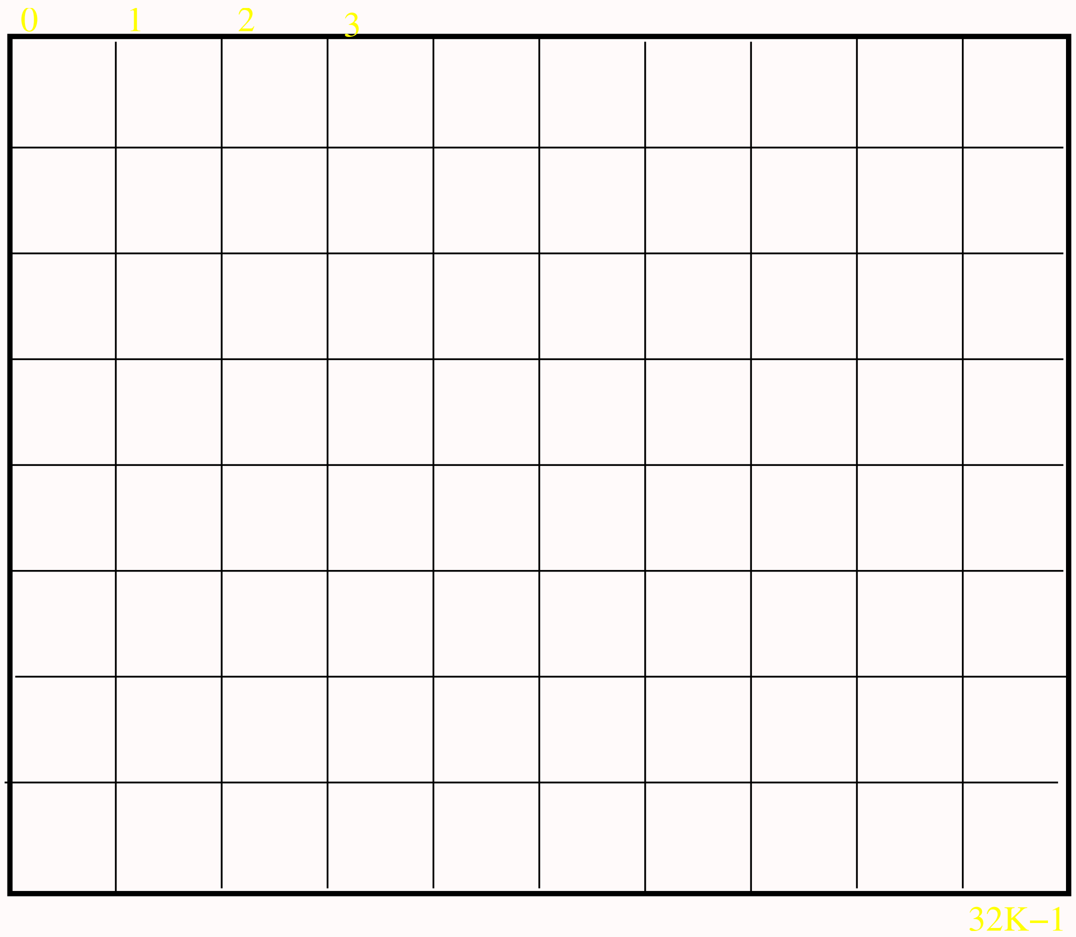
GUI: User Interface



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

0	1	2	3						
		4							
						3.1			
			true						
						"#a"			

Assume all other cells are filled with null

State

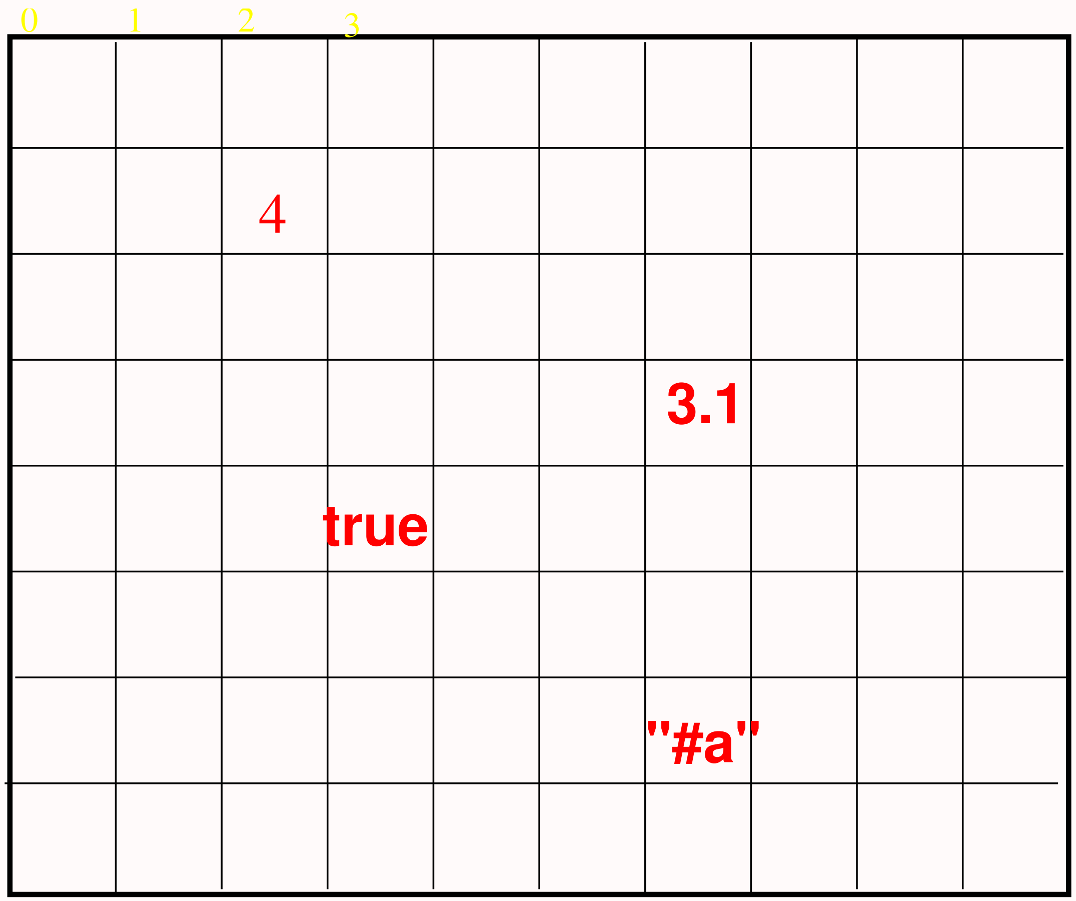
The state σ

- $\sigma(12) = 4 : int$
- $\sigma(20) =$ null
- $\sigma(36) = 3.1 : real$
- $\sigma(43) =$ 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

0	1	2	3						
		5							
						3.1			
			true						
						"#a"			

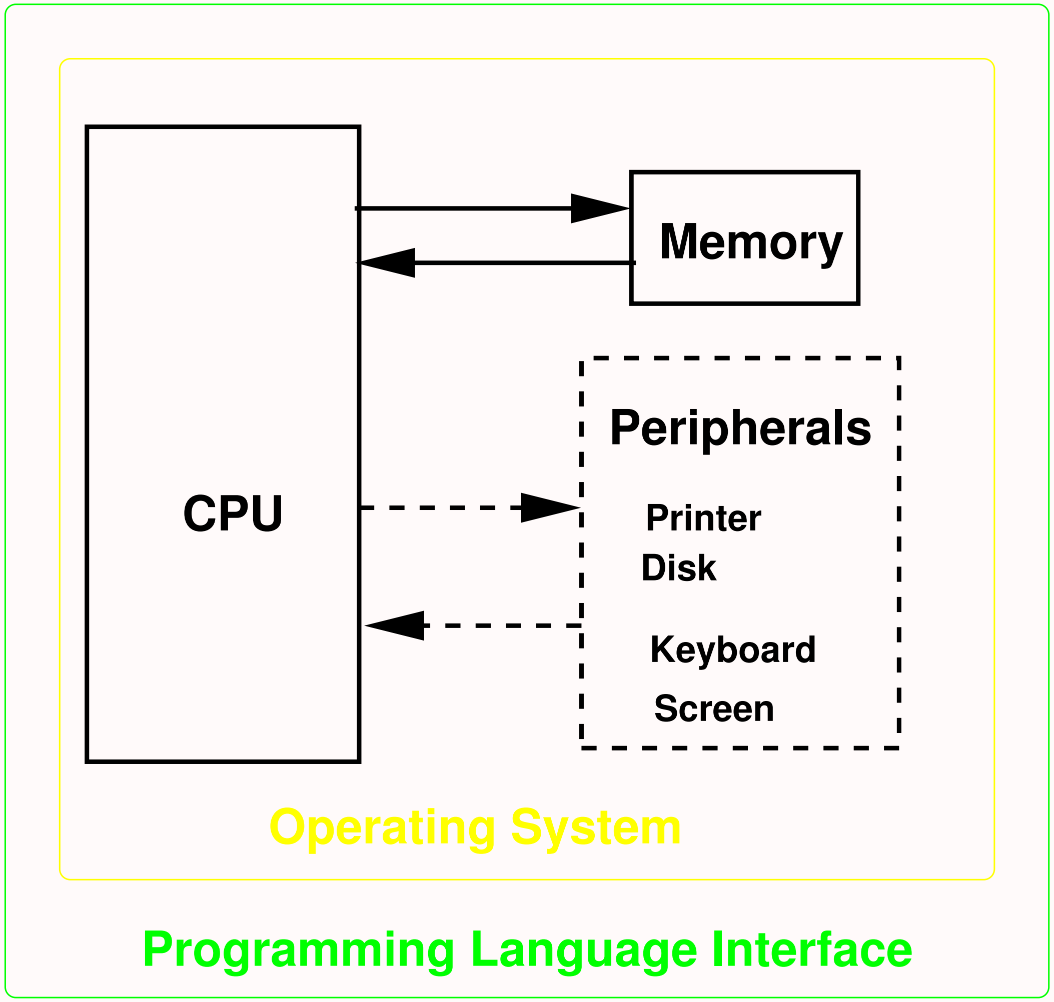
Assume all other cells are filled with null

State Changes: σ_2

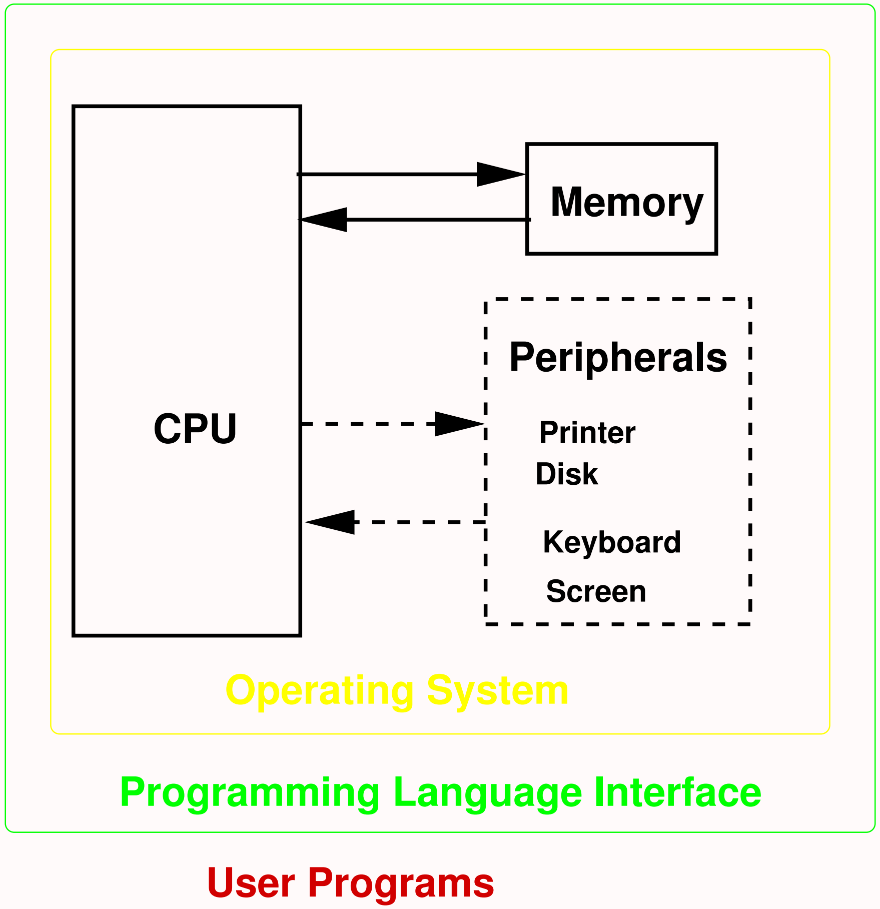
0	1	2	3						
		5							
						3.1			
			true						
12						"#a"			

Assume all other cells are filled with null

Languages


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

User Programs



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 $!y = 2$.

Assignment Commands

Store the value 5 *in* x.

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

x

5

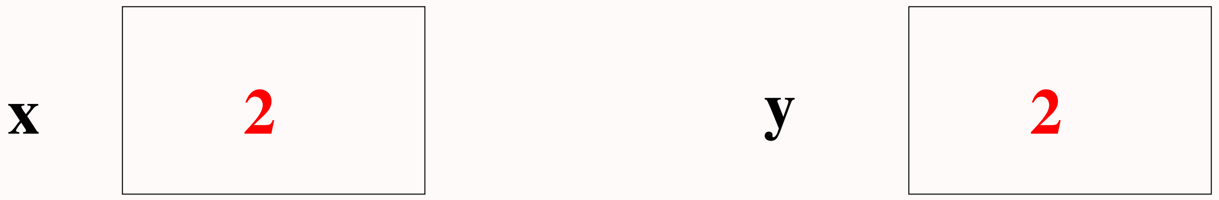
y

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.

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

.

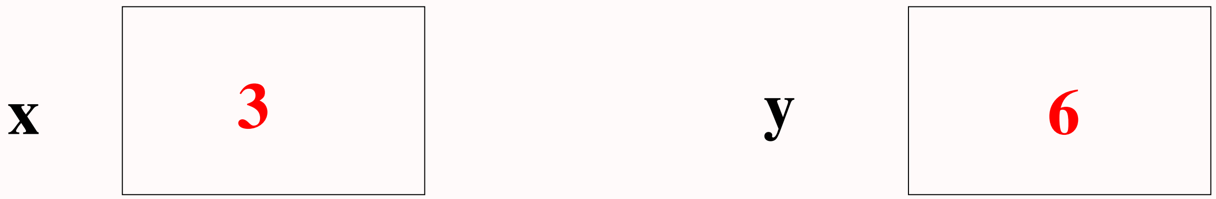
x 3

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?

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

Then the rest is easy

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

Swap

Could be made simpler!

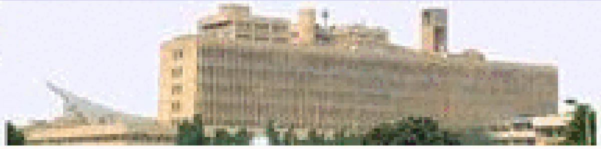
```
val t = ref (!x);
x := !y;
y := !t;
```

Swap

Could use a temporary functional variable t instead of an imperative variable

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

10.2. 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 $ref : \alpha \rightarrow \alpha\ ref$

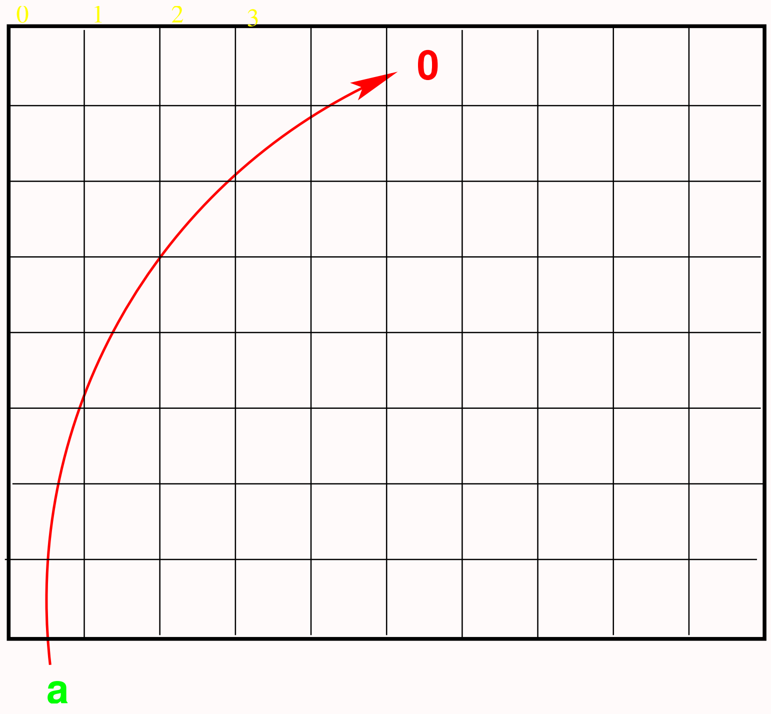
Dereferencing $! : \alpha\ ref \rightarrow \alpha$

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

Deallocation of memory is automatic!

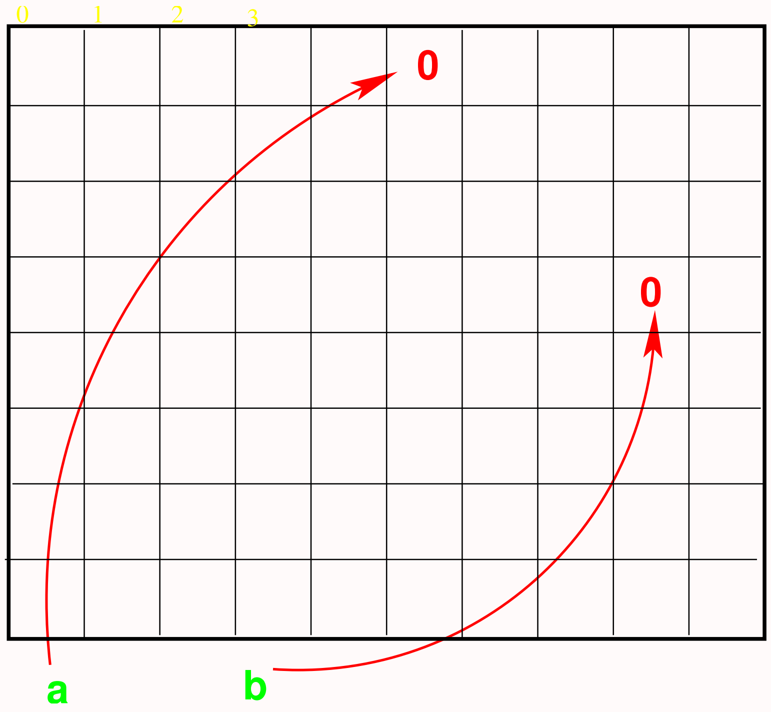
References: Experiments

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



References: Experiments

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

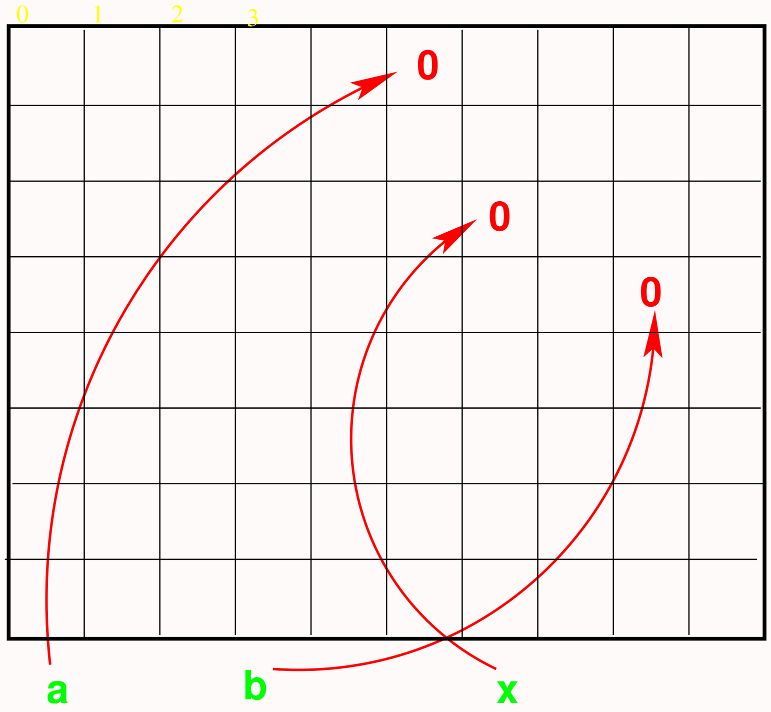


References: Experiments

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

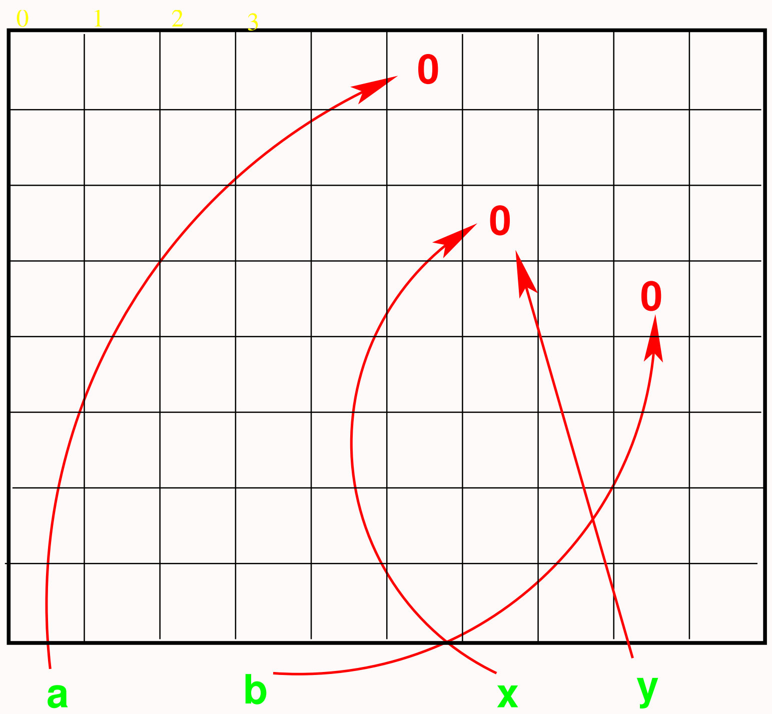
Aliases

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



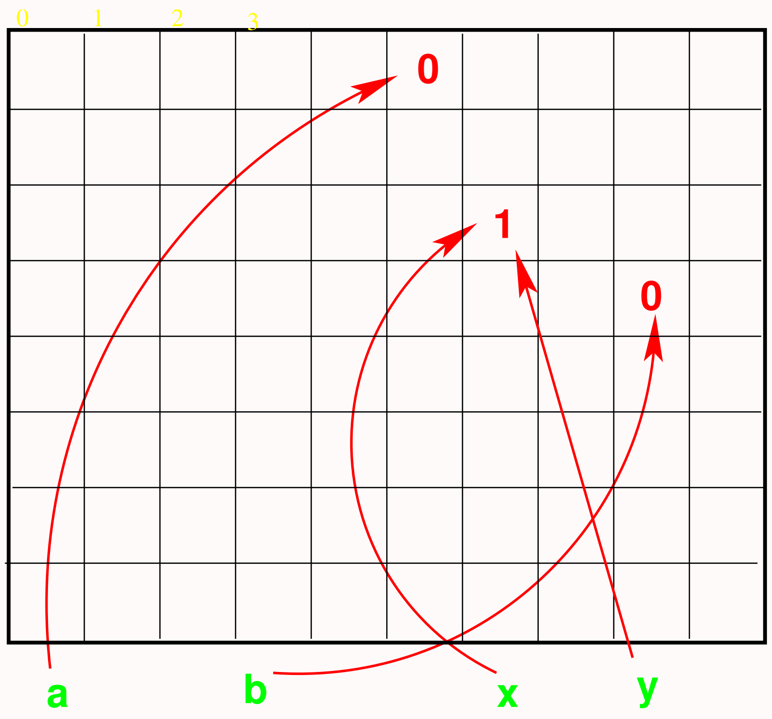
References: Experiments

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



References: Aliases

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

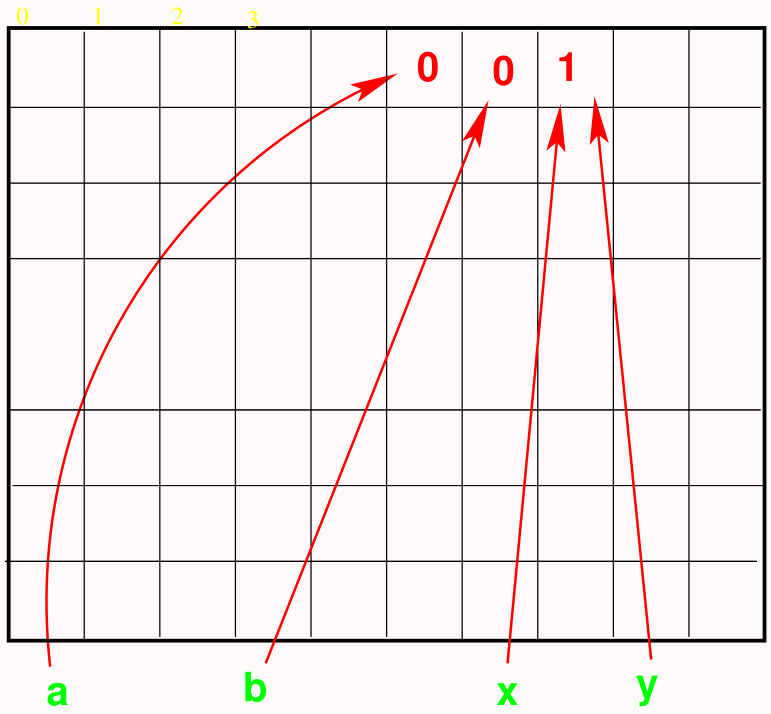


References: Experiments

```
- !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

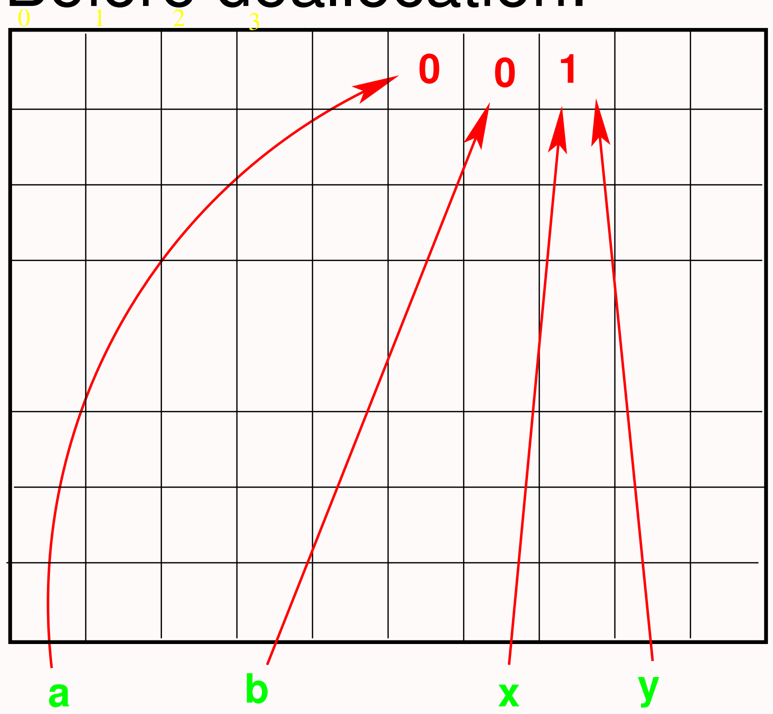
- Does not provide direct access to memory addresses
- Does not allow for uninitialized imperative variables
- Provides a type with every memory cell
- Manages the memory completely automatically and securely
  - Memory has to be managed by the user program in most languages
  - Prone to various errors

Common Errors

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

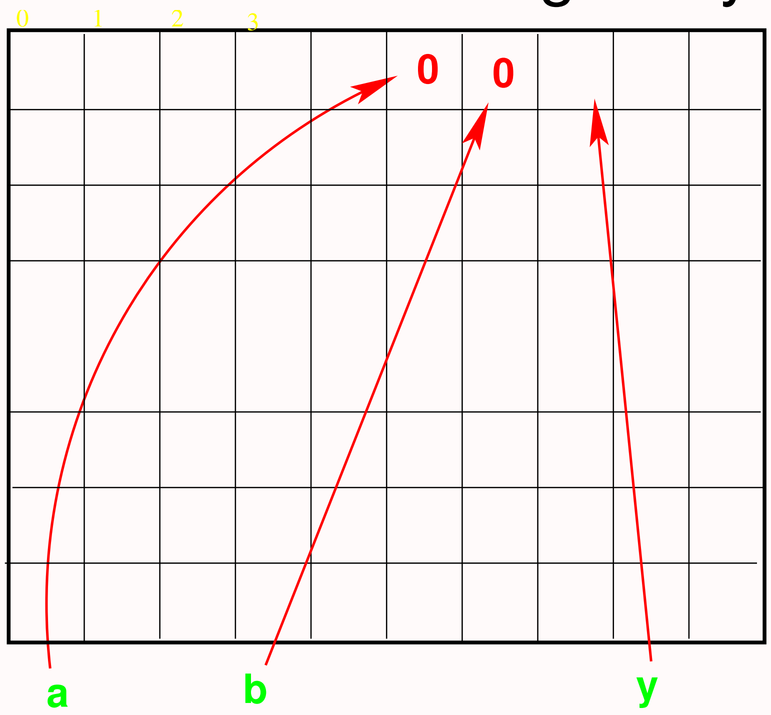
Aliasing & References

Before deallocation:



Dangling References

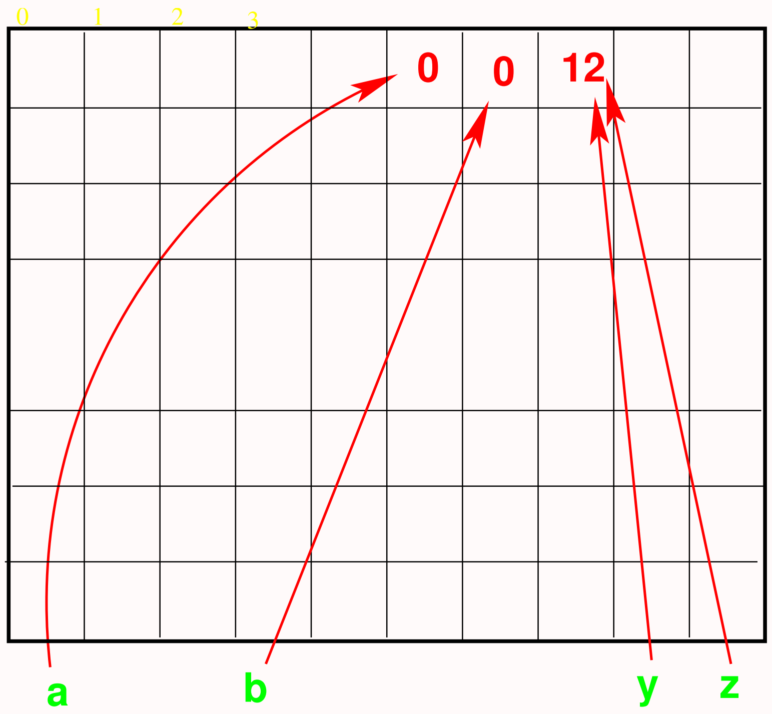
Deallocate x through a system call



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

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

10.3. Arrays

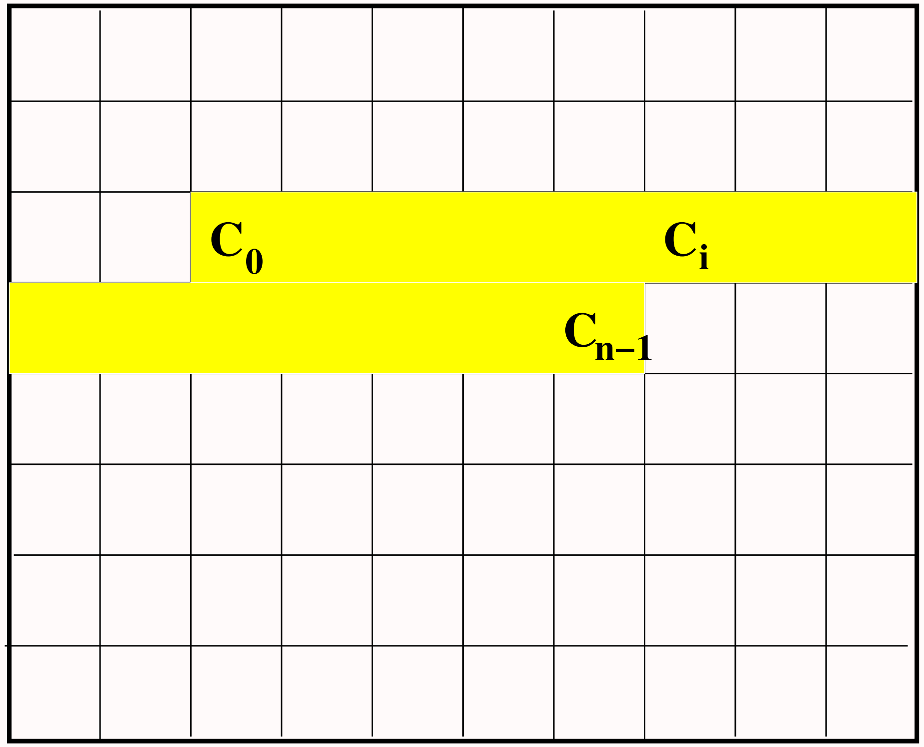
Why Imperative

- 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

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

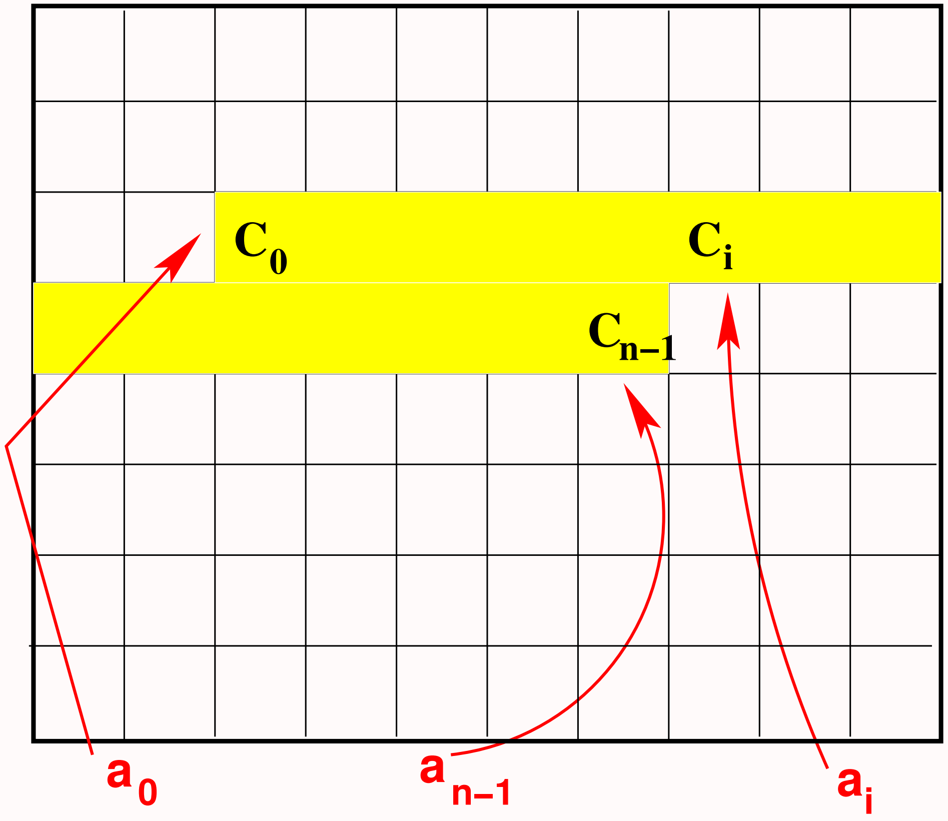


Indexing Arrays

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

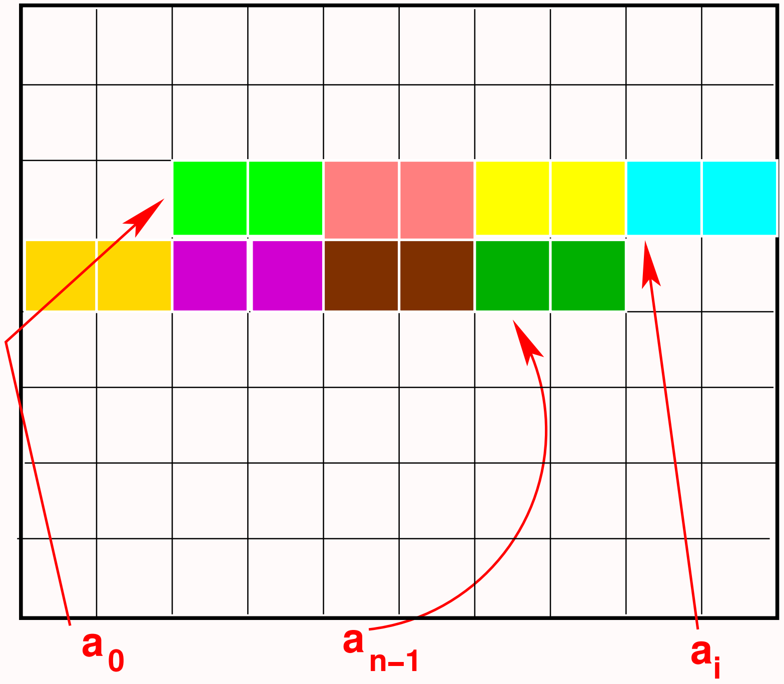
- The *start* address of the array and the *address* of C_0 are the same (say a_0)
- The address a_i of cell C_i is

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

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

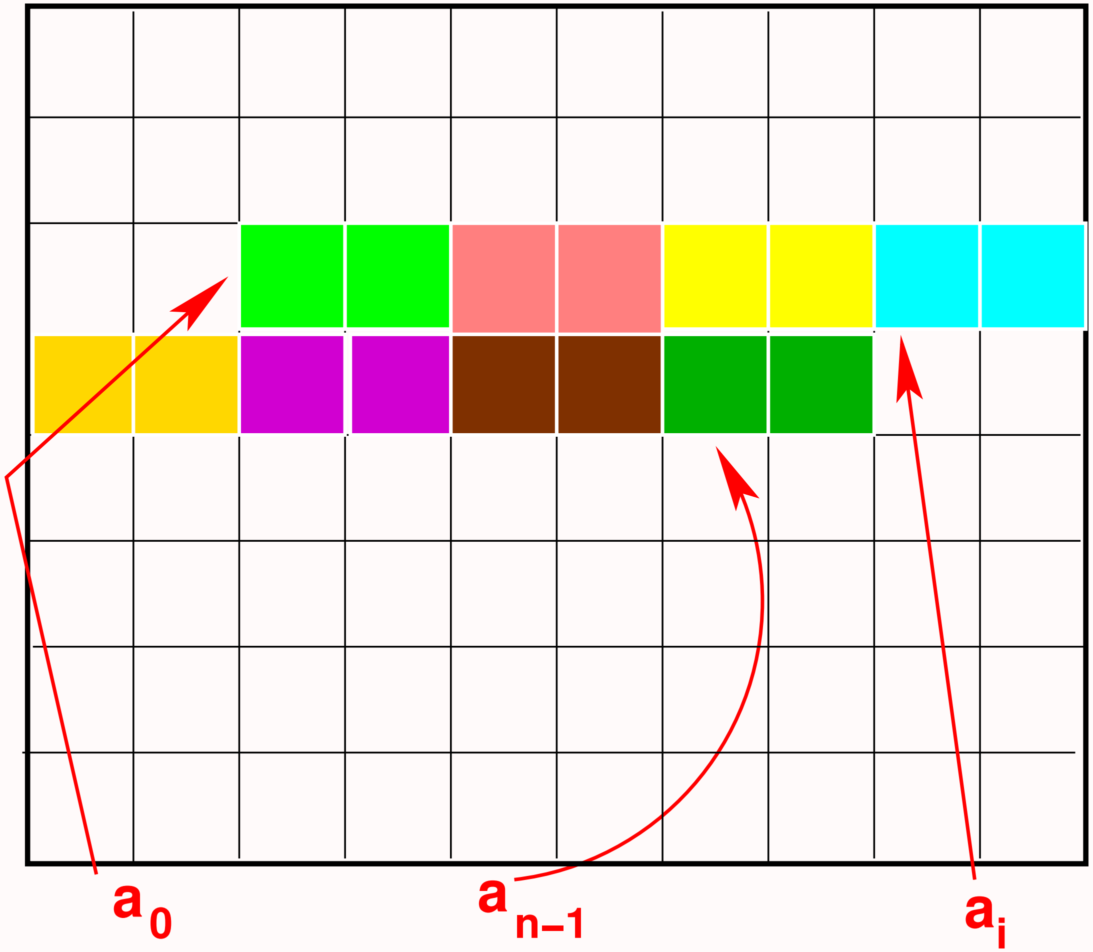
There is a natural “$<$” ordering on indices given by

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

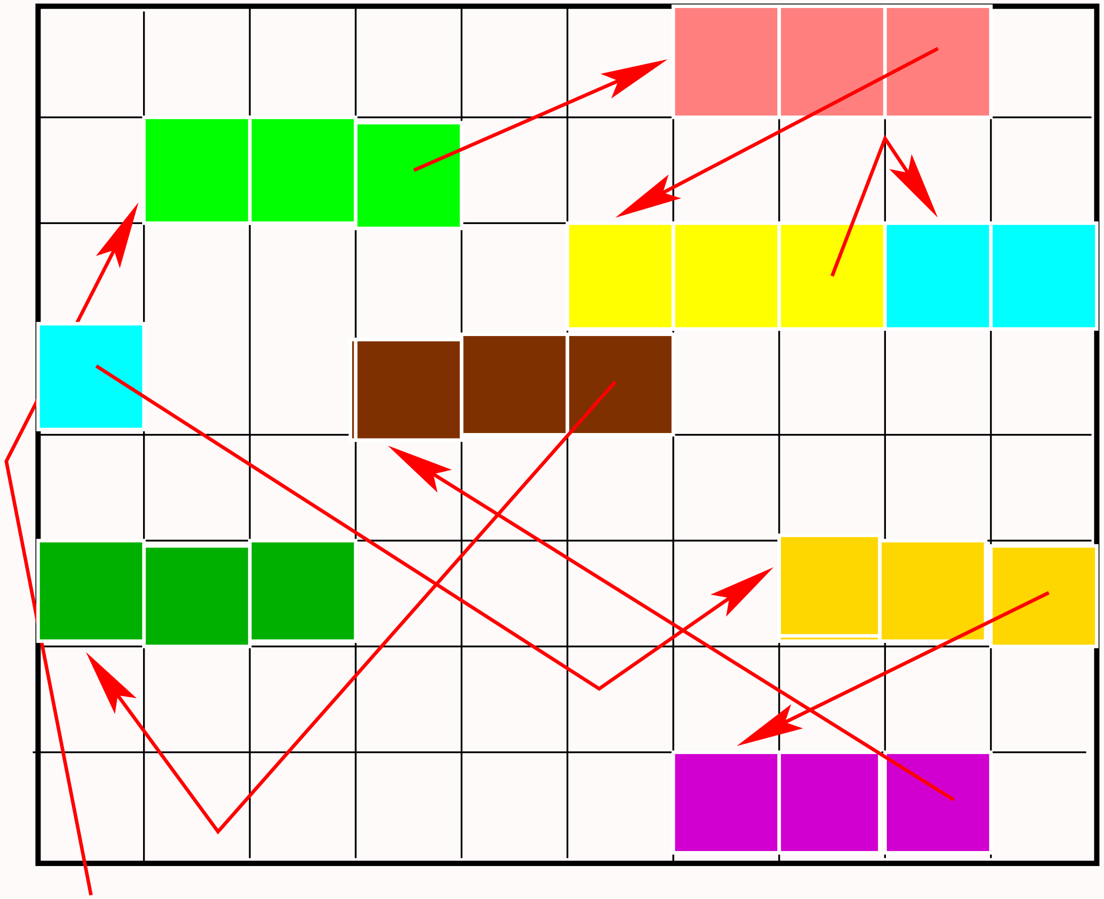
Arrays vs. Lists

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

Arrays: Physical



Lists: Physical



11. A large Example: Tautology Checking

11.1. 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 if and only if every proposition p_i , $1 \leq i \leq m$, is a tautology.
- otherwise at least one clause p_i must be false
- Clause $p_i = \bigvee_{j=1}^{m} l_{ij}$ is false if and only if every literal l_{ij}, $1 \leq j \leq m$ is false

Falsifying

For a proposition in CNF ($\bigwedge_{i=1}^{n} p_i$) that is not a tautology

- A clause $p_i = \bigvee_{j=1}^{m} l_{ij}$ for some i, $1 \leq i \leq n$ is false
- A clause $p_i = \bigvee_{j=1}^{m} l_{ij}$ is false if and only if every literal l_{ij}, for each $j = 1, \ldots, m$ in p_i 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

11.2. 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 <u>single</u> 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 ⟺ and ⇒
- `nnf` converts into NNF
- `conj_of_disj` converts into CNF

Rewrite into NNF

- Eliminate $\Longleftrightarrow$ 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 ¬ 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 ∧ and ∨ 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 ∨ inward

Use distributivity of ∨ over ∧

```
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

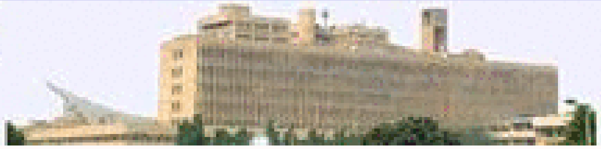
Falsifying a proposition

- A proposition Q in CNF is 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!

```
(*======================== THE SIGNATURE PropLogic ====================== *)
signature PropLogic =

sig
    exception Atom_exception
    datatype Prop =
         ATOM of string      |
         NOT of Prop         |
         AND of Prop * Prop  |
         OR of Prop * Prop   |
         IMP of Prop * Prop  |
         EQL of Prop * Prop
    type Argument = Prop list * Prop
    val show    : Prop -> unit
    val showArg : Argument -> unit
    val falsifyArg : Argument -> Prop list list
    val Valid   : Argument -> bool * Prop list list
end;

(* Propositional formulas *)

(*======================== THE STRUCTURE PL ====================== *)
```

```
structure PL:PropLogic =
(* structure PL = *) (* This is for debugging purposes only *)
struct

    datatype Prop =
        ATOM of string     |
        NOT of Prop        |
        AND of Prop * Prop |
        OR of Prop * Prop  |
        IMP of Prop * Prop |
        EQL of Prop * Prop
    ;

    (* --------------- Propositions to CNFs --------------------- *)

    exception Atom_exception;
    fun newatom (s) = if s = "" then raise Atom_exception
                      else (ATOM s);
    fun drawChar (c, n) =
                     if n>0 then (print(str(c)); drawChar(c, (n-1)))
                     else ();
    fun show (P) =
let  fun drawTabs (n) = drawChar (#"\t", n);
     fun showTreeTabs (ATOM a, n) = (drawTabs (n);
                                    print (a);
                                    print("\n")
                                    )
     |   showTreeTabs (NOT (P), n) = (drawTabs(n); print ("NOT");
                                     showTreeTabs (P, n+1)
                                     )
     |   showTreeTabs (AND (P, Q), n) =
                      (showTreeTabs (P, n+1);
                      drawTabs (n); print("AND\n");
                      showTreeTabs (Q, n+1)
                      )
     |   showTreeTabs (OR (P, Q), n)  =
                      (showTreeTabs (P, n+1);
                      drawTabs (n); print("OR\n");
                      showTreeTabs (Q, n+1)
                      )

     |   showTreeTabs (IMP (P, Q), n) =
                      (showTreeTabs (P, n+1);
                      drawTabs (n); print("IMPLIES\n");
```

```
                         showTreeTabs (Q, n+1)
                         )
        |   showTreeTabs (EQL (P, Q), n) =
                         (showTreeTabs (P, n+1);
                         drawTabs (n); print("IFF\n");
                         showTreeTabs (Q, n+1)
                         )
        ;
    in  (print ("\n"); showTreeTabs(P, 0); print ("\n"))

    end
;


(* The function below evaluates a formula given a truth assignment.
   The truth assignment is given as a list of atoms that are true
   (all other atoms are false).
*)


fun lookup (x:Prop, []) = false
|   lookup (x, h::L)    = (x = h) orelse lookup (x, L)
```



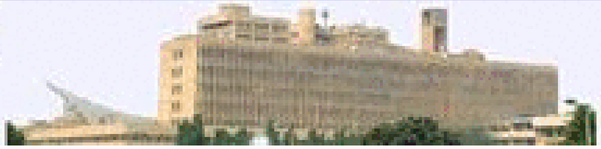
```
;

fun eval (ATOM a, L)      = lookup (ATOM a, L)
|   eval (NOT (P), L)     = if eval (P, L) then false else true
|   eval (AND (P, Q), L) = eval (P, L) andalso eval (Q, L)
|   eval (OR (P, Q), L)  = eval (P, L) orelse eval (Q, L)
|   eval (IMP (P, Q), L) = eval (OR (NOT (P), Q), L)
|   eval (EQL (P, Q), L) = (eval (P, L) = eval (Q, L))
;

(* We could also write a tautology checker with out using truth
   assignments by first converting everything into a normal form.
*)

(* First rewrite implications and equivalences *)

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

```
;

(* Convert all formulas not containing IMP or EQL into Negation Normal
   Form.
*)

fun nnf (ATOM a)           = ATOM a
|   nnf (NOT (ATOM a))     = NOT (ATOM a)
|   nnf (NOT (NOT (P)))    = nnf (P)
|   nnf (AND (P, Q))       = AND (nnf(P), nnf(Q))
|   nnf (NOT (AND (P, Q))) = nnf (OR (NOT (P), NOT (Q)))
|   nnf (OR (P, Q))        = OR (nnf(P), nnf(Q))
|   nnf (NOT (OR (P, Q)))  = nnf (AND (NOT (P), NOT (Q)))
;

(* Distribute OR over AND to get a NNF into CNF *)

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)

(* Now the CNF can be easily computed *)
```

```
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
;

fun cnf (P) = conj_of_disj (nnf (rewrite (P)));

(* ------------------ Propositions to CNFs ------------------------- *)


(* ------------------ CNFs: Pure Tautology Checking --------------- *)

(* A proposition in CNF is a tautology
          iff
   Every conjunct is a tautology
          iff
   Every disjunct in every conjunct contains both positive and negative
   literals of at least one atom

So we construct list of all the positive and negative atoms in every
disjunct to check whether the lists are all equal. We need a binary
function on lists to determine whether two list are disjoint
*)

fun isPresent (a, [])    = false
|   isPresent (a, b::L) = (a = b) orelse isPresent (a, L)
;

fun disjoint ([], M)                     = true
|   disjoint (L, [])                     = true
|   disjoint (L as a::LL, M as b::MM) =
        not(isPresent (a, M)) andalso
        not(isPresent(b, L))  andalso
        disjoint (LL, MM)
;

(* ABHISHEK : Defining a total ordering on atoms (lexicographic
   ordering on underlying strings), and extending it to a list of atoms.
*)

exception notAtom;

fun atomLess (a, b) =
        case (a, b) of
           (ATOM(x), ATOM(y)) => x<y
         | (_,_)              => raise notAtom;

fun listLess (a, b) =
        case (a, b) of
           (_, [])            => false
         | ([], _)            => true
         | (x::lx, y::ly)     => if atomLess(x,y) then true
                                 else if atomLess(y,x) then false
                                 else listLess(lx,ly);

(* ABHISHEK : Once we have a list of falsifiers, we would want to remove
   any duplication, firstly of atoms within a falsifier, and secondly of
   falsifiers themselves.

   In order to do this, we maintain all lists in some sorted order.
   Instead of sorting a list with a possibly large number of duplicates,
   we check for duplicates while inserting, and omit insertion if a
   previous instance is detected.

*)
```

```
fun merge less ([],l2) = l2
|   merge less (l1,[]) = l1
|   merge less (x::l1,y::l2) =
    if less(x,y) then x::merge less (l1,y::l2)
    else if less(y,x) then y::merge less (x::l1,l2)
    else merge less (x::l1,l2);

(* ABHISHEK : Claim is that if all lists are built through the above
   function, then there is no need to sort or remove duplicates.

   Hence all '@' operations have been replaced by merge.
*)

(* To separate the positive from the negative literals in a clause *)

exception not_CNF;

fun separate (ATOM a)       = ([ATOM a], [])
  | separate (NOT (ATOM a)) = ([], [ATOM a])
  | separate (OR (P, Q))    =
    let val (posP, negP) = separate (P);
```

```
       val (posQ, negQ) = separate (Q)
    in  (merge atomLess (posP, posQ), merge atomLess (negP, negQ))
    end
  | separate (P)             = raise not_CNF

(* Check whether a formula in CNF is a tautology *)

fun taut (AND (P, Q)) = taut (P) andalso taut (Q)
  | taut (P) = (* if it is not a conjunction then it must be a disjunct *)
        not (disjoint (separate (P)));

fun tautology1 (P) =
    let val Q = cnf (P)
    in  taut (Q)
    end
;

(* ------------------ CNFs: Pure Tautology Checking ------------------ *)


(* ------------------ CNFs: Falsifications ------------------ *)
(* The main problem with the above is that it checks whether a given
   proposition is a tautology, but whenever it is not, it does not yield
   a falsifying truth assignment. We rectify this problem below.

*)

(* Assume Q is a proposition in CNF. Then it is only necessary to list out
   all the lists of truth assignments that can falsify P.

   Suppose Q is in CNF, but not a tautology. Further let

      Q = AND (D1, ..., Dn)

   where each Di is a disjunction of literals. Each Di = Pi + Ni where
   Pi and Ni are the lists of atoms denoting the positive and negative
   literals respectively.

   If Pi and Ni are disjoint, then Ni is a truth assignment which falsifies
   the proposition Q. That is,

   (i)    if every atom in Ni is assigned TRUE, and
   (ii)   (implicitly) every other atom is assigned the value FALSE,
```



```
   then the clause Di is FALSE and hence Q is FALSE.

   We refer to Ni as a FALSIFIER of Q. Hence it is only
   necessary to list out all the FALSIFIERS of Q (if any)

*)

(* The following function assumes Q is in CNF and outputs a list of list
   of atoms that can falsify Q. If this list of list of atoms is empty then
   clearly Q cannot be falsified and is hence  a tautology.
*)


fun falsify (Q) =
    let fun list_Falsifiers (AND (A, B)) =
                  merge listLess (list_Falsifiers (A),list_Falsifiers (B))
        |   list_Falsifiers (A) = (* Assume A is a disjunct of literals *)
                      let val (PLA, NLA) = separate (A)
                      in  if disjoint (PLA, NLA) then [NLA]
                          else []
                      end
    in list_Falsifiers (Q)
    end
;

fun tautology2 (P) =
    let val Q = cnf (P);
            val LL = falsify (Q)
    in      if null (LL) then (true, [])
            else (false, LL)
    end
;

val tautology = tautology2;

(* ---------------- CNFs: Falsifications ---------------- *)

(* ---------------- Logical Arguments: Validity ---------------- *)

type Argument = Prop list * Prop;

fun showArg (A: Argument) =
    let fun printArg (A:Argument as ([], c)) =
            (drawChar (#"-", 80); print("\n");
             show (c); print ("\n\n")
            )
        |   printArg (A:Argument as (p::plist, c)) =
             (show (p); print ("\n");
              printArg (plist, c)
             )
    in (print ("\n\n"); printArg (A))
    end
;

fun leftReduce (F) =
    let exception emptylist;
           fun lr ([]) = raise emptylist
           |   lr ([a]) = a
           |   lr (a::L) = F (a, lr (L))
    in  lr
    end
;

val bigAND = leftReduce (AND);
```

```
    fun Valid ((L, P):Argument) =
        if null (L) then tautology (P)
        else tautology (IMP (bigAND (L), P))
    ;

    fun falsifyArg ((L, P): Argument) =
        if null (L) then falsify (cnf(P))
        else falsify (cnf (IMP (bigAND (L), P)))
    ;

end (* struct *);

(* open PL; *)
```

12. Lecture-wise Index to Slides

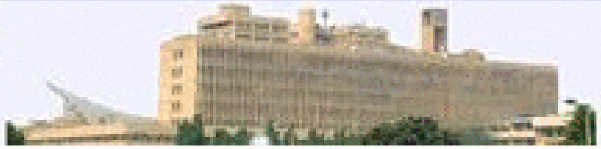
Index

Introduction to Computing (3-8)

Next: Our Computing Tool

Our Computing Tool (9-33)

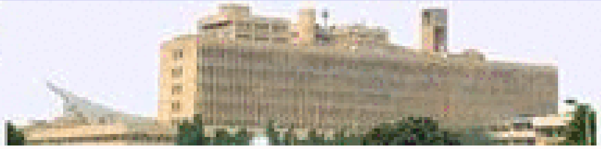
Previous: Introduction to Computing



Next: Primitives: Integer & Real

Primitives: Integer & Real (34-60)

Previous: Primitives: Integer & Real



Next: Technical Completeness & Algorithms

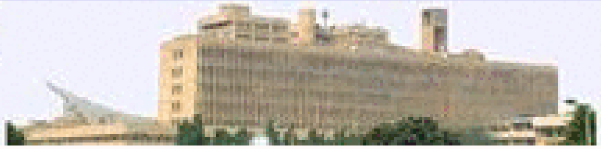
Technical Completeness & Algorithms (61-92)

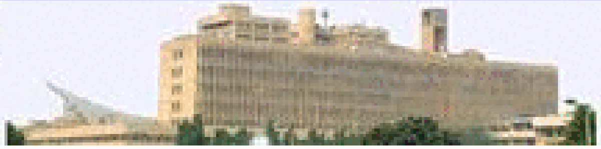
Invariance & Correctness (296-319)

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

Structured Data

Imperative Programming: (500-524)





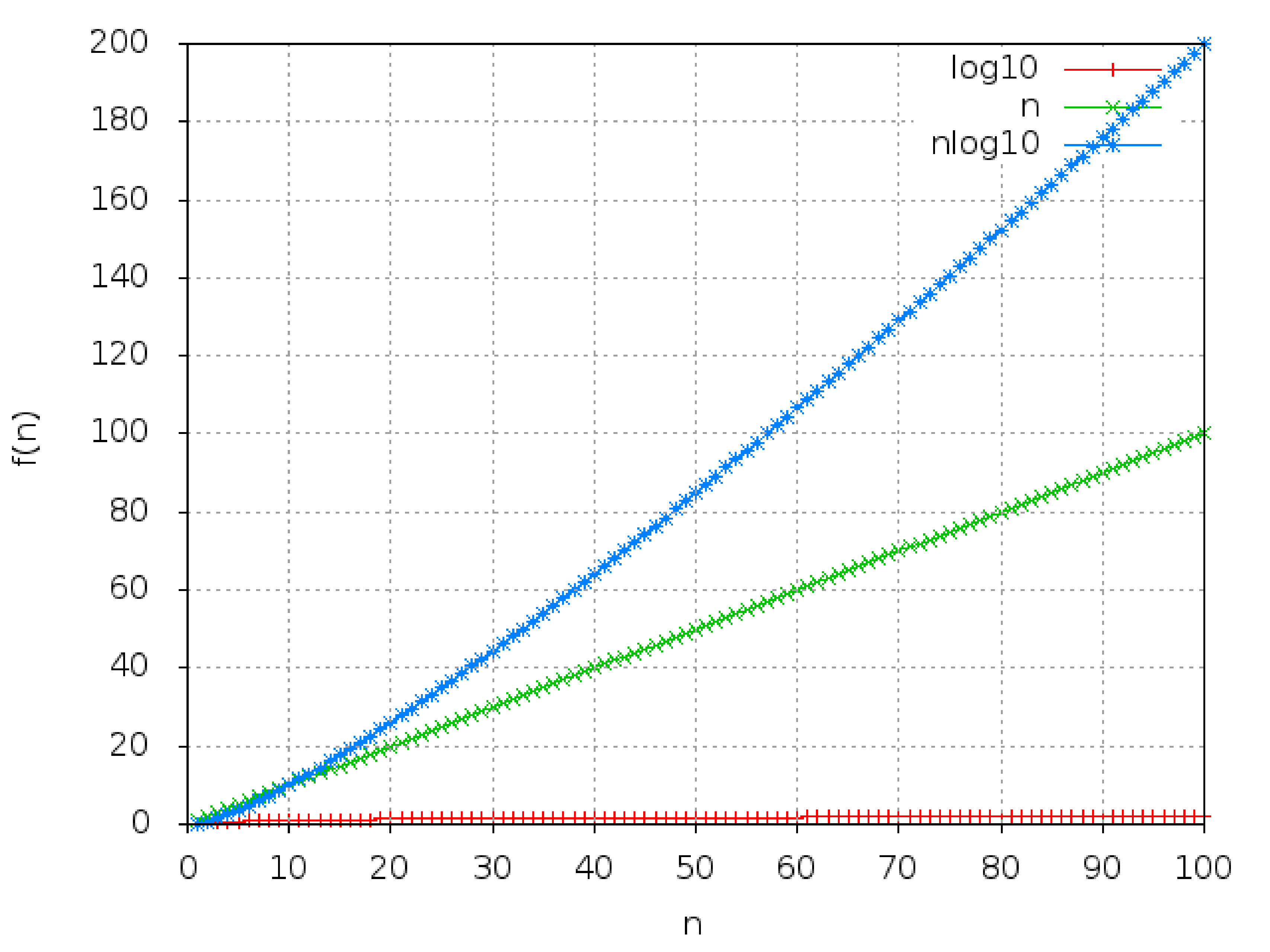


Figure 1: Plot of $y = \log_{10} n$, $y = n$ and $y = n \log_{10} n$

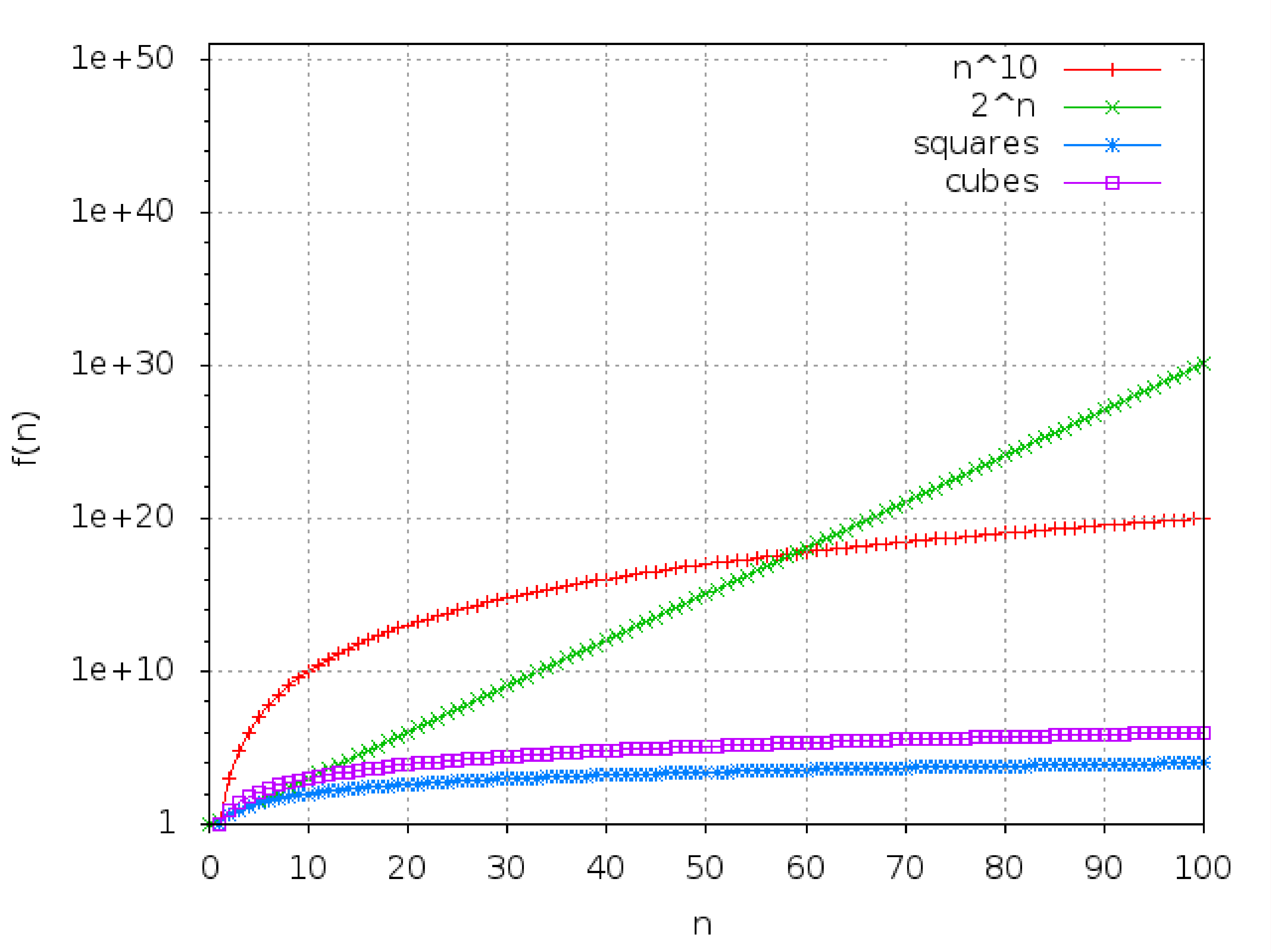


Figure 2: Plots of $y = n^2$, $y = n^3$, $y = n^10$ and $y = 2^n$