Functional Programming with ML

Winter Term 2001/2002

Dept. Mathematik/Informatik, Universität Osnabrück

Ute Schmid & Marieke Rohde (Tutor)

Requirements:

- Basic knowledge in programming and algorithm design (Lecture “Informatik A”)
- helpful: Basic knowledge in theoretical computer science (Lecture “Informatik D”)

`http://www.vorlesungen.uos.de/informatik/fp01/`

This scriptum is a collection of slides and information presented in the lecture. It is based mainly on the text book “ML for the Working Programmer” from Larry Paulson. Further text books and papers on which the lecture is based are given in the reference section at the beginning of this scriptum.

Thanks to Elmar Ludwig for error-checking and helpful comments!

Inhaltsverzeichnis

Literature

Text Books:

L. C. Paulson (1997). ML for the Working Programmer, 2nd Edition. Cambridge University Press.

P. Pepper (2000). Funktionale Programmierung in Opal, ML, Haskell und Gofer. Springer.

A. Field and P. Harrison (1988). Functional Programming. Addison-Wesley.

Lisp/AI Applications:

P. Winston and B. Horn (1984). Lisp, 2nd edition. Addison-Wesley.

P. Norvig (1996). Paradigms of artificial intelligence programming: case studies in Common LISP, 3rd print. Morgan Kaufmann.

Theoretical Background:

B. A. Davey and H. A. Priestley (1990). Introduction to Lattices and Order. Cambridge University Press.

J. Loeckx and K. Sieber (1984). The Foundations of Program Verification. Teubner.

J. Loeckx, K. Mehlhorn, and R. Wilhelm (1986). Grundlagen der Programmiersprachen. Teubner.

Papers: *Here you will find the papers which are used during the lecture.*

J. Backus (1978). Can Programming be Liberated from the von Neumann Style? *Communications of the ACM*, *8*, 613–641.

J. Giesl (1999). Context-Moving Transformations for Function-Verification. Proc. of the 9th Int. Conf. on Logic-Based Program Synthesis and Transformation (LOPSTR-99), pp. 293-312. Springer LNCS 1817.

J.A. Robinson (1965). A machine-oriented logic based on the resolution principle, *Journal of the ACM*, *12*(*1*), 23-41.

R.S. Boyer and J.S. Moore (1975). Proving theorems about LISP functions. *Journal of the ACM*, *22*(*1*), 129-144.

1 Introduction

1.1 Why Functional Programming?

Reasons for studying functional programming (P. Pepper):

- Conceptional level: understand what programming is about; learning concepts not given in conventional languages.
- Practical level: hightened productivity of software development and verification
- "*Informatiker wird man nicht, indem man ein oder zwei spezielle Programmiersprachen beherrscht – das kann jeder Hacker. Informatiker zeichnet aus, dass sie das Prinzip des Programmierens und das Prinzip des 'Sich-in Programmiersprachen-Ausdrückens' beherrschen.*"
- Spectrum: Imperative/Objectoriented and logical/functional languages.

Functional Programming with ML (L. Paulsen)

- "*Using ML, students can learn how to analyse problems mathematically, breaking the bad habits learned from low-level languages. Significant computations can be expressed in a few lines.*"

1.2 'Can Programming Be Liberated from the von-Neumann Style?'

- John Backus' Turing Award Lecture 1977.
- Developper of the (imperative!) Languages Fortran and Algol 60.

1.2.1 Conventional Programming Languages: Fat and Flabby

- Each successive language incorporates, with a little cleaning up, all the features of its predecessors plus a few more.
  Each new language claims new and fashionable features, but the plain fact is that few languages make programming sufficiently cheaper or more reliable to justify the cost of producing and learning to use them.
  $\hookrightarrow$ Large increases in size bring only small increases in power!
- (Programming) is now the province of those who prefer to work with thick compedia of details rather than wrestle with new ideas.
  Discussions about programming languages often resemble medieval debates about the number of angles that can dance on the head of a pin instead of exciting contests between fundamentally differing concepts.
- $\hookrightarrow$ basic defects in the framework of conventional languages make their expressive weakness and their cancerous growth inevitable.
  $\hookrightarrow$ alternate avenues of exploration toward the design of new kinds of languages

1.2.2 Models of Computing Systems

Simple Operation Models: (Turing machines, automata)
Foundations: concise and useful
History sensitivity: have storage, history sensitive
Semantics: state transition with very simple states
Program clarity: unclear, conceptually not helpful

Applicative Models: (lambda-Calculus, combinators, pure lisp)
Foundations: concise and useful
History sensitivity: no storage, not history sensitive
Semantics: reduction semantics, no states
Program clarity: can be clear and conceptually helpful

Von Neumann Models: (conventional programming languages)
Foundations: complex, bulky, not useful
History sensitivity: have storage, history sensitive
Semantics: state transition with complex states
Program clarity: can be moderately clear, conceptually not helpful

Reduction semantics: stepwise transformation (reduction) until a “normal form” is reached.

1.2.3 Von Neumann Computers and Von Neumann Languages

- von-Neumann bottleneck: pumping single words back and forth between CPU and store
  Task of a program: change store in some major way.
  Word must be generated in the CPU or sent from the store (its address must have been sent from the store or generated by the CPU, ...)
- The tube is an *intellectual bottleneck* that has kept us tied to word-at-a-time thinking instead of of encouraging us to think in terms of the larger conceptual units of the task at hand.
  Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself but where to find it!
- Conventional programming languages are basically high level, complex versions of the von Neumann computer.
  Our belief that there is only one kind of computer is the basis of our belief that there is only one kind of programming language!

Remark: Although Java is an object oriented language, methods are mostly/typically realized in an imperative/von Neumann style: assignments of values to variables and fields.

Characteristics of von Neumann Languages

- Variables imitate the computer's storage cells; assignements imitate fetching and storing.
- *The assignement statement is the von Neumann bottleneck of programming languages*!
- The assignement statement splits programming into two worlds:
  - Right side of assignement statements: orderly world of expressions, useful algebraic properties (except when destroyed by side-effects), useful computations
  - Statements: assignement statement is primary, all other statements must be based on this construct

Remark: side effect means that the value of a variable can be changed by a statement given in a different part of the program (different method, even different class).
Functional languages are free of side effcts: referential transparency, expressions can be replaced by values without a change of the overall result!

1.2.4 Comparison of von Neumann and Functional Programming

von Neumann program for inner product:

```
c := 0;
for i := 1 step 1 until n do
   c := c + a[i] x b[i]
```

- statements operate on an invisible state
- non hierachical, no construction of complex entities from simpler ones
- dynamic and repetitive (to understand: mentally execute)
- word-at-a-time computition by repetition (of assignement) and modification (of i)
- part of the data is in the program (n), thus it works only for vectors of length n
- arguments are named; it can only be used for vectors a and b (to become general: procedure declaration)
- “housekeeping” operations are scattered; not possible to consolidate them into single operations; one must always start again from square one when writing programs (writing `for i := ...`)

functional program for inner product:

```
Def Innerproduct ==
      (insert +) o (ApplyToAll x) o Transpose
```

$DefIP \equiv (/+) \circ (\alpha\times) \circ Trans$

- **Functional Forms:** combination of existing functions to form new ones:
  $f \circ g$: function obtained by applying first g and then f
  αf: function obtained by applying f to every *member* of the argument
- Application: $f : x$

Example evaluation: $x = \langle\langle 1,2,3\rangle, \langle 6,5,4\rangle\rangle$

$IP : \langle\langle 1,2,3\rangle, \langle 6,5,4\rangle\rangle =$	
Definition of IP $\Rightarrow$	$(/+) \circ (\alpha\times) \circ Trans : \langle\langle 1,2,3\rangle, \langle 6,5,4\rangle\rangle$
Effect of composition $\Rightarrow$	$(/+) : ((\alpha\times) : (Trans : \langle\langle 1,2,3\rangle, \langle 6,5,4\rangle))$
Applying Transpose $\Rightarrow$	$(/+) : ((\alpha\times) : \langle\langle 1,6\rangle, \langle 2,5\rangle, \langle 3,4\rangle\rangle)$
Effect of α $\Rightarrow$	$(/+) : \langle\times : \langle 1,6\rangle, \times : \langle 2,5\rangle, \times : \langle 3,4\rangle\rangle)$
Applying $\times$ $\Rightarrow$	$(/+) : \langle 6,10,12\rangle$
Effect of $/$ $\Rightarrow$	$+ : \langle 6, + : \langle 10,12\rangle\rangle$
Applying $+$ $\Rightarrow$	$+ : \langle 6,22\rangle$
Applying $+$ $\Rightarrow$	28

- operates only on its arguments; no hidden states or complex transition rules; only two kinds of rules: applying a function to its argument and obtaining a function denoted by a functional form
- hierarchical: built from simpler functions ($+$, $\times$, Trans) and functional forms ($f \circ g$, αf, $/f$)
- static and non-repetitive: can be understood without mental execution
- operates on whole conceptual units, not words; three steps, non is repeated
- incorporates no data; completely general; works for all pairs of conformable vectors
- does not name arguments; can be applied to any pair of vectors
- housekeeping forms and functions which are generally useful; only $+$ and $\times$ are not concerned with housekeeping

1.2.5 Language Frameworks versus Changeable Parts

- Framework: overall rules of the system
  e. g., for-statement
  $\hookrightarrow$ fixed features, general environment for its changeable features
- Changable parts: library functions, user-defined procedures
- Language with a small framework: support many different features and styles without being changed itself
- von Neumann languages have large frameworks, because:
  - semantics closely coupled to states, every feature must be built into the state and its transitition rules
  - changable parts have little expressive power (“their gargantuan size is eloquent proof of this”)

1.2.6 Changable Parts and Combining Forms

- Powerful changable parts: combining forms
- von Neumann languages: only primitive combining forms (for, while, if-then-else)
  - split between expressions and statements (in a functional language there are only expressions!)
    expressions can only be used to produce a one-word result!
  - elaborate naming conventions, substitution rules required for calling procedures
    complex mechanism must be built into the framework

1.2.7 Von Neumann Languages Lack Useful Mathematical Properties

It is hard to reason about von Neumann programs (correctness, termination).

- **Denotational Semantics:** understanding the domain and function spaces implicit in programs
  when applied to functional (recursive) programs: powerful tool for describing the language and proving properties of programs
  when applied to von Neumann languages: precise semantic description, helpful for identifying troublespots of the language, but: complexity of the language is reflected in complexity of description
- **Axiomatic Semantics:** (Hoare calculus), precisely restates the inelegant properties of the von Neumann programs
  success: (1) restriction to small subsets of von Neumann languages; (2) predicates and transformations are more orderly
  "*... it is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea?*" (Hoare, 1989)
- "... using denotational or axiomatic semantics to describe a von Neumann language can not produce an elegant and more powerful language any more then the use of elegant and modern machines to build an Edsel can produce an elegant and modern car."

- Proofs about programs use the language of logic, not the language of programming; proofs talk *about* programs but do not involve them directly.
- Ordinary proofs are derived by algebraic methods in a language that has certain algebraic properties; proofs are performed in a mechanical way by application of algebraic laws
- Programs in a functional language have an associated algebra, proofs use the language of the programs themselves!

$ax + bx = a + b$, $a + b \neq 0$
$(a + b)x = a + b$
$(a + b)x = (a + b)1$
$x = 1$

1.2.8 What Are the Alternatives to von Neumann Languages

Functional Style of Programming: FP, based on use of combining forms

Algebra of Functional Programs: Algebra whose variables denote FP programs and whose operations are FP functional forms (combining forms of the FP programs); some algebraic laws; relation to Church (Lambda calculus) and Curry (combinators)

(For obtaining history sensitivity, an applicative state-transition system is proposed.)

1.2.9 FP Systems

- Fixed set of combining forms, called functional forms.
- Functional forms and simple definitions are the only means of building new functions from existing ones.
- no variables, no substitution rules.
- All functions map objects into objects and take a single argument.
- Has not the freedom and power of lambda-calculus “with unrestricted freedom comes chaos”

1.3 Possible Topics for Student Projects

Implementationen in ML:

- Interpreter for FP
- Tautology-Checker for OBDDs (Bryant, 1992; Moore, 1994)
- Theorem Prover (Tableau Method; Lisp Theorem Prover, Boyer & Moore)

2 Backus' FP Systems

Backus (1978, sect. 11)

2.1 Components of an FP System

An FP system comprises the following:

1. a set O of *objects*,
2. a set F of *functions* f, that map objects into objects,
3. an operation *application*,
4. a set $\mathbf{F}$ of *functional forms*, to combine functions ore ovbjects to new functions in F,
5. a set D of *definitions* that define functions in F and assign a name to each.

2.1.1 Objects

Objects O:

- an *atom*,
- a sequence $\langle x_1, \ldots, x_n \rangle$ whose elements x_i are objects,
- $\perp$ (bottom, the undefined object)

Atom ϕ denotes the empty sequence (object which is atom and sequence!, cf., *nil* in Lisp).
Atoms T and F denote "true" and "false".
If x is a sequence containing $\perp$, then $x = \perp$ (sequence constructor is "bottom preserving/strict")

2.1.2 Application

Application: If f is a function and x is an object, then $f : x$ is an application which denotes the object which is the result of applying f to x.
f is the *operator* and x is the *operand*.
Application is the only operation in FP.

Examples:
$+ : \langle 1, 2 \rangle = 3$ $tl : \langle A, B, C \rangle = \langle B, C \rangle$

2.1.3 Functions

Functions F:
All functions $f \in F$ map objects into objects and are bottom preserving ($f : \perp = \perp$, "f is undefined at x")

- primitive (supplied with the system), or
- defined, or
- functional form.

Selector Functions:
$1 : x \equiv x = \langle x_1, \ldots, x_n \rangle \rightarrow x_1; \perp$
$s : x \equiv x = \langle x_1, \ldots, x_n \rangle \wedge n \geq s \rightarrow x_s; \perp$
(Variant of the McCarhty Conditions: $p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n; e_{n+1}$)
Tail: $tl : x \equiv x = \langle x_1 \rangle \rightarrow \phi; x = \langle x_1, \ldots, x_n \rangle \wedge n \geq 2 \rightarrow \langle x_2, \ldots, x_n \rangle; \perp$
Identity: $id : x \equiv x$
Atom: $atom : x \equiv x \; is \; an \; atom \; \rightarrow T; x \neq \perp \rightarrow F; \perp$
Equals: $eq : x \equiv x = \langle y, z \rangle \wedge y = z \rightarrow T; x = \langle y, z \rangle \wedge y \neq z \rightarrow F; \perp$
Null: $null : x \equiv x = \phi \rightarrow T; x \neq \perp \rightarrow F; \perp$
Reverse: $reverse : x \equiv x = \phi \rightarrow \phi; x = \langle x_1, \ldots x_n \rangle \rightarrow \langle x_n, \ldots x_1 \rangle; \perp$
Distribute form left; from right:
$distl : x \equiv x = \langle y, \phi \rangle \rightarrow \phi; x = \langle y, \langle z_1, \ldots, z_n \rangle \rangle \rightarrow \langle \langle y, z_1 \rangle, \ldots \langle y, z_n \rangle \rangle; \perp$
$distr : x \equiv x = \langle \phi, y \rangle \rightarrow \phi; x = \langle \langle y_1, \ldots y_n \rangle, z \rangle \rightarrow \langle \langle y_1, z \rangle, \ldots \langle y_n, z \rangle \rangle; \perp$
Length: $length : x \equiv x = \langle x_1, \ldots x_n \rangle \rightarrow n; zx = \phi \rightarrow 0; \perp$
Add, Subtract, Multiply, Divide:
$+ : x \equiv x = \langle y, z \rangle \wedge y, z \; are \; numbers \; \rightarrow y + z; \perp$
$- : x \equiv x = \langle y, z \rangle \wedge y, z \; are \; numbers \; \rightarrow y - z; \perp$
$* : x \equiv x = \langle y, z \rangle \wedge y, z \; are \; numbers \; \rightarrow y * z; \perp$
$\div : x \equiv x = \langle y, z \rangle \wedge y, z \; are \; numbers \; \rightarrow y \div z; \perp (where \; y \div 0 = \perp)$
Transpose:
$trans : x \equiv x = \langle \phi, \ldots, \phi \rangle \rightarrow \phi; x = \langle x_1, \ldots x_n \rightarrow \langle y_1, \ldots y_m \rangle; \perp$
$where \; x_i = \langle x_{i1}, \ldots x_{im} \; and \; y + j = \langle x_{1j}, \ldots x_{nj} \rangle, 1 \leq i \leq n, 1 \leq j \leq m$
And, Or, Not:
$and : x \equiv x = \langle T, T \rangle \rightarrow T; x = \langle T, F \rangle \vee x = \langle F, T \rangle \vee x = \langle F, F \rangle \rightarrow F; \perp$
...
Append left/right:
$appendl : x \equiv x = \langle y, \phi \rangle \rightarrow \langle y \rangle; x = \langle y, \langle z_1, \ldots, z_n \rangle \rangle \rightarrow \langle y, z_1, \ldots, z_n \rangle; \perp$
$appendl : x \equiv x = \langle \phi, z \rangle \rightarrow \langle z \rangle; x = \langle \langle y_1, \ldots, y_n \rangle, z \rangle \rightarrow \langle y_1, \ldots, y_n, z \rangle; \perp$
Right selectors; right tail: $1r : x \equiv x = \langle x_1, \ldots, x_n \rangle \rightarrow x_n; \perp$
$2r : x \equiv x = \langle x_1, \ldots, x_n \rangle \rightarrow x_{n-1}; \perp$
... $tlr : x \equiv x = \langle x_1 \rangle \rightarrow \phi; x = \langle x_1, \ldots, x_n \rangle \wedge n \geq 2 \rightarrow \langle x_1, \ldots, x_{n-1} \rangle; \perp$
Rotate left/right: $rotl : x \equiv x = \phi \rightarrow \phi; x = \langle x_1 \rangle \rightarrow \langle x_1 \rangle; x = \langle x_1, \ldots x_n \rangle \wedge$
$n \geq 2 \rightarrow \langle x_2, \ldots, x_n, x_1 \rangle; \perp$
...

2.1.4 Functional Forms

Functional form **F**: an expression denoting a function; that functions depends on the functions or objects which are the parameters of the expression.

Composition: $(f \circ g) : x \equiv f : (g : x)$
Construction: $[f_1, \ldots, f_n] : x \equiv \langle f_1 : x, \ldots f_n : x \rangle$
Condition:
$(p \to f; g) : x \equiv (p : x) = T \to f : x; (p : x) = F \to g : x; \perp$
Constant: $\bar{x} : y \equiv y = \perp \to \perp; x$ ($\bar{x}$ denotes is a functional form, the constant function of x)
Insert:
$/f : x \equiv x = \langle x_1 \rangle \to x_1; x = \langle x_1, \ldots, x_n \rangle \wedge n \geq 2 \to f : \langle x_1, /f : \langle x_2, \ldots, x_n \rangle \rangle; \perp;$
Extension for unique right unit (identity element): $/f : \phi = u_f$
Apply to all:
$\alpha f : x \equiv x = \phi \to \phi; x = \langle x_1, \ldots, x_n \rangle \to \langle f : x_1, \ldots, f : x_n \rangle; \perp$
Binary to Unary: $(bu\ fx) : y \equiv f : \langle x, y \rangle\ (bu\ +\ 1) : x = 1 + x$
While:
$(while\ p\ f) : x \equiv p : x = T \to (while\ pf) : (f : x); p : x = F \to x; \perp$

2.1.5 Definitions

Definitions: $\mathbf{Def}\ l \equiv r$
where the left side is an unused function symbol and the right side is a functional form (which may depend on l)
A set of D is *well-formed*, if no two left sides are the same.

$Def\ last1 \equiv 1 \circ reverse$
$Def\ last \equiv null \circ tl \to 1; last \circ tl$

$last : \langle 1,2 \rangle \equiv$

definition of last $\Rightarrow$	$(null \circ tl \to 1; last \circ tl) : \langle 1,2 \rangle$
action $(p \to f; g)$	$last \circ tl : \langle 1,2 \rangle$
	since $null \circ tl : \langle 1,2 \rangle = null : \langle 2 \rangle = F$
action $f \circ g \Rightarrow$	$last : (tl : \langle 1,2 \rangle)$
definition of $tl \Rightarrow$	$last : \langle 2 \rangle$
definition of $last \Rightarrow$	$(null \circ tl \to 1; last \circ tl) : \langle 2 \rangle$
action $f \circ g \Rightarrow$	$1 : \langle 2 \rangle$
	since $null \circ tl : \langle 2 \rangle = null : \phi = T$
definition of selector $\Rightarrow$	2

Remark: Substituting the name of a (recursive) function by its body is called *unfolding*.

2.2 Semantics of FP Programs

An FP-system is determined by the choice of the following sets:

- The set of atoms A (which determines the set of objects)
- The set of primitive functions P
- The set of functional forms $\mathbf{F}$
- A well formed set of definitions D.

Reduction Semantics:
Computation of $f : x$ for any function f (primitive, functional form, definition, none of these) and any object x

Remark: Discrimination between syntactic structures and semantics is realized hear partially by using different names for the sets.

Example: Factorial
$Def\ ! \equiv eq0 \rightarrow \bar{1}; \times \circ [id, ! \circ sub1]$
where
$Def\ eq0 \equiv eq \circ [id, \bar{0}]$
$Def\ sub1 \equiv - \circ [id, \bar{1}]$

(Proving the semantics: see theorems and algebra of FP programs in sect. 12 of Backus ,1978)

2.2.1 Proof of Correctnes by Evaluation

Example:
$Def\ length1 \equiv eq \circ [id, \phi] \to \bar{0}; / + \circ \alpha\bar{1}$
to show: $length1 : \langle x_1, \ldots, x_n \rangle$ is 0 for the empty sequence and n for a sequence of length n (which does not contain a $\perp$).

- $n = 0$
  $length1 : \phi \Rightarrow eq \circ [id, \phi] \to \bar{0}; / + \circ \alpha\bar{1} : \phi$
  $\Rightarrow \bar{0} : \phi$ (because $eq \circ [id, \phi] : \phi \Rightarrow T$)
  $\Rightarrow 0$

- $n \geq 1$
  $length1 : \langle x_1, \ldots, x_n \rangle \Rightarrow eq \circ [id, \phi] \to \bar{0}; / + \circ \alpha\bar{1} : \langle x_1, \ldots, x_n \rangle$
  $\Rightarrow / + \circ \alpha\bar{1} : \langle x_1, \ldots, x_n \rangle$ (because $eq \circ [id, \phi] : \langle x_1, \ldots, x_n \rangle \Rightarrow F$)
  $\Rightarrow /+ : \langle \underbrace{1, \ldots, 1}_{n\ times} \rangle$
  $\Rightarrow + : \langle 1, / + \langle \underbrace{1, \ldots, 1}_{n-1\ times} \rangle \rangle \Rightarrow \ldots \Rightarrow + \langle 1, + \langle 1, \ldots \rangle \ldots \rangle$
  $\Rightarrow\ n$

(Remember constant functions: $\bar{(}1) : x = 1$)

2.2.2 Complete Induction

For recursive function definitions: complete induction

$Def\ length \equiv eq \circ [id, \phi] \rightarrow \bar{0}; + \circ [\bar{1}, length \circ tl]$

Induction Hypothesis: For sequences $x \equiv \langle x_n, \ldots, x_1 \rangle$ holds $length : x$ is n.

Base case: $n = 0$
$length : \phi \Rightarrow 0$ (because $eq \circ [id, \phi] : \phi \Rightarrow T$)

Induction step: $n \rightarrow n + 1$
to show: $length : \langle x_{n+1}, x_n, \ldots, x_1 \rangle = n + 1$
$+ \circ [\bar{1}, length \circ tl] : \langle x_{n+1}, x_n, \ldots, x_1 \rangle$ (because $eq \circ [id, \phi] : \langle x_{n+1}, x_n, \ldots, x_1 \rangle \Rightarrow F$) $\Rightarrow + \circ [\bar{1}, length] : \langle x_n, \ldots, x_1 \rangle$
$\Rightarrow + \circ \langle \bar{1} : \langle x_n, \ldots, x_1 \rangle, length : \langle x_n, \ldots, x_1 \rangle \rangle$
$\Rightarrow + : (\langle \bar{1} : \langle x_n, \ldots, x_1 \rangle, length : \langle x_n, \ldots, x_1 \rangle \rangle)$
$\Rightarrow 1 + n$ (by of assumption)

More details about semantics and proofs for FP Systems will follow later.

3 Mathematical Functions and First Steps in ML

3.1 Characteristics of Functional and Imperative Programs

(Pepper, p. 3)

Functional Program:
(1) E/A Relation, that is mapping of input-data to output-data
(2) “time-less”: independent of the current state of the executing machine
(3) abstract, mathematical formulation
(4) Lisp, ML, Haskell, Miranda, Opal, ...

Imperative Program:
(1) Sequence of commands for transforming input-data into output-data
(2) Effect of a command depends on the current state of the machine; to understand a program, one has to follow its steps in time
(3) concrete relation to what the computer does
(4) Algol, Fortran, Pascal, C (C++, Java, Smalltalk)

- Two classes of declarative languages (in contrast to procedural, imperative languages): logical and functional
- Specifying “what” to do rather than “how” to do it
- typical use: artificial intelligence
- Prolog was designed for natural language processing and automated reasoning
- ML was designed for implementing theorem provers
- Relation of declarative languages and specification languages (automatic programming, program transformation)

3.2 Basic Mathematical Concepts: Sets, Functions, Terms, Expressions

3.2.1 Elementar Concepts of 'Sets'

- $x \in M$: member test, is x contained in set M?
- $A \subseteq B$: subset, are all elements of set A contained in set B
- $\emptyset$, $\{\ \}$: empty set
- $\{x_1, \ldots, x_n\}$: enumeration
- $\{x \mid p(x)\}$: set comprehension, set of all elements with attribute p
- $A \cup B$: union (formal definition: $A \cup B = \{x \mid x \in A \vee x \in B\}$)
- $A \cap B$: intersection
- $A \setminus B$: set difference, all elements of A which are not also in B
- Hierarchical conception of sets (Russel paradox)
- extensional definition by enumeration vs. intensional definition by comprehension
- set vs. bag vs. sequence/list

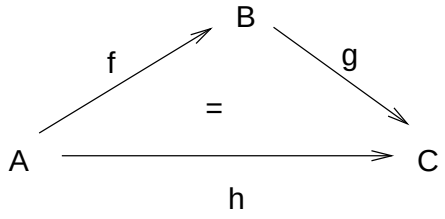
3.2.2 Tupel

- $A \times B$: pair, special case of product, set of all ordered pairs $\langle a, b \rangle$ with $a \in A$ and $b \in B$
  projections $\pi_1\langle a, b\rangle = a$, $\pi_2\langle a, b\rangle = b$
- $A_1 \times \ldots \times A_n$: product with projections $\pi_1, \ldots, \pi_n$, empty product: $\langle\ \rangle$
- A^n: power, n-th product of A $A \times \ldots \times A$
- A^*: sequence, set of all sequences of elements from A, $A^0 \cup A^1 \cup A^2 \cup \ldots$ (the infinite set of all words over A)
  remember: the set of all subsets of A is called power-set and: $|\mathcal{P}(\mathcal{A})| = \in^{\setminus}$ for $|A| = n$.

3.2.3 Relations and Functions

- $R \subseteq A \times B$: is a binary **relation** between sets A and B; R is a set of pairs
- $A \to B$: is a **function space**, the set of all functions from A to B
- $f = \langle D_f, W_f, R_f\rangle$: is a function with a domain ("Definitionsbereich") D_f, a codomain ("Wertbereich") W_f and a function graph $R_f \subseteq D_f \times W_f$. The function graph must be uniquely defined for every element of D_f ("rechtseindeutig"), that is: $\langle a, b_1\rangle, \langle a, b_2\rangle \in R_f \Rightarrow b_1 = b_2$.
- f maps x to y if $\langle x, y\rangle \in R_f$.
- D_f and W_f can be tuples!
- Some mathematicans call f as defined above a mapping and say that f is a function only if the co-domain is R. More typically, function and mapping are used as synonyms.

- A function f is called *partial* if $\pi_1(R) \subset D_f$ and *total* if $\pi_1(R) = D_f$. With $\hat{D} = \pi_1(R)$ and $\hat{W} = \pi_2(R)$ the elements of D and W which are really mapped by f are denoted.e "totalized" introducing a special element $\perp$. For a partial function $f = \langle D, W, R\rangle$ a total function $f^\perp = \langle D^\perp, W^\perp, R^\perp\rangle$ is defined as:
  - $D^\perp = D \cup \{\perp\}$ and $W^\perp = W \cup \{\perp\}$ where $\perp$ represents the "undefined" element.
  - $R^\perp$ is an extension of R such that for every element $x \in D^\perp$ which is not mapped into an element of W a pair $\langle x, \perp\rangle$ is introduced into R.
- $f(x)$: function application, returns y for $\langle x, y\rangle$ in y where y might be $\perp$.
- $g \circ f$: composition, a new function $h(x) = (g \circ f)(x) = g(f(x))$.
  For $f = \langle D_f, W_f, R_f\rangle$ and $g = \langle D_g, W_g, R_g\rangle$ holds: if $W_f \subseteq D_g$ then $h = \langle D_f, W_g, R_h\rangle$ with $R_h = \{\langle x, z\rangle \mid \exists\langle x, y\rangle \in R_f \wedge \langle y, z\rangle \in R_g\}$.



3.2.4 Terms and Expressions

- A constant $c \in D$ is a term. A variable ("place-holder" for a term) is a term. If $t_1 \in D_1$ and $t_2 \in D_2$ are terms and f is a function with domain $D_1 \times D_2$ then $f(t_1, t_2)$ is a term.
- Terms which do not contain variables are called *ground terms*.
- A variable can be *bound* to a value.
- An expression is basically a term, but might contain additional constructs (λ-expression, typed expression, ...).

3.3 ML: Value Declarations

- ML (Meta-Language) was developed 1974 for the programming of proof strategies.
- We will use the interpreter for *Standard ML* (`sml`).
- Interaction with the interpreter:
  every expression must be ended with a semicolon (`;`)

```
2 + 2;
> 4 : int;
Math.sqrt 2.0;
> 1.414213562 : real
```

- ML returns the *value* and the *type* of an expression (type inference !)
- Load an ML-program in the interpreter:

```
use "myprog.ml";
```

Value Declarations:

- A declaration gives something a name (values, types, signatures, structors, functors).
- Most names stand for values (numbers, strings, functions).
- Functions are values in ML!

3.3.1 Naming Constants

```
val seconds = 60;
> val seconds = 60 : int
val minutes = 60;
> val minutes = 60 : int
val hours = 24;
val hours = 24 : int
seconds * minutes * hours;
> 86400 : int
it div 24;
> 3600 : int
val secsinhour = it;
> val secsinhour = 3600 : int
val pi = 3.14159;
> val pi = 3.14159 : real
val r = 2.0;
> val r = 2.0 : real
val area = pi * r * r;
> val area = 12.56636 : real
```

- `it` stores the value of the last expression typed at top level.
- `it` can be saved

3.3.2 Function Declarations

```
fun area (r) = pi*r*r;
> val area = fn : real -> real
fun area2 r = pi*r*r;
> val area2 = fn : real -> real
```

- Declaration with keyword `fun`.
  `area` is the function name.
  `r` is the **formal parameter**
- left side: function head, right side: function body
- The **value** of a function is printed as `fn`. Functions are **abstract values**, their inner structure is hidden.

```
area(2.0);
> 12.56636 : real
area 1.0;
> 3.14159 : real
```

3.3.3 Comments

- `(* comment *)`
- can extend over several lines
- can be nested
- can be inserted nearly everywhere

```
fun area r = (* area of circle with radius r *)
      pi*r*r;
```

If the code and the comments disagree, then both are probably wrong. (N. Schryer)

3.3.4 Redeclaring Names

- Value names are called **variables**.
- In contrast to imperative languages variables *cannot be updated*!
- A name can be reused for another purpose.
- A re-declaration does *not* affect existing uses of the name. (different in Common Lisp!)
- The set of bindings visible at any point is called **environment**.
- Permanence of names: **static binding**
  ↪ redeclaring a function cannot damage the system, the library, or a program!
- When a function is modified, one must recompile!

```
val pi = 0.0; (* redeclaration of pi *)
> val pi = 0.0 : real
area(1.0); (* refers to the original environment *)
> 3.14159 : real
```

3.3.5 Identifiers in Standard ML

- **Alphabetic names**: must beginn with a letter, can be followed by letters, digits, underscores, primes (single quotes)
  mathematicias like variables called `x, x', x''`
- avoid ML keywords:
  `abstype and andalso as case datatype do else end eqtype exception fn fun functor handle if in include infix infixr let local nonfix of op open orelse raise rec sharing sig signature struct structure then type val where while with withtype`
- **Symbolic names**: consist of
  `! % & $ # + - * / : < = > ? @ \ ~ ` ^ |`
- Reserved symbols `: | = >= -> # :>`
- Names are known as **identifiers**. An identifier can simultaneously denote a value, a type, a structure, a signature, a functor, and a record field.

4 Basic Datatypes and Lists in ML

4.1 Numbers, Character Strings, and Truth Values

4.1.1 Arithmetic

- `type int` (in some ML systems with unlimited precision)
  unary minus: `~5` integer operations: `+ - * div mod` (all infix, parantheses where necessary)
  `(((m*n)*k) - (m div j)) + j`
- `type real` (decimal point or E notation or both)
  `~1.2E12`: $-1.2 \cdot 10^{12}$
  real operations: `+ - *` (overloaded built-in functions!) `/` (all infix)
- Function applications binds more tightly than infix operators.
- Arithmetic and the **Standard Library**: Structure `Int` contains functions such as `abs`, `min`, `max`, `sign`; Structure `Real` contains analogous and additional functions, especially conversion functions; Structure `Math` contains higher mathematical functions on real numbers.

```
Int.min(7, Int.sign 12);
> val it = 1 : int
- ~(~1);
val it = 1 : int
```

Type `int` and `real`	**Nubmers**
`~ : num -> num`	unary minus
`+ - * : num * num -> num`	addition, subtr., multipl.
`abs : num -> num`	abolute value
`/ : real * real -> real`	real division
`div mod : int * int -> int`	integer quotient and remainder
`< > <= >= :`	relations for
`numtext * numtext -> bool`	int, real, char, string
`real : int -> real`	coercion to nearest real
`round : real -> int`	coercion to nearest int
`floor : real -> int`	coercion to least int
`ceil : real -> int`	coercion to greatest int
`trunc : real -> int`	coercion to absolute greatest int

```
- open Math;
opening Math
  type real = ?.real
  val pi : real
  val e : real
  val sqrt : real -> real
  val sin : real -> real
  val cos : real -> real
  val tan : real -> real
  val asin : real -> real
  val acos : real -> real
  val atan : real -> real
  val atan2 : real * real -> real
  val exp : real -> real
  val pow : real * real -> real
  val ln : real -> real
  val log10 : real -> real
  val sinh : real -> real
  val cosh : real -> real
  val tanh : real -> real
```

Remarks:

- Structures will be introduced in a later section.
- `?.real` represents the built-in type `real`.
- Types can be redefined!

```
- type real = int;
type real = int
- 1 + 1;
val it = 2 : int
- 1.0 + 1.0;
val it = 2.0 : ?.real
- (1:real) + 1;
val it = 2 : real
open Real; (* ... *)
- 1 + 1;
stdIn:28.1-28.6 Error: operator and operand don't agree [literal]
  operator domain: real * real
  operand:         int * int
  in expression:
    1 + 1
- 1.0 + 1.0;
val it = 2.0 : real
```

4.1.2 Type Constraints

- ML can deduce the types in most expressions from the types of the functions and constants in it.
- But: some built-in functions are overloaded (having more than one meaning)!
  Example: $+$, $-$, $*$ are defined for integers and reals.
- If possible: infer type of an overloaded function from the context; occasionally types must be stated explicitely.
- Type constraints can appear almost everywhere (argument, result, body, within the body)

```
fun square x = x*x;
> Error - Unable to resolve overloading for *
fun square(x : real) = x*x; (* specify arg type *)
> val square = fn : real -> real
fun square x : real = x*x; (* specify res type *)
> val square = fn : real -> real
fun square x = x*x : real; (specify body type *)
> val square = fn : real -> real
```

4.1.3 Strings and Characters

- String constants in double quotes: `"How now! a rat?"`
- Characters: hashmark followed by a string of length 1: `#"a"`
- Special characters: Escape sequences.
- In the Standard Library are structures `String`, `Substring`, and `Char` with operations for these types.

`\n`	newline
`\t`	tab
`\"`	double quote
`\\`	backslash
`\` followed by white-space	continue a string across a line-break

type `char` and `string` and `substring`	Characters/Strings
`^ : string * string -> string`	concatenation
`concat : string list -> string`	conc. of a list of strings
`explode : string -> char list`	coercion to list of chars
`implode : char list -> string`	coercion to string
`str : char -> string`	coercion to 1-char string
`size : string -> int`	number of chars
`substring : string * int * int -> string`	substring, pos, size
`chr : int -> char`	char with given ASCII-code
`ord: char -> int`	ACII-code of char

```
fun digit i = chr(i + ord #"0");
> val digit = fn : int -> char
fun digit i = String.sub("0123456789", i);
> val digit = fn : int -> char
str(digit 5);
> val "5" : string
```

4.1.4 Truth Values and Conditional Expressions

Boolean Operations

- logical or `orelse`
- logical and `andalso`
- logical negation `not`
  `not true  not(true)`
- Functions that return boolean values are called **predicates**.
- Operators `orelse` and `andalso` behave differently from ordinary functions: the second operand is evaluated only if necessary (sequential behavior)

```
fun isLower c = #"a" <= c andalso c <= #"z";
```

datatype `bool = true | false`	Truth Values
`not : bool -> bool`	logical negation
`= <> : ''a * ''a -> bool`	equality test

Remark: With `'`a *we denote any type and with* `''`a *we denote a type with equality.*
Remark: In ML `bool` *is a datatype, namely an enumeration type.*

Conditional Expression:
`if E then E1 else E2`

- the `else` part is mandatory!
- We will here more about conditional expressions (and the McCarthy conditional `cond`) in a later section.

```
fun sign (n) =
  if n > 0 then 1
  else if n=0 then 0
       else (* n<0 *) ~1;
```

4.2 Tuples

- The ordered collection of n values is called n-**tuple**. A 2-tuple is called **pair**.
- Components of a tuple can be aribtrary values (expressions, other tuples, ...).
- In classical ML $(x_1, \ldots, x_{n-1}, x_n)$ was an abbreviation for $(x_1, \ldots, (x_{n-1}, x_n) \ldots)$
- With functions, tuples give the effect of multiple arguments and results.

```
(2.5, ~1.2);
> (2.5, ~1.2) : real * real
val zerovec = (0.0, 0.0);
> val zerovec = (0.0, 0.0) : real * real
val a = (1.5, 6.8);
> val a = (1.5, 6.8) : real * real
val b = (3.6, 0.9);
> val b = (3.6, 0.9) : real * real
fun lengthvec (x, y) = Math.sqrt(x*x + y*y);
val lengthvec = fn: real * real -> real
lengthvec a;
> 6.963476143 : real
lengthvec(1.0, 1.0);
> 1.414213562 : real
fun negvec (x, y) : real*real = (~x, ~y);
> val negvec = fn : real * real -> real * real
type vec = real*real;
```

- `sqrt` is a function from the Math library.
  It constraints the overloaded operator to type `real`.
- For `negvec` a type constraint must be given because ~ is overloaded.
- Vectors have all the rights of built-in values (like integer): can be arguments and results of functions, can be given names.
- Type declaration is possible, also.

4.2.1 Functions with multiple arguments and results

```
fun average(x,y) = (x+y)/2.0;
> val average = fn : (real * real) -> real
```

- Strictly speaking, every ML function has one argument and one result. With tuples, functions can have any number of arguments and results.
- **Currying** gives the effect of multiple arguments (introduced lin a later lesson).
- Vectors can be paired (combined).
- In `addvec` (below) `vec` constrains + to real numbers.
  The ML system may abbreviate `(real * real)` writing the name of the declared type `vec`. (see above).
  `addvec` takes: one argument (a pair of pairs of reals), two argumens (each a pair of reals), four arguments (real numbers, oddly grouped)

```
((2.0, 3.5), zerovec);
> val it = ((2.0,3.5),(0.0,0.0)) : (real * real) * (real * real)
fun addvec ((x1,y1),(x2,y2)) : vec = (x1 + x2, y1 + y2);
> val addvec = fn : (real * real) * (real * real) -> vec
fun subvec(v1, v2) = addvec(v1, negvec v2);
> val subvec = fn : (real * real) * (real * real) -> vec
fun distance (v1, v2) = lengthvec(subvec(v1, v2));
> val distance = fn : (real * real) * (real * real) -> vec
fun distance pairv = lengthvec(subvec pairv);
```

4.2.2 Selecting Components of a Tuple

- A function is defined on a **pattern**, such as `(x, y)` and refers to the components of its arguments though the pattern variables x and y.
- A `val` declaration may also match a value against a pattern: each variable in the pattern refers to a corresponding component.

```
fun scalevec (r, (x, y)) : vec = (r*x, r*y);
> val scalevec = fn : real * (real * real) -> vec
scalevec(2.0, a);
> val it = (3.0, 13.6) :vec
val (xc, yc) = scalevec(4.0, a);
> val xc = 6.0 : real;
> val yc = 27.2 : real;
```

4.2.3 0-tuple and type 'unit'

- The 0-tuple `( )` has no components and is called **unity**. It is the sole value of **type unit** and serves as placeholder in situations where no data needs to be conveyed.
- Procedural programming in ML: functions which return `unit`.
  `use: string -> unit`
  Functions with argument `unit`: only evaluation of body (delayed evaluation for infinite lists, see later section)

4.3 Records

- The last and most complex basic type offered by ML are Records.
- A record is a tuple whose components (fields) have labels.
  `label = { components }`
- Because of the labels, in records the sequence of entries is arbitrary.

```
val mr_jones = {name="Jones", age=25, salary=15300};
```

- As for tuples, we can give **patterns** for records.
- If we need not all fields, we can write three dots.
- **Field selectors** are noted as `#label`.

```
{salary=salaryJones, ...} = mr_jones;
> val salaryJones = 15300 : int
#age mr_jones;
> 25 : int
```

- A tupel $(x_1, x_2, \ldots, x_n)$ corresponds to a record of the form:
  {1=x1, 2=x2, ..., n=xn}.
- Selectors are defined for tuples also:
  `#2 ("a", "b", 3, false)`
- We can declare **record types**
- and functions on these types.
  The type constraint is mandatory!

```
type employee = {name : string,
                 age : int,
                 salary : int};
> type employee
fun monthlySalary(e : employee) = #salary / 12;
fun monthlySalary({salary, ...} : employee) = salary / 12;
```

- Record-types are represented by their components. Therefore types with identical components are equivalent!
- Because (the functional core of) ML does not allow updates, it is not possible to change components of record-values. Nondestructive changes can be performed by generating a new value copying some components and replacing others.

```
- type employee = {name : string, age : int, salary : int};
type employee = {age:int, name:string, salary:int}
- val mrjones = {name="jones", age=25, salary=15300};
val mrjones = {age=25,name="jones",salary=15300}
  : {age:int, name:string, salary:int}
- fun name(e : employee) = "mr. " ^ #name e;
val name = fn : employee -> string
- type employee2 = {name : string, age : int, salary : int};
type employee2 = {age:int, name:string, salary:int}
- val mrdoe : employee2 = {name="doe",age=35,salary=30000};
val mrdoe = {age=35,name="doe",salary=30000} : employee2
- name(mrjones);
val it = "mr. jones" : string
- name(mrdoe);
val it = "mr. doe" : string
```

4.4 Infix Operators

- In Lisp, operators are prefix: `(+ (- 17 2) 5)`.
- Most functional languages let programmers declare their own infix operators.
- One can give a precedence directive (between 0 and 9) for `infix`.
  `infix 6 +; infix 7 *; infix 8 pow`
- Default for a newly defined infix operator is `0`.
- Operators defined as infix can be used as prefix with `op` or `nonfix`. Changing the infix status of established operators leads to madness!

```
op+(1,2);
> val it = 3 : int
nonfix +;
> nonfix +
+(1,2);
> val it = 3 : int

infix xor;
fun (p xor q) = (p orelse q) andalso not (p andalso q);
> val xor = fn : (bool * bool) -> bool
true xor false;
> true : bool;
```

Precedence of infixes (all but `::` and `@` associate to the left

```
7   / * div mod
6   + - ^
5   :: @
4   = <> <  >  <= >=
3   := o
0   before
```

4.5 A First Look at ML Datatype List and Pattern Matching

- Processing collections of items: Lists vs. Arrays
- Lists are dynamic datatypes: arbitrary number of elements
- Typically direct access is to the first (or last) element only.
- "Lists are easy to understand mathematically, and turn out to be more efficient than commonly thought." (Pauson, chap. 3)

- A **list** is a finite sequence of elements. The order of elements is significant and elements may appear more than once.
- In ML every element of a list must have the same type τ. This type can be of arbitrary complexity.
- The empty list `[ ]` or `nil` has the polymorphic type α list.

```
[1, 2, 3] : int list
[(1, "One"), (2, "Two"), (2, "Two")] : (int*string) list
[[3.1],[],[5.7, ~0.6]] : (real list) list
```

- The type operator *list* has a postfix syntax.
- It binds more thightly then `*` and `->`.
- `int * string list` is the same as `int * (string list)`!

datatype 'a list = nil | :: of 'a * 'a list	**Lists**
`@ : 'a list * 'a list -> 'a list`	concatenation
`length : 'a list -> int`	length
`rev : a'list -> 'a list`	reversal
`hd : 'a list -> 'a`	head
`tl : 'a list -> a' list`	tail
`null : 'a list -> bool`	empty test

The more complex, higher order functions on lists are introduced later!

4.5.1 Building a List

List Constructor: `'a :: 'a list`

```
[1, 2, 3] = 1 :: (2 :: (3 :: nil))

fun upto(m, n) =
  if m>n then [] else m :: upto(m+1, n)
```

Write the stepwise evaluation of this linear recursive function!

- In Lisp, lists are constructed with `(cons 'a list)` and a list is a nested cons-expression `(cons 1 (cons 2 (cons 3 nil)))`.
- In Prolog `[5 | [6]]` represents `[5, 6]` while in ML `[5 :: [6]]` represents `[[5,6]]`!!
- In many functional languages there is no special type `string`, but a string is represented as list of characters!
  ML provides `implode` and `explode` for conversion. (see above)

4.5.2 Fundamental List Functions: null, hd, tail

```
fun null [] = true
|   null (_::_) = false;
> val null = fn : 'a list -> bool

fun hd (x::_) = x;
> ***Warning: Patterns not exhaustive
> val hd = fn : 'a list -> 'a

fun tl (_::xs) = xs;
> ***Warning: Patterns not exhaustive
> val hd = fn : 'a list -> 'a list
```

- **Pattern Matching**:
  Remember patterns for tuples: `fun f (x, y)`, `type vec = real * real`
- The underscore represents a wildcard!
- The functions are polymorphic, allowing lists over arbitrary elements of type α.
- A function can consist of clauses, separated by a vertical bar. Each clause represents one argument pattern.
  Alternatively, a function with a conditional expression can be defined, which is often more complex to read.

```
fun prod [] = 1
| prod (n:ns) = n * (prod ns);

fun prod l = if null(tl(l)) then 1 else hd(l) * prod(tl(l));
```

5 Evaluation of Expressions and Recursive Functions

- An imperative program specifies commands to update the machine state. During execution, the state changes millions of times per second. Its structure changes, too: local variables are created and destroyed.
- In functional programming, there are no state changes. **Execution is reduction of an expression to its value, replacing equals by equals.**
- When a function is applied, as in $f(E)$, the argument E must be supplied to the body of f. If the expression contains several function calls, one must be choosen according to some **evaluation rule**.
- Two kinds of evaluation rules:
  - **call-by-value** or **strict evaluation** (used by ML)
  - **call-by-need** or **lazy evaluation** (typical for purely functional languages)
- When a function is called, the argument is substituted for the function's formal parameter in the body. The evaluation rules differ over when, and how many times the argument is evaluated.
- The formal parameter indicates where in the body to substitute the argument. The name of the formal paramter has no other significance and no significance outside the function definition.

Two critical cases for the different evaluation strategies:

```
fun sqr(x) : int = x*x; (* uses its argument twice *)
fun zero(x :int) = 0; (* ignores its argument *)
```

5.1 Call-by-Value, or Strict Evaluation

- To compute the value of $f(E)$ first compute the value of E.

Evaluation of `sqr(sqr(sqr(2)))`:

sqr(sqr(sqr(2))) $\Rightarrow$ sqr(sqr(2 $\times$ 2))
$\Rightarrow$ sqr(sqr(4))
$\Rightarrow$ sqr(4 $\times$ 4)
$\Rightarrow$ sqr(16)
$\Rightarrow$ 16 $\times$ 16
$\Rightarrow$ 256

Evaluation of `zero(sqr(sqr(sqr(2))))`:

zero(sqr(sqr(sqr(2)))) $\Rightarrow$ zero(sqr(sqr(2 $\times$ 2)))
$\Rightarrow$ zero(sqr(sqr(4)))
...
$\Rightarrow$ zero(256)
$\Rightarrow$ 0

Such waste!

5.2 Recursive Functions under Call-by-Value

```
fun fact n = if n=0 then 1 else n * fact(n-1);
fun facti(n, p) = if n=0 then p else facti(n-1, n*p);
```

- The first definition is called **linear** recursive. It corresponds to the natural, mathematical definition:

$$x! = \begin{cases} 0 & if\ x = 0 \\ x \times (x-1)! & else \end{cases}$$

  The call-by-value evaluation of a linear recursive function must depend on a stack because to evaluate a function call, the value(s) of its argument(s) must be calculated first!

- The second definition is a special case of linear recursion, called **tail recursion** (or iteration).
  Often, tail recursive forms of a linear recursion can be obtained by introducing an addition parameter which takes over the collection of values calculated so far (it replaces the stack).

- Good compilers can detect iterative forms of recursion! (See Optimization, Programm Transformation Techniques in Field and Harrison).

- Construction of the tail recursive function: detecting that by making use of the associative law of multiplication each multiplication can be done at once:

$$4 \times (3 \times fact(2)) = (4 \times 3) \times fact(2) = 12 \times fact(2)$$

- In a later section we will prove that $facti(n, p) = n! \times p$. If "collector" p is initially given the value 1 (identity element for multiplication), then $facti(n, p) = n!$.

fact(4)
$$\begin{aligned}
&\Rightarrow 4 \times fact(4-1)\\
&\Rightarrow 4 \times fact(3)\\
&\Rightarrow 4 \times (3 \times fact(3-1))\\
&\Rightarrow 4 \times (3 \times fact(2))\\
&\Rightarrow 4 \times (3 \times (2 \times fact(2-1)))\\
&\Rightarrow 4 \times (3 \times (2 \times fact(1)))\\
&\Rightarrow 4 \times (3 \times (2 \times (1 \times fact(1-1))))\\
&\Rightarrow 4 \times (3 \times (2 \times (1 \times fact(0))))\\
&\Rightarrow 4 \times (3 \times (2 \times (1 \times 1)))\\
&\Rightarrow 4 \times (3 \times (2 \times 1))\\
&\Rightarrow 4 \times (3 \times 2)\\
&\Rightarrow 4 \times 6\\
&\Rightarrow 24
\end{aligned}$$

facti(4,1)
$$\begin{aligned}
&\Rightarrow facti(4-1, 4 \times 1)\\
&\Rightarrow facti(3, 4)\\
&\Rightarrow facti(3-1, 3 \times 4)\\
&\Rightarrow facti(2, 12)\\
&\Rightarrow facti(2-1, 2 \times 12)\\
&\Rightarrow facti(1, 24)\\
&\Rightarrow facti(1-1, 1 \times 24)\\
&\Rightarrow facti(0, 24)\\
&\Rightarrow 24
\end{aligned}$$

5.3 Conditional Expressions

- The conditional expression permits definition of cases.
- Recursive functions *must* be defined with cases to obtain a non-recursive base-case (termination).
- The conditional expression does not correspond to the cond-expression (McCarthy conditional) such that $cond(E, E_1, E_2)$!

```
fun cond(p, x, y) : int = if p then x else y;
> val cond = fn : bool * int * int -> int
fun badf n = cond(n=0, 1, n*badf(n-1));
> val badf = fn : int -> int
```

badf(0) $\Rightarrow cond(true, 1, 0 \times badf(-1))$
$\Rightarrow cond(true, 1, 0 \times cond(false, 1, -1 \times badf(-2)))$
...

- ML's boolean infix operators `andalso` and `orelse` are not functions but stand for conditional expressions:
  `E1 andalso E2 == if E1 then E2 else false`
  `E1 orelse E2 == if E1 then true else E2`
- These operators evaluate E_2 only if necessary!
- When defining recursive functions, `andalso` or `orelse` can be used because they are abbreviations of conditional expressions!

```
fun even n = (n mod 2 = 0);
fun powoftwo n = (n=1) orelse (even(n) andalso powoftwo(n div 2));
```

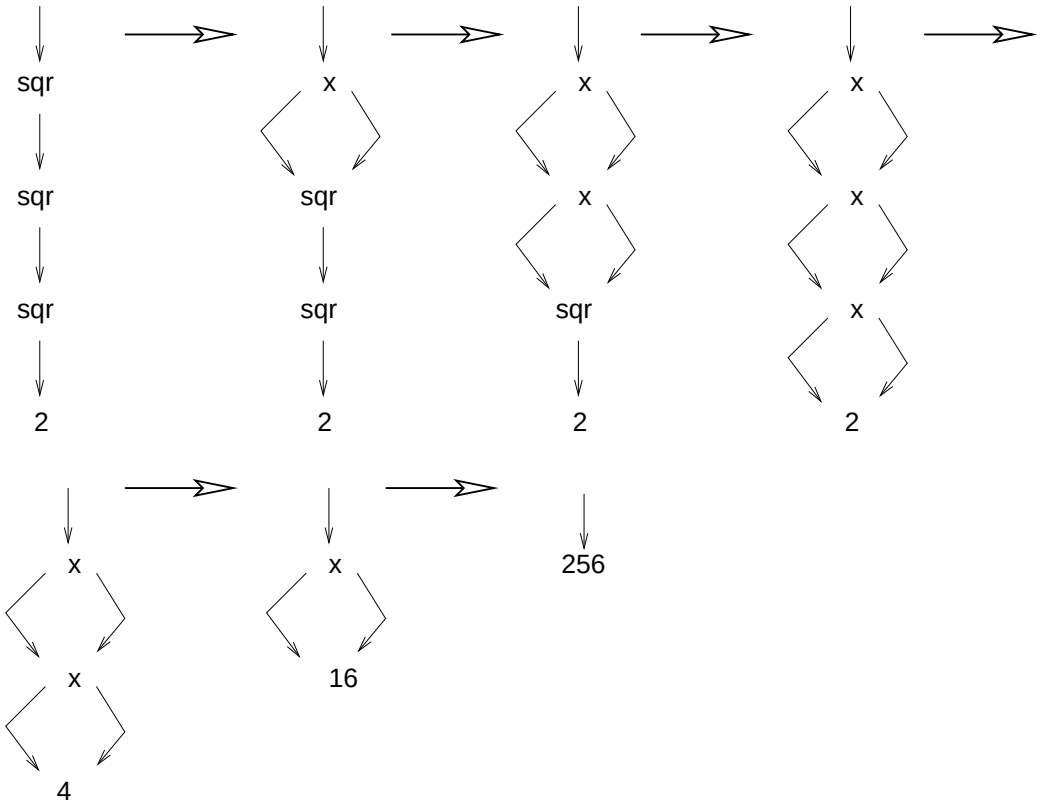
5.4 Call-by-Name

- Problems with call-by-value: superfluous evaluations, conditional expressions cannot be functions, users cannot define operators as `andalso`.
- The **call-by-name** rule: To compute the value of $f(E)$, substitute E immediately in the body of f. Then compute the value of the resulting expression.
- `zero(sqr(sqr(sqr(2)))) = 0` in one step!
- but: in `sqr(sqr(sqr(2)))` it duplicates the argument: `sqr(sqr(2)) × sqr(sqr(2))`.
- Arithmetic operations need special treatment. They must be applied to values, not to expressions.
  strict functions: to evaluate $E_1 \times E_2$, the expressions E_1 and E_2 must be evaluated first.
- If one looks at the evaluation of `sqr(sqr(sqr(2)))` call-by-name cannot be the evaluation rule we want! (it will reach a result finally, but with a lot of unnecessary steps)

sqr(sqr(sqr(2)))
$$\begin{aligned}
&\Rightarrow sqr(sqr(2)) \times sqr(sqr(2)) \\
&\Rightarrow (sqr(2) \times sqr(2)) \times sqr(sqr(2)) \\
&\Rightarrow ((2 \times 2) \times sqr(2)) \times sqr(sqr(2)) \\
&\Rightarrow (4 \times sqr(2)) \times sqr(sqr(2)) \\
&\Rightarrow (4 \times (2 \times 2)) \times sqr(sqr(2)) \\
&\ldots
\end{aligned}$$

5.5 Call-by-Need or Lazy Evaluation

- Call-by-need is like call-by-name but ensures that each argument is evaluated at most once!
- Rather than substituting an expression into the function's body, the occurences of the argument are linked by pointers. If the argument is ever evaluated, the value will be shared with its other occurences.
- The pointer structure forms a directed graph of functions and arguments. As a part of the graph is evaluated, it is updated by the resulting value: **graph reduction**. (see Field and Harrison)
- Lazy evaluation of `cond(E, E1, E2)` behaves like a conditional expression, provided that the tuple `(E, E1, E2)` is itself evaluated lazily. (Tuple formation must be viewed as a function.)
  The idea that data structures like `(E, E1, E2)` can be partially evaluated (either `E1` or `E2` but not both) leads to **infinite lists** (introduced in a later section).
- Lazy evaluation seems to give as the best of both worlds, but graph manipulations are expensive.



With lazy evaluation, reduction of `facti(n, p)` results in a **space leak**! (n is evaluated immediately for $n - 0$ but not p)

facti(4,1) $\Rightarrow facti(4-1, 4\times 1)$
$\Rightarrow facti(3-1, 3\times(4\times 1))$
$\Rightarrow facti(2-1, 2\times(3\times(4\times 1)))$
$\Rightarrow facti(1-1, 1\times(2\times(3\times(4\times 1))))$
$\Rightarrow 1\times((2\times(3\times(4\times 1))))$
...
$\Rightarrow 24$

5.6 Comparison of Strict and Lazy Evaluation

Paulson is in favour of strict evaluation:

- Strict evaluation is more natural, corresponds to the mathematical intuition of caluclating the result of an expression.
- Curch, the inventor of λ-calculus, provided a variant which banned constant functions like `zero`.
- Lazy evaluation needs much bookkeeping. It needs sophisticated concepts for efficient implementation. (Application of graph reduction to combinators, see Field and Harrison)
- Lazy programming languages are mostly purely functional, combination with imperative concepts (for input/output) is difficult. In lazy evaluation it cannot easily be predicted when a subexpression is evaluated (problem of writing reliable programs).

6 Local Declarations and Modules

6.1 Let-Expressions

Calculation the greatest common divisor: Euclid's algorithm:

```
fun gcd(m,n) =
  if m=0 then n
  else gcd(n mod m, m)
```

Example: gcd(247,1313) ⇒ gcd(78, 247) ⇒ gcd(13,78) ⇒ gcd(0,13) ⇒ 13

Calculation least terms for fraction n/d: for example: (5, 10) ⇒ (1, 2)

```
fun fraction (n,d) = (n div gcd(n,d), d div gcd(n,d));
```

Transparent but inefficient: `gcd(n, d)` is calculated twice.

Improvement:

```
fun divideboth (n, d, com) = (n div com, d div com);
fun fraction(n, d) = divideboth(n, d, gcd(n, d));
```

Better: use `let`-expressions!

- Declarations of names within an expression: `let D in E end;`
- `D` can be a compound declaration $D_1; D_2; \ldots; D_n$ (semicolons are optional).
- Evaluation: First `D` is evaluated and the result is named and only visible inside the let-expression; then `E` is evaluated and the value returned.
  For compound declarations $D_1; D_2; \ldots; D_n$, the name D_i is visible for all D_j, $i < j \leq n$.
- Let-expressions can be nested.

Example: real square roots
The Newton-Raphson method

```
fun findroot(a, x, acc) =  (* for x >=0 *)
  let val nextx = (a/x + x) / 2.0
  in if abs (x-nextx) < acc*x
     then nextx else findroot(a, nextx, acc)
  end;

fun sqroot a = findroot(a, 1.0, 1.0E~10);

sqroot 2.0;
> val it = 1.41421356237 : real
it * it;
> val it = 2.0 : real
```

- The `nextx` approximation is used several times and therefore defined as a let-expression.
- Arguments a and `acc` are passed unchanged in every recursive call of `findroot`.
  ↪ Make them global to `findroot` for efficiency and clarity, using a nested let-expression.
  Now `findroot` is not visible outside of `sqroot`!

```
fun sqroot a =
  let val acc = 1.0E~10;
      fun findroot x =
        let val nextx = (a/x + x) / 2.0;
        in if abs (x-nextx) < acc*x
           then nextx else findroot nextx
        end;
  in findroot 1.0
  end;
> sqroot = fn : real -> real
```

When not to use let-expressions:

```
let val a = f x
    val b = g x
in if a<b then a else b
end;
```

much more transparent:

```
fun min(a, b) : real = if a<b then a else b;
min(f x, g x);
```

6.2 Local declarations

`local D1 in D2 end`

- While `let` is frequently used, `local` is not.
- Sole purpose: hide a declaration, make `D1` private to `D2`.
- Declaration `D1` is only visible within `D2`.
- Since a list of declarations is regarded as one declaration, both `D1` and `D2` can declare any number of names.

```
local
  fun itfib(n, prev, curr) : int =
      if n=1 then curr
      else itfib(n-1, curr, prev+curr)
in
  fun fib(n) = itfib(n, 0, 1)
end;
> val fib = fn : int -> int
```

6.3 Simultaneous Declarations and Mutual Recursive Functions

- A simultaneous declaration defines several names at once.
- val Id1 = E1 and ... and Idn = En
- Evaluation of expressions `E1` to `En` and then declaration that identifies `Id1` to `Idn` have the corresponding values.
  ↪ order is immaterial because all expressions are evaluated before declarations take effect.

```
val pi = 4.0 * Math.atan 1.0
and e = Math.exp 1.0
and log2 = Math.ln 2.0;
> pi = 3.141592654 : real
> e = 2.718281828 : real
> log2 = 0.693147806 : real
```

```
(* The chimes of Big Ben *)
val one = "BONG";
val three = one^one^one;
val five = three^one^one; (* must be separate in this order *)

val one = three and three = one;
> val one = "BONG BONG BONG" : string
> val three = "BONG" : string
```

Declarations can be done at the same time!!! Consecutive declatations would give identical bindings for one and `three`!

Note the equivalence between declaring value tuples and simultanous declarations of values (not functions!):

```
- val sm = "bong";
val sm = "bong" : string
- val bg = "bang";
val bg = "bang" : string
- val(ld, sm) = (sm, bg);
val ld = "bong" : string
val sm = "bang" : string
(* ---------------------------*)
- val sm = "bong";
val sm = "bong" : string
- val bg = "bang";
val bg = "bang" : string
- val ld = sm and sm = bg;
val ld = "bong" : string
val sm = "bang" : string
```

- Functions are **mutually recursive** if they are declared recursively in terms of each other:

$$\{f_i = E_i | i = 1..n \; and \; E_i \; refers \; to \; some \; f_j, j \in \{1, \ldots, n\} \wedge j \neq i\}.$$

- Example: recursive descend parser
  has a function for each element of the grammar and most grammars are mutually recursive (e.g., an ML declaration can contain expressions and expressions can contain declarations)

Example: Summing the series for $\pi/4$

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \ldots + \frac{1}{4k+1} - \frac{1}{4k+3} \ldots (for \; k = 0 \ldots)$$

```
fun pos d = neg(d-2.0) + 1.0/d
and neg d = if d>0.0 then pos(d-2.0) - 1.0/d else 0.0;
> val pos = fn : real -> real
> val neg = fn : real -> real
4.0 * pos(201.0);
> val it = 3.15149340107 : real
4.0 * neg(8003.0);
> val it = 3.14134277853 : real
```

- Mutually recursive functions can often be combined into one function with help of an additional argument.

```
fun sum(d, one) =
  if d >0.0 then sum(d-2.0,~one) + one/d else 0.0;
(* sum(d, 1.0) returns pos(d), sum(d,~1.0) returns neg(d) *)
```

- A combination of goto- and assignement-statements (the worst of procedural code!) can be translated into a set of mutually recursive functions.

- Functional programs are referential transparent, yet can be totally opaque. It is often better to omit mutual recursion.

Transformation of a Mutual Recursion:

- Let $f_1, \ldots, f_n$ be a system of mutually recursive functions and function f_i has m_i arguments with $f_i(x_{i,1}, \ldots x_{i,m_i}) = r_i$.
- $f_1, \ldots f_n$ can be combined in a direct recursive function g with $m_1 + \ldots + m_n + 1$ argument: $g(x_{1,1}, \ldots, x_{1,m_1}, \ldots, x_{n,1}, \ldots, x_{n,m_n}, y) =$
  $if\ y_1 = 1\ then\ r'_1$
  $\ldots$
  $if\ y = n\ then\ r'_n$.
  r'_i is constructed from r_i such that each occurence of $f_j(t_1, \ldots, t_{m_j})$ is replaced by $g(x_{1,1}, \ldots, x_{1,m_1}, \ldots, t_1, \ldots, t_{m_j}, \ldots, x_{n,1}, \ldots, x_{n,m_n}, j)$.
  That is, the last argument of g marks which function f_j is simulated in the next call of g.

6.4 Modules: Structures and Signatures

- (Functional) programs are sets of data and functions.
- A module clusters related components, it defines its own datastructures and methods operating on it.
- Typically, a module is separated into an interface and an implementation.
- Advantages: independet parts of a complex program which can be implemented by different programmers (compiler can check, whether a module meets its interface specification).
  A module can be reused in different contexts, that is, combined with different other modules.
- Theory: H. Ehrig & B. Mahr (1990). Fundamentals of Algebraic Specification 2 – Module Specifications and Constraints. Springer.

Modules in ML:

- An ML **structure** combines related types, values, and other structures with an uniform naming discipline.
  `struct D end;` binding to a name: `structure name =`
- An ML **signature** specifies a class of structures by listing the name and type of each component.
  `sig body end;` binding to a name: `signature name =`
- Preview: ML provides functors, that is structures which take other structures as parameters. In this context we will introduce also the declaration of abstract types.

```
signature ARITH =
  sig
  type t
  val zero: t
  val sum: t * t -> t
  val diff: t * t -> t
  val prod: t * t -> t
  val quo: t * t -> t
  end;

structure Rational : ARITH =
  struct
  type t = int * int;
  val zero = (0, 1);
  fun sum ((n1, d1), (n2, d2)) = ((n1*d2 + n2*d1), (d1*d2)) : t;
  fun diff ((n1, d1), (n2, d2)) = ((n1*d2 - n2*d1), (d1*d2)) : t;
  fun prod ((n1, d1), (n2, d2)) = (n1*n2, d1*d2) : t;
  fun quo ((n1, d1), (n2, d2)) = (n1*d2, n2*d1) : t;
  end;


Rational.sum((1,2),(2,4));
val it = (8,8) : Rational.t
- Rational.prod((1,2),(2,4));
val it = (2,8) : Rational.t
```

The functions could be realized more elegantly, using `gcd`.

- If the purpose of a structure is to define a type, this type is commonly called `t`. Alternative: equality-types (`eqtype`) or abstract datatypes!
- If a structure is visible, its components are known by compound names (`Rational.sum`). Inside the structure body, the names are known unqualified.
- Structures look a bit like records, but their components can be not only values, but types, functions, and other structures (compare to classes in oo!).
- Structures are encapsulating environments.
- If a structure is declared without explicitely implementing a signature, a signature is inferred. (If Rational would have been declared without the use of `ARITH`, this signature would have been inferred without a name).
- A signature corresponds to the interface of a module and contains type checking information.
- Note that declaring that a structure implements a signature is realized using a `:` (like giving a type constraint)
- A signature can be implemented by different structures. For example, `ARITH` can be also implemented by `Nat` (or, more exotically, by `List` or `Bool`).
- If a value is implemented in a structure but not specified in the signature, it is hidden and cannot be used outside of the structure itself (good for helper functions).
- Of theoretical and practical interest is the question when structures can be combined savely! (see Ehrig & Mahr)

The interface/signature of the structure `Rational`

```
open Rational;
opening Rational
  type t = int * int
  val zero : t
  val sum : t * t -> t
  val diff : t * t -> t
  val prod : t * t -> t
  val quo : t * t -> t
```

7 Polymorphic Type Checking

- Two rigid positions:
  - Weakly typed languages like Lisp and Prolog give programmers the freedom they need.
  - Strongly typed languages like Pascal give programmers the security they need by restricting their freedom to make mistakes.
- Remark: See literature to **type theory** and the Curry-Howard-Isomorphism (proof as program).
- Middle way: Polymorphic type checking offers security and flexibility: Programs are not cluttered with type specifications since most type information is deduced automatically.

7.1 Types and Type Schemes

- A type denotes a collection of values.
- A function's argument type specifies which values are acceptable as arguments. A function's result type specifies which values could be returned.
- Strict functions: `div` expects two integers and returns an integer. If the divisor is 0, *no* result is returned (exception), that is, `div` is faithful to its type. (We will discuss exception handling in a later section.)
- Polymorphic functions: can have many types. ML polymorphism is based on type schemes, that is, patterns for types.

```
fun id x = x;
> val id = fn : 'a -> 'a
id 2;
> val it = 2 : int
fun f x = id x + 1;
> val f = fn : int -> int
f 3;
> val it = 4 : int
```

7.2 Type Inference

- ML can infer all types involved in a function declaration with little or no explicit type information.
- Type inference follows a natural but rigorous procedure:
  - Note the type of any constant.
  - Apply type checking rules for each form of expression.
  - Each variable must have the same type everywhere in the declaration.
  - The type of each overloaded operator (like +) must be determined from the context.

Example: Type checking rule for conditional expression:

$$\frac{A \vdash e : \mathit{bool} \quad A \vdash e' : \tau \quad A \vdash e'' : \tau}{A \vdash (\mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e'') : \tau}$$

otherwise, the expression is ill-typed.

Example: type checking for `facti`:

```
fun facti(n, p) = if n=0 then p else facti(n-1, n*p);

0:int --> n=0: ? * int -> bool --> n = int
1:int --> n-1: int * int -> int
n*p: int * ? -> ? --> p = int
argument type: int * int
satisfied by recursive call
return type: int (because of then p)

> val facti = fn : int * int -> int
```

7.3 Polymorphic Function Declarations

- If type inference leaves some types completely unconstrained then the declaration is polymorphic ("having many forms").
- Most polymorphic functions involve pairs, lists and other data structures.
- Type variables are traditionally small Greek letters ($\alpha, \beta, \gamma, \ldots$). In ML written as `'a, 'b, 'c, ....`
- A polymorphic type is a type scheme. Substituting types for type variables forms an instance of the scheme.

```
fun pairself x = (x, x);
> val pairself = fn : 'a -> 'a * 'a
pairself 4.0;
> (4.0, 4.0) : real * real
pairself 7;
> (7, 7) : int * int
val pp = pairself ("Help!", 999);
> val pp = (("Help!", 999), ("Help!", 999))
>     : (string * int) * (string * int)
fun fst (x, y) = x; (* projection *)
> val fst = fn : 'a * 'b -> 'a
fun snd (x, y) = y;
> val snd = fn : 'a * 'b -> 'b
fst pp;
> ("Help!", 999) : string * int
snd(fst pp);
> 999 : int
fun fstfst z = fst(fst z);
> val fstfst = fn : ('a * 'b) * 'c -> 'a
fstfst pp;
> "Help!" : string
```

outer fst: $\alpha \times \beta \to \alpha$
therefore, inner fst must have $\alpha \times \beta$ as result
inner fst: $(\alpha \times \beta) \times \gamma \to (\alpha \times \beta)$
Infer the type of `fun silly x = fstfst(pairself(pairself x));`

- R. Milner (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, *17*, 348–375.
  gives an algorithm for polymorphic type checking and proves that a type-correct program cannot suffer a run-time type error.

- The types inferred with the ML-algorithm are **principal**: as polymorphic as possible.

- Equality testing is polymorphic in a limited sense: it is defined for most, not all, types. ML provides a class of equality type variables to range over this restricted collection of types. (discussed in a later section)

- Overloading sits uneasily with polymorphism. It complicates the type checking algorithm and frequently forces the programmer to write type constraints.
  Remember: ML has a small set of overloaded built-in functions. Programmers cannot introduce further overloading.

7.4 The Type Checking Algorithm 'W'

(see Field & Harrison, chap. 7)

- $\mathcal{W}$ is a slight variant of Milner's algorithm.
- Syntactic and semantic issues:
  (1) **well-typed** expressions: there must be defined a syntactic typing scheme which assigns a unique, most general type to each valid expression.
  (2) **semantically sound** typing scheme: each expression which is syntactically well-typed is also semantically free from type violations
  (3) **syntactical soundness** of the algorithm: if it succeeds in finding a type for an expression then that expression is well-typed.
  (4) **completeness** of the algorithm: if an expression has a well-typing then the algorithm will succeed in finding a typing for it which is at least as general.
- $\mathcal{W}$ is based upon Robinson's unification algorithm.

7.4.1 Most general unifiers

- There is an unification algorithm $\mathcal{V}$ which takes any pair of expressions σ, τ over some alphabet of variables such that:
  Either $\mathcal{V}(\sigma, \tau)$ succeeds yielding a substitution U with the properties that
  - $U\sigma = U\tau$ (U unifies σ and τ)
  - If R unifies σ and τ, then for some substitution S, $R = SU$ (U is a most general unifier)
  - U involves only variables occuring in σ or τ.

  or else $\mathcal{V}(\sigma, \tau)$ fails.

(For first order logic and for type expressions, there exists a unique most general unifier.)

Substitution: $[\sigma_1/\alpha_1, \ldots, \sigma_n/\alpha_n]$ of variables α by terms (here type expressions) σ.

Examples:
$\mathcal{V}(\alpha, \alpha)$ succeeds with $U = I$ (identity substitution $S = [\,]$)
$\mathcal{V}(\alpha \to \beta, int \to int)$ succeeds with $U = [int/\alpha, int/\beta]$
$\mathcal{V}(\alpha \to bool, int \to int)$ fails.

7.4.2 Disagreement Pairs

Implementation of $\mathcal{V}$ using **disagreement pairs**:

- For simplification: write every type expression prefix, with $\times$ and $\to$ as type operations
  $\alpha \to (\beta \times \gamma) \Rightarrow\to (\alpha, \times(\beta, \gamma))$
- For identical expressions $e = e'$ the disagreement pair is empty, $D = \pi$.

$D(T_i(\sigma_1, \ldots, \sigma_{n(i)}), T_j(\tau_1, \ldots, \tau_{n(j)})) =$

if $T_i \neq T_j$

then $(T_i(\sigma_1, \ldots, \sigma_{n(i)}), T_j(\tau_1, \ldots, \tau_{n(j)}))$

else if $n(i) = 0$

then π

else $D'(1)$

where $D'(k) =$ **if** $k = n(i)$

then $D(\sigma_k, \tau_k)$

else if $D(\sigma_k, \tau_k) = \pi$

then $D'(k+1)$

else $D(\sigma_k, \tau_k)$

Examples:
$D(\rightarrow (int, int), \rightarrow (int, int)) = \pi$
$D(\rightarrow (int, int), \rightarrow (\alpha, int)) = (int, \alpha)$
$D(\alpha, \rightarrow (\alpha, \beta)) = (\alpha, \rightarrow (\alpha, \beta))$
$D(\rightarrow (\gamma, \rightarrow (int, \beta)), \rightarrow (\rightarrow (\alpha, int), int)) = (\gamma, \rightarrow (\alpha, int))$
(cannot be unified, because of $(\rightarrow (int, \beta), int)$)

7.4.3 Algorithm 'V'

We extend $\mathcal{V} : term \times term \rightarrow substitution$ to $unify : substitution \times term \times term \rightarrow substitution$ and start with the identity substitution I.

$\mathcal{V}(e, e') = unify(I, e, e')$

$unify(S, e, e') =$ **if** $Se = Se'$

then S

else let $(u, v) = D(Se, Se')$ **in**

if u is a variable not occuring in v

then $unify([v/u]S, e, e')$

else if v is a variable not occuring in u

then $unify([u/v]S, e, e')$

else $fail$.

- Composition of substitutions S and T: ST.
  Example: $[v/u]S$ replacing variable u by term v in S
- Application of a substituion S to an expression τ: $S\tau$.
  For each variable u in τ which occurs as v/u in S: replace u by v.
- If neither u nor v are variables, the unification algorithm fails.
- If u is a variable occuring in v (or the other way round), then there is danger of cyclic substitutions ($\mathcal{V}$ might not terminate!).
  Check for possible cycles: **occurs check**. (problem: this algorithm is incomplete!, additional intelligence is needed)

Example:
$\mathcal{V}(\alpha \to \beta, \beta \to \gamma)$
$= unify(I, \alpha \to \beta, \beta \to \gamma)$
$= unify([\beta/\alpha], \alpha \to \beta, \beta \to \gamma)$
since $D(\alpha \to \beta, \beta \to \gamma) = (\alpha, \beta)$
$= unify([\gamma/\beta][\beta/\alpha], \alpha \to \beta, \beta \to \gamma)$
since $D([\beta/\alpha](\alpha \to \beta), [\beta/\alpha](\beta \to \gamma)) = D(\beta \to \beta, \beta \to \gamma) = (\beta, \gamma)$
$= [\gamma/\beta][\beta/\alpha]$
since $[\gamma/\beta][\beta/\alpha](\alpha \to \beta) = [\gamma/\beta][\beta/\alpha](\beta \to \gamma) = \gamma \to \gamma$
$= [\gamma/\beta, \beta/\alpha]$

Realization in ML:

- Representing type expressions:

```
datatype typeterm = INT | REAL | BOOL | STRING |
                    --> of typeterm * typeterm |
                    ** of typeterm * typeterm |
                    V of int;
infix -->;
infix **;
```

- Five rules for algorithm $\mathcal{V}$ (most general unifier, mgu), realized with pattern-matching:
  - `mgu (sub, l1 --> r1, l2 --> r2)`:
    calculate mgu for `(l1, l2)` and `(r1, r2)`; use unifier obtained for `(l1, l2)` when calculating mgu for `(r1, r2)`.
  - `mgu (sub, l1 ** r1, l2 ** r2)`:
    analogous
  - `mgu (sub, V(x), t)`:
    occurs-check and if ok extension of `sub`
  - `mgu (sub, t, V(x))`:
    analogous
  - `mgu (sub, t1, t2)`:
    termination case, return `sub` if `t1=t2` and fail otherwise

7.4.4 Algorithm 'W'

- Type an expression e using assumptions A (types of constants and built-in functions occuring in e, can contain type variables).
- $\mathcal{W}$ returns a type expression τ and a substitution T for which TA defines the type assignements to the type variables in A.
- Remark: A type expression is implicitely universally quantified: $\alpha \to \beta$ is $\forall\alpha\forall\beta\alpha \to \beta$. If type variables are all quantified at toplevel (as in ML), we speak of **shallow types**. Each shallow type is τ: $\forall\alpha_1, \ldots, \alpha_n.\tau$ with no quantification in τ.

$\mathcal{W}(A, e) = (T, \tau)$ **where**

1. $e = x$ (Identifier)
   $T = I$
   If $x : \forall\alpha_1, \ldots, \alpha_n.\sigma \in A$ then
   $\tau = [\beta_1/\alpha_1] \ldots [\beta_n/\alpha_n]\sigma$
   where $\{\beta_i \mid 1 \leq i \leq n\}$ are new type variables.

2. $e = fg$ (Application)
   let
   $(R, \rho) = \mathcal{W}(A, f)$
   $(S, \sigma) = \mathcal{W}(RA, g)$
   $U = \mathcal{V}(S\rho, \sigma \to \beta)$
   where β is new.
   Then
   $T = USR$ and $\tau = U\beta$.

3. **if** p **then** f **else** f' (Condition)
   let
   $(R, \rho) = \mathcal{W}(A, p)$
   $U = \mathcal{V}(\rho, bool)$
   $(S, \sigma) = \mathcal{W}(URA, f)$
   $(S', \sigma') = \mathcal{W}(SURA, f')$
   $U' = \mathcal{V}(S'\sigma, \sigma')$.
   Then
   $T = U'S'SUR$ and $\tau = U'\sigma'$.

4. ... (Abstraction)
5. ... (Fixed point)
6. ... (Let Expression)

see Field and Harrison for details

8 Datatypes

8.1 Lists again

Rememter:

- A List over arbitrary elements is a **datatype** involving **constructors** `nil` and `::`.
- `datatype 'a list = nil | :: of 'a * 'a list`
- `val mlis = 1 :: nil : int list`

8.1.1 Length, Append, Reverse

```
fun nlength [] = 0
  | nlength (x::xs) = 1 + nlength xs;
> val nlength = fn : 'a list -> int
nlength [[1,2,3],[4,5,6]];
> 2 : int
```

More efficient:

```
local
  fun addlen (n, []) = n
    | addlen (n, x::l) = addlen (n+1, l)
in
  fun length(l) = addlen (0,l)
end;
length (explode "Throw physics to the dogs!");
> 25 : int
```

```
infix 5 @;   (* append *)
fun ([] @ ys) = ys
  | ((x::xs) @ ys) = x :: (xs @ ys);
> val @ = fn : 'a list * 'a list -> 'a list
```

- This version of append dates from the early days of Lisp.
- Costs are proportional to the length of the first list and independent to the length of the second.
- Why is it not more efficient to provide a tail recursive implementation?

Remark: Lists and Pointers

- Joining lists using pointers: point the last pointer of first list to the start of second list. Destructive updating is faster than copying!
- ML has explicit pointers, but: copying is saver.
- built-in operators are realized with safely implemented internal pointers.

```
fun nrev [] = []
  | nrev (x::xs) = nrev(xs) @ [x];
```

very inefficient: total number of conses is $\frac{n(n+1)}{2}$, that is $O(n^2)$

```
fun revAppend ([], ys) = ys
  | revAppend (x:xs, ys) = RevAppend(xs, x::ys);
fun rev xs = revAppend(xs,[]);
```

here effort is linear in the length of the list

8.1.2 Lists of Lists and Lists of Pairs

- Pattern-matching and polymorphism cope nicely with combination of data structures:
  `concat = fn : 'a list list -> 'a list`
  `zip = fn : 'a list * 'b list -> ('a * 'b) list`
  `unzip = fn : ('a * 'b) list -> 'a list * 'b list`

```
fun concat [] = []
  | concat (l::ls) = l @ concat ls;
```

reasonably fast, because `l` is usually much shorter than `concat ls`.

```
fun zip(x::xs, y::ys) = (x,y) :: zip(xs,ys)
  | zip _             = [];  (* wild card *)

fun conspair ((x,y), (xs, ys)) = (x::xs, y::ys);
fun unzip [] = ([],[])
  | unzip (pair::pairs) = conspair(pair, unzip pairs);

(* alternative *)
fun unzip [] = ([],[])
  | unzip ((x,y)::pairs) =
          let val (xs, ys) = unzip pairs
          in (x::xs, y::ys) end;
```

- Structure `List` provides functions `take` (return the first i elements of a list), `drop` (return the list without the first i elements), `concat`, ...
- Structure `ListPair` provides `zip`, `unzip`.

8.2 Equality Test in Polymorphic Functions

- Polymorphic functions like `length` or `rev` accept lists having elements of any type because **they do not perform operations** on those elements.
- A function `mem`, which tests whether a value `e` is a member of list `l` involves equality testing with element `e`. Equality testing is polymorphic in a restricted sense.
- **Equality Types**: types whose values admit equality testing.
- Equality test on functions is not computable: f and g are equal just when $f(x) = g(x)$ for every possible argument x.
- Equality test on abstract types (will be introduced later) is not computable.
- Equality is defined for basic types: `int`, `real`, `char`, `string`, `bool`.
- Equality test for structured values: comparison of components: tuples, records, lists, and other datatypes such as trees built over basic types.
- **Equality type variables**: $\alpha^{=}, \beta^{=}, \gamma^{=}, \ldots$. In ML: `''a, ''b, ''c, ...`
- Examples:
  `int`, `bool * string`, `(int list) * ''b` are equality types
  `int -> bool`, `bool * 'b` are no equality types.

```
op= ;
> fn (''a * ''a) -> bool
infix mem;
fun (x mem []) = false
  | (x mem (y::l)) = (x=y) orelse (x mem l);
> val mem fn : ''a * ''a list -> bool
```

8.2.1 Polymorphic Set Operations

- A function's type contains equality type variables if it performs polymorphic equality testing, even indirectly, e. g., via `mem`.
- Examples are functions for using lists as sets.
- Sets ought to be declared as abstract types (see later section) to hide equality test on lists.

Set-Constructor:

```
fun newmem(x, xs) = if x mem xs then xs else x::xs;
```

Conversion to Set

```
fun setof [] = []
  | setof(x::xs) = newmem(x, setof xs);
```

Union and Intersection

```
fun union([], ys) = ys
  | union(x::xs, ys) = newmem(x, union(xs, ys));

fun inter([], ys) = []
  | inter(x::xs, ys) = if x mem ys then x::inter(xs,ys)
                       else inter(xs,ys);
```

Subset and Equality

```
infix subs;
fun ([] subs ys) = true
  | ((x:xs) subs ys) = (x mem ys) andalso (xs subs ys);

infix seq;
fun (xs seq ys) = (xs subs ys) andalso (ys subs xs);
```

Powerset

```
fun powset ([], base) = [base]
  | powset (x:xs, base) =
       powset(xs, base) @ powset(xs, x::base);
```

Carthesian Product

```
fun cartprod ([], ys) = []
  | cartprod (x::xs, ys) =
       let val xsprod = cartprod(xs,ys)
           fun pairx [] = xsprod
             | pairx (y::ytail) = (x,y) :: (pairx ytail)
       in pairx ys end;
```

- `powset` does not perform equality tests. `base` should be empty in the initial call.
  $powset(S, B) = \{T \cup B \mid T \subseteq S\}$.
- `catprod` does not perform equality tests either.
  $S \times T = \{(x, y) \mid x \in S, y \in T\}$.
  Carthesian product can be calculated much more elegantly using higher-order functions (later section).

Remark: In Paulson, chap. 3 there are ML implementation of many standard algorithms on lists, such as greedy search, backtracking, sorting algorithms, matrix operations, graph algorithms, computational algebra.

8.2.2 Association Lists

Lists of *key/value* pairs. The keys must be an equality type.

```
fun assoc ([], a) = []
  | assoc ((x,y)::pairs, a) = if a=x then [y]
                              else assoc(pairs, a);
> val assoc = fn : (''a * 'b) list * ''a -> 'b list
```

- Equality polymorphism has its origin in Lisp, where `mem` and `assoc` are among the most basic primitives.
- Equality polymorphism complicates the language definition and its implementation. Overloading of the equality test operation in ML is, as overloading in general, not elegantly realized. Alternative: type classes in Haskell, but they have other problems ↪ more research is needed!

8.3 Datatype Declarations

- Lists are an example for a datatype, also called “concrete data” in ML.
- An ML `datatype` declaration defines a new type along with its **constructors**.
- In an **expression**: constructors create values of a data type (`hd 1::(2::nil)`)
  In **patterns**: constructors describe how to take values apart (`x::xs`)
- A datatype can represent a class consisting of distinct subclasses (compare to variant records in Pascal).
- **Recursive datatypes** can be defined, such as list and tree.
- Functions on a datatype are declared by pattern-matching.

8.3.1 Enumeration Types

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron;

datatype bool = false | true;
fun not true = false
  | not false = true;

datatype order = LESS | EQUAL | GREATER;
String.compare ("York", "Lancaster");
> GREATER : order

datatype person = King
                | Peer of degree * string * int
                | Knight of string
                | Peasant of string;
```

The type *degree* consists of 5 constructors (`con name : type`). *degree* and `bool` are enumeration types, because they consist of a finite number of constants. Type `person` is not a simple enumeration type.

8.3.2 Polymorphic Datatypes

- A datatype declaration can introduce type operators (type constructors). `List` is a type operator taking one argument `'a`. Therefore, `list` is not a type, while `(int)list` or `((string * real)list)list` are.

```
datatype 'a option = NONE | SOME of 'a;
```

- A datatype *disjoint sum* (type union) can express all non-recursive datatypes (in a clumsy way).

```
datatype ('a, 'b) sum = In1 of 'a | In2 of 'b;
```

- Constructors:
  $In1 : \alpha \to (\alpha, \beta)sum$
  $In2 : \beta \to (\alpha, \beta)sum$

- $(\sigma, \tau)sum$ is the disjoint sum of types σ and τ.
  Its values have the form $In1(x)$ for x of type σ and $In2(y)$ for y of type τ. (`In1` and `In2` are "labels" that distinguish σ from τ.

```
[In2(King), In1("Scottland")] : ((string, person)sum)list
[In1("tyrant"), In2(1040)] : ((string, int)sum)list

(* concatenate all string in In1 in a list *)
fun concat1 [] = " "
  | concat1 ((In1, s)::l) = s ^ concat1 l
  | concat1 ((In2, _)::l) = concat1 l;
> val concat1 = fn : (sing, 'a)sum list -> string
```

Datatype `Person` als disjoint sum: $((unit, string \times string \times int)sum, (string, string)sum)sum$ with constructors

```
King = In1(In1())  (* unit represents the one king *)
Peer(d,t,n) = In1(in2(d,t,n))
Knight(s) = In2(In1(s))
Peasant(s) = In2(In2(s))
```

Storage Requirements

- With current compilers, datatypes require a surprising amout of space.
- Typical: 4 bytes for the tag (identifying the constructor), 4 bytes for each component of the associated tuple; header for garbage collector needs another 4 bytes.
- E. g: 12 bytes for a Knight or Peasant, 20 bytes for a Peer (strings a seperate objects).
- Internal values of an enumeration type require no more space than integers.
- List cells typically occupy 8 to 12 bytes
- Not having types at run-time (run-time type information) saves storage (Lisp).

See Paulson, p. 130 for more details

8.3.3 Pattern-Matching with ‘val’, ‘as’, ’case’

- A pattern is an expression consisting solely of variables, constructors, and wildcards.
- The constructors comprise:
  - numeric, character, and string constants
  - pairing, tupling, and record formation
  - list and datatype constructors.
- In a pattern all names except constructors are variables.
  Variables in a pattern must be distinct.
  These conditions ensure that values can be matched efficiently against the pattern and analysed uniquely to bind variables.
- Constructors absolutely must be distinguished from variables (in Haskell: constructors must begin with a capital letter, variables with a small letter).
  In ML standard constructors `nil`, `true`, `false` are also in small letters. In the standard library constructors are typically in all captial letters.

Patterns in value declarations: `val P = E`

```
val a = (1.5, 6.8);  (* selecting components from a tuple *)
val (xc,yc) = scalevec(4.0, a);
> val xc = 6.0 : real
> val yc = 27.2 : real


val [x,y,z] = uptop(1,3);
> val x = 1 : int
> val y = 2 : int
> val z = 3 : int
```

- If the value of an expression does not match the pattern, the declaration fails (raises an exception).
- The following declarations do not declare variables:

  ```
  val King = King;
  val [1,2,3] = upto(1,3);
  ```

- Constructor names cannot be declared for another purpose.
  In the scope of `Person`, the names `King`, `Peer`, etc. are reserved as constructors. The following declarations are regarded as attempts to pattern macthing and will be rejected:

  ```
  val King = "Henry V";
  val Peer = 925;
  ```

Layered Patterns: `Id as P`

```
fun nextrun(run, []) = ...
  | nextrun(run as r::_, x::xs) =
      if x < r then (rev run, x::xs)
               else nextrun(x::run, xs);
```

- `run` and `r::_` are the same list! Instead of `hd run` we can use `r`

Pattern Matching with 'Case' Expressions
Form of the `case` expression:

```
case E of P1 => E1 | ... | Pn => En
```

- The value of E is successively matched against patterns P_1 to P_n. If P_i is the first pattern to match E then the result is the value of E_i.
- `Case` is equivalent to an expression that declares a function by cases and applies it to E.
- No symbol terminates the case-expression: use parentheses!

```
case p-q of
  0 => "zero"
  1 => "one"
  2 => "two"
  n => if n < 10 then "lots" else "lots and lots"
```

9 Datatype Exception and Recursive Datatypes

9.1 Exceptions

- The ML exception mechanism is similar to Ada:
  - can declare and raise exceptions
  - no hierarchical organization (such as Java)
  - no explicit throws declaration (such as Java)
  - handling by pattern matching (cases)
  - propagation to invoking functions if not handled
- In difference to Ada: exceptions can be parametrized
- Handling must return the correct result type of the function!

Example:

```
exception BadMath of string * int * int;  (* declaration *)

fun avg(sum, count) =
  if count = 0 then raise BadMath("div", sum, count)
               else sum div count;

fun saveAverage(sum, count) =
  avg(sum, count)
  handle BadMath(s, a, b) => (
    print("Math error: attempted ");
    print(s);
    print(" of");
    print(Int.toString(a));
    print(" and ");
    print(Int.toString(b));
    print("Result of zero used.");
    print("\n");
    0  (* return value! *)
  );
```

9.1.1 Declaring Exceptions

`exception <name> [of < parameter types >]`

- An exception in ML is a constructor of the built-in type `exn`. This is a datatype of unique property: its set of constructors can be extended by new, user-declared exceptions.
- Exceptions can be declared locally, using let. This can result in different exceptions having the same name.
- The type of a top-level exception must be monomorphic!
- Values of type `exn` can be used as any other values (sorted in lists, returned by functions, ...). Additionally, they have a special role in the operations `raise` and `handle`.

```
exception Failure;
exception Failedbecause of string;
exception Badvalue of int;
```

- Remark: **Dynamic Typing**: Type `exn` can be extended with new constructors and therefore potentially includes the values of any type.
- Example: provide a uniform interface for expressing arbitrary data as strings. Conversion functions: `exn -> string`. Extension to a new type, such as `Complex.t`:

  ```
  exception ComplexToString of Complex.t;
  fun convert_complex (ComplexToString z) = ...
  ```

  This function only works if it is applied to the constructor `ComplexToString`. A collection of similar functions might be stored in a dictionary, identified by uniform keys (such as strings). $\hookrightarrow$ we obtain a basic form of object-oriented programming.

9.1.2 Raising Exceptions

`raise <exception>`

- Raising an exception creates an **exception packet** containing a value of type `exn`.
- If `Ex : exn` is an exception and evaluates to e then `raise Ex` evaluates to a packet containing `e`.
- Packets are *not* ML values! The only operations recognizing them are `raise` and `handle`. Type `exn` mediates between packets and ML values.
- During evaluation, packets propagate under the **call-by-value rule**. If expression E returns an exception packet then that packet is the result of the application $f(E)$. Thus $f(raise(Ex))$ is equivalent to $raise\ Ex$.
- `raise` itself propagates exceptions, thus: `raise (Badvalue (raise Failure))` raises exception `Failure`!
- Expressions in ML are evaluated left to right. If E_1 returns a packet, then that is the result of pair (E_1, E_2) and E_2 is not evaluated at all. The evaluation order matters when E_1 and E_2 raise different exceptions. (cf. evaluation order in the conditional expression)
  In `let val P = E1 in E2 end`, if E_1 evaluates to an exception packet then so does the entire let-expression.
- Exception packets are not propagated by testing. The ML system efficiently jumps to the correct exception handler if there is one, otherwise terminating execution.

Standard Exceptions

- `Match`: failure of pattern-matching.
- `Bind`: `val P = E` if `E` does not match `P`
- `Overflow`
- `Div`
- `Domain`: in structure `Math`, e.g., square root of a negative number
- `Chr`: invalid character code k in `chr(k)`
- `Subscript`: index out of range (array, string, list operations)
- `Size`: creation of array, string, list with negative or excessive size
  `Fail`: miscellaneous erros
- ...

```
exception Empty; (* in List *)
fun hd (x::_) = x
  | hd [] = raise Empty;

exception Subscript;
fun nth (x::_, 0) = x;
  | nth (x::xs, n) = if n > 0 then nth(xs,n-1)
                     else raise Subscript
  | nth _          = raise Subscript;
```

9.1.3 Handling Exceptions

`E handle` $\mathtt{P}_1 \rightarrow \mathtt{E}_1$ | `...` | $\mathtt{P}_n \rightarrow \mathtt{E}_n$

- An exception handler tests whether the result of an expression E is an exception packet.
  If so, the packets content (a value of type *exn*) may be examined by cases.

- If E returns a normal value, then the handler passes this value on.
  If E returns a packet, then its contents are matched against $P_1 \ldots P_n$.
  If P_i is the first pattern to match, then the value of E_i is returned.
  If no pattern matches, the handler propagates the packet (different to the case-expression, where this would result in a match-exception).

- An exception handler must be written with care:
  If an exception name is misspelled it will be taken as a variable and matches all exceptions.
  Be carefull to give the handler the right scope: in `if E then E1 else E2 handle ...` the handler will only detect an exception raised by $E2$ (use parentheses)

9.1.4 Exceptions versus Pattern-Matching

```
fun length (nil) = 0
  | length (x::xs) = 1 + length(xs);

fun len l = 1 + len(tl l) handle _ => 0;
```

Evaluatuation (writing □ for the exception packet):

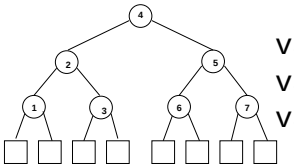
len[1] ⇒ 1 + len(tl[1]) handle _ => 0
⇒ 1 + len[] handle _ => 0
⇒ 1 + (1 + len(tl[]) handle _ => 0) handle _ => 0
⇒ 1 + (1 + len □ handle _ => 0) handle _ => 0
⇒ 1 + (1 + □ handle _ => 0) handle _ => 0
⇒ 1 + (□ handle _ => 0) handle _ => 0
⇒ 1 + 0 handle _ => 0
1

- Evaluation of the *length* function with pattern matching is less complicated.
  Test for different cases in advance, if possible!
- Lazy evaluation is typically not combined with exception handling (call-by-need evaluation makes error-propagation problematic).

9.2 Recursive Datatypes

9.2.1 Binary Trees

```
datatype 'a tree = Lf | Br of 'a * 'a tree * 'a tree;
```




```
val tree2 = Br(2, Br(1, Lf, Lf), Br(3, Lf, Lf));
val tree5 = Br(5, Br(6, Lf, Lf), Br(7, Lf, Lf));
val tree4 = Br(4, tree2, tree5);
```

Polymorphic Functions `size` and `depth`:

```
fun size Lf = 0
  | size (Br(v, t1, t2)) = 1 + size t1 + size t2;

fun depth Lf = 0
  | depth (Br(v, t1, t2)) = 1 + Int.max(depth t1, depth t2);
```

- Remember: $size(t) \leq 2^{depth(t)} - 1$; complete binary tree: $size(t) = 2^{depth(t)} - 1$; concepts of balanced trees

Creating a complete binary tree of integers:

```
fun comptree (k, n) =  (* n is the depth, k is the least value *)
  if n = 0 then Lf
  else Br(k, comptree(2*k, n-1), comptree(2*k+1, n-1)),
```

Mirror a tree:

```
fun reflect Lf = Lf
  | reflect (Br(v, t1, t2)) = Br(v, reflect t2, reflect t1);
```

Enumerating the Contents of a Tree

```
fun preorder Lf = []
  | preorder (Br(v, t1, t2)) = [v] @ preorder t1 @ preorder t2;

fun inorder Lf = []
  | inorder (Br(v, t1, t2)) = inorder t1 @ [v] @ inorder t2;

fun postorder Lf = []
  | postorder (Br(v, t1, t2)) = postorder t1 @ postorder t2 @ [v];
```

- Quadratic effort on badly unbalanced trees (because of @).
- Use additional argument.

```
fun preord (Lf, vs) = vs
  | preord (Br(v, t1, t2), vs) = v :: preord(t1, preord(t2, vs));

fun inord (Lf, vs) = vs
  | inord (Br(v, t1, t2), vs) = inord(t1, v :: inord(t2, vs));

fun postord (Lf, vs) = vs
  | postord (Br(v, t1, t2), vs) = postord(t1, postord(t2, v::vs));
```

Making a balanced tree from a preorder list of labels (inverse function to enumeration):

```
fun balpre [] = Lf
  | balpre(x::xs) =
      let val k = length xs div 2
      in Br(x, balpre(List.take(xs, k)), balpre(List.drop(xs, k)))
      end;
```

A Structure for Binary Trees

```
datatype 'a tree = Lf | Br of 'a * 'a tree * 'a tree;

structure Tree =
struct
fun size ...
fun depth ...
fun preord ...
...
fun balpre ...
...
end;
```

9.2.2 Tree-based Datastructures

```
(*** Dictionaries as Binary search trees ***)
signature DICTIONARY =
  sig
  type key                                (*type of keys*)
  type 'a t                               (*type of tables*)
  exception E of key                      (*errors in lookup, insert*)
  val empty: 'a t                         (*the empty dictionary*)
  val lookup: 'a t * key -> 'a
  val insert: 'a t * key * 'a -> 'a t
  val update: 'a t * key * 'a -> 'a t
  end;

(*Structure Order can vary; Tree avoids referring to a free structure. *)
structure Dict : DICTIONARY =
  struct
  type key = string;
  type 'a t = (key * 'a) tree;
  exception E of key;
  val empty = Lf;

  fun lookup (Lf, b) = raise E b
    | lookup (Br ((a,x),t1,t2), b) =
        (case String.compare(a,b) of
             GREATER => lookup(t1, b)
           | EQUAL   => x
           | LESS    => lookup(t2, b));
  fun insert (Lf, b, y) = Br((b,y), Lf, Lf)
    | insert (Br((a,x),t1,t2), b, y) =
        (case String.compare(a,b) of
             GREATER => Br ((a,x),  insert(t1,b,y),  t2)
           | EQUAL   => raise E b
           | LESS    => Br ((a,x),  t1,  insert(t2,b,y)));
  fun update (Lf, b, y) = Br((b,y), Lf, Lf)
    | update (Br((a,x),t1,t2), b, y) =
        (case String.compare(a,b) of
             GREATER => Br ((a,x),  update(t1,b,y),  t2)
           | EQUAL   => Br ((a,y),  t1,  t2)
           | LESS    => Br ((a,x),  t1,  update(t2,b,y)));

end;
```

```
(*** Functional and flexible arrays ***)

(*Braun trees*)
structure Braun =
  struct
  fun sub (Lf, _) = raise Subscript
    | sub (Br(v,t1,t2), k) =
        if k = 1 then v
        else if k mod 2 = 0
             then sub (t1, k div 2)
             else sub (t2, k div 2);

  fun update (Lf, k, w) =
        if k = 1 then Br (w, Lf, Lf)
        else raise Subscript
    | update (Br(v,t1,t2), k, w) =
        if k = 1 then Br (w, t1, t2)
        else if k mod 2 = 0
             then Br (v,  update(t1, k div 2, w),  t2)
             else Br (v,  t1,  update(t2, k div 2, w));

  fun delete (Lf, n) = raise Subscript
    | delete (Br(v,t1,t2), n) =
        if n = 1 then Lf
        else if n mod 2 = 0
             then Br (v,  delete(t1, n div 2),  t2)
             else Br (v,  t1,  delete(t2, n div 2));

  fun loext (Lf, w) = Br(w, Lf, Lf)
    | loext (Br(v,t1,t2), w) = Br(w, loext(t2,v), t1);

  fun lorem Lf = raise Size
    | lorem (Br(_,Lf,Lf)) = Lf  (*No evens, therefore no odds either*)
    | lorem (Br(_, t1 as Br(v,_,_), t2)) = Br(v, t2, lorem t1);
  end;
```

```
(** Flexible arrays as abstract type **)
signature FLEXARRAY =
  sig
  type 'a array
  val empty:  'a array
  val length: 'a array -> int
  val sub:    'a array * int -> 'a
  val update: 'a array * int * 'a -> 'a array
  val loext:  'a array * 'a -> 'a array
  val lorem:  'a array -> 'a array
  val hiext:  'a array * 'a -> 'a array
  val hirem:  'a array -> 'a array
  end;

(*These arrays are based from ZERO for compatibility with arrays in
  the Basis Library.  They check bounds and raise standard exceptions.*)
structure Flex : FLEXARRAY =
  struct
  datatype 'a array = Array of 'a tree * int;
  val empty = Array(Lf,0);

  fun length (Array(_,n)) = n;
  fun sub (Array(t,n), k) =
      if 0<=k andalso k<n then Braun.sub(t,k+1)
      else raise Subscript;
  fun update (Array(t,n), k, w) =
      if 0<=k andalso k<n then Array(Braun.update(t,k+1,w), n)
      else raise Subscript;
  fun loext (Array(t,n), w) = Array(Braun.loext(t,w), n+1);
  fun lorem (Array(t,n)) =
      if n>0 then Array(Braun.lorem t, n-1)
      else raise Size;
  fun hiext (Array(t,n), w) = Array(Braun.update(t,n+1,w), n+1);
  fun hirem (Array(t,n)) =
      if n>0 then Array(Braun.delete(t,n) , n-1)
      else raise Size;
  end;
```

```
(*** Priority queues ***)
signature PRIORITY_QUEUE =
  sig
  type item
  type t
  val empty    : t
  val null     : t -> bool
  val insert   : item * t -> t
  val min      : t -> item
  val delmin   : t -> t
  val fromList : item list -> t
  val toList   : t -> item list
  val sort     : item list -> item list
  end;

structure Heap : PRIORITY_QUEUE =
  struct
  type item = real;
  type t = item tree;
  val empty = Lf;

  fun null Lf = true
    | null (Br _) = false;
  fun min (Br(v,_,_)) = v;
  fun insert(w: real, Lf) = Br(w, Lf, Lf)
    | insert(w, Br(v, t1, t2)) =
        if w <= v then Br(w, insert(v, t2), t1)
                  else Br(v, insert(w, t2), t1);
  fun leftrem (Br(v,Lf,Lf)) = (v, Lf)
    | leftrem (Br(v,t1,t2)) =
        let val (w, t) = leftrem t1
        in  (w, Br(v,t2,t))  end;
  fun siftdown (w:real, Lf, Lf) = Br(w,Lf,Lf)
    | siftdown (w, t as Br(v,Lf,Lf), Lf) =
        if w <= v then Br(w, t, Lf)
                  else Br(v, Br(w,Lf,Lf), Lf)
    | siftdown (w, t1 as Br(v1,p1,q1), t2 as Br(v2,p2,q2)) =
        if w <= v1 andalso w <= v2 then Br(w,t1,t2)
        else if v1 <= v2 then Br(v1, siftdown(w,p1,q1), t2)
           (* v2 < v1 *) else Br(v2, t1, siftdown(w,p2,q2));
```

```
fun delmin Lf = raise Size
  | delmin (Br(v,Lf,_)) = Lf
  | delmin (Br(v,t1,t2)) =
      let val (w,t) = leftrem t1
      in  siftdown (w,t2,t)  end;
fun heapify (0, vs) = (Lf, vs)
  | heapify (n, v::vs) =
      let val (t1, vs1) = heapify (n div 2, vs)
          val (t2, vs2) = heapify ((n-1) div 2, vs1)
      in  (siftdown (v,t1,t2), vs2)  end;

fun fromList vs = #1 (heapify (length vs, vs));

fun toList (t as Br(v,_,_)) = v :: toList(delmin t)
  | toList Lf = [];

fun sort vs = toList (fromList vs);
end;
```

9.3 Elementary Theorem Proving

- Terms and logical formulas can be represented as trees.
- Now: Application of datatypes, example **Tautology Checker** for propositional logic.

Propositional Logic (Aussagenlogik)

- An **atom** a is a proposition.
- If p and q are propositions, then the following expressions are propositions:
  $\neg p$, $p \wedge q$, $p \vee q$.

Remarks:

- Constants T and F are special atoms: T evaluates to *true* and F evaluates to *false*.
- Atoms a are interpreted as truth values (can be *true* or *false*).
- All propositional expressions can be represented with negation, logical 'and' and logical 'or'. For example: $p \rightarrow q \equiv \neg p \vee q$ (de Morgan).
- A proposition is a tautology if it evaluates to *true* for all possible assignments of truth values to atoms.
  Example: $(p \wedge (p \rightarrow q) \rightarrow q) \equiv p \vee \neg p \vee \neg q \vee q \equiv true$.
- Tautology checking can be used to prove theorems: If for a set of assumptions p_i and a conclusion q, the expression $\wedge_{i=1}^{n} p_i \rightarrow q$ evaluates to true, then the conclusion follows from the assumptions and is a propositional theorem or a tautology.

A Simple Tautology Checker

- Define a datatype `prop` as introduced above:

```
datatype prop = Atom of string
              | Neg of prop
              | Conj of prop * prop
              | Disj of prop * prop;
```

- Implication can be rewritten: `fun implies(p,q) = Disj(Neg p, q);`
- The most simple tautology check is, to have a conjunctive normal form, that is, $p_1 \wedge \ldots \wedge p_n$, where each p_i is a disjunction of literals (positive and negated atoms). Then it can be checked whether:
  - each of the p_i is a tautology,
  - for every $p_i = q_1 \vee \ldots \vee q_m$: check whether some q_j appears positive and negative. If yes, p_i is true.

  The check of each disjunction (clause) can be done by putting all positive literals in a list and all negative literals in another list and check whether the intersection is not empty.

Example:
Ass1: $p \rightarrow q$
Ass2: $\neg(r \wedge q)$
Conc: $p \rightarrow \neg r$

Does hold: $[(p \rightarrow q) \wedge \neg(r \wedge q)] \rightarrow p \rightarrow \neg r$?

1. **Eliminate implication:** (use rewrite rule given above)
   $\neg((\neg p \vee q) \wedge \neg(r \wedge q)) \vee (\neg p \vee \neg r)$
2. **Calculate Negative Normal Form** (negation is only applied to atoms):
   The following equivalences can be used for rewriting: $\neg(p \wedge q) \equiv \neg p \vee \neg q$ and $\neg(p \vee q) \equiv \neg p \wedge \neg q$
   $(p \wedge \neg q) \vee (r \wedge q) \vee (\neg p \vee \neg r)$
3. **Calculate Conjunctive Normal Form (CNF):**
   The following Distributive Laws can be used:
   $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$ and $(q \wedge r) \vee p \equiv (q \vee p) \wedge (r \vee p)$
4. Check each member of the CNF whether it is a tautology.
   For each clause: **Build lists of the positive and negative literals** and check for non-empty intersection.

Remarks:

- More efficient representations for practical tautology checkers (invented in hardware design and verification): OBDDs
- Davis-Putnam procedure (working on CNFs) is the heart of SAT-solvers.

```
(*** Propositional logic -- tautology checker ***)

(*REQUIRES: set intersection*)
infix mem;
fun x mem []  =  false
  | x mem (y::l)  =  (x=y) orelse (x mem l);
fun inter([],ys) = []
  | inter(x::xs, ys) =
        if x mem ys then x::inter(xs, ys)
                    else    inter(xs, ys);

datatype prop =
    Atom of string
  | Neg  of prop
  | Conj of prop * prop
  | Disj of prop * prop;

fun show (Atom a) = a
  | show (Neg p) = "(~ " ^ show p ^ ")"
  | show (Conj(p,q)) = "(" ^ show p ^ " & " ^ show q ^ ")"
  | show (Disj(p,q)) = "(" ^ show p ^ " | " ^ show q ^ ")";

(*naive version*)
fun nnf (Atom a) = Atom a
  | nnf (Neg (Atom a)) = Neg (Atom a)
  | nnf (Neg (Neg p)) = nnf p
  | nnf (Neg (Conj(p,q))) = nnf (Disj(Neg p, Neg q))
  | nnf (Neg (Disj(p,q))) = nnf (Conj(Neg p, Neg q))
  | nnf (Conj(p,q)) = Conj(nnf p, nnf q)
  | nnf (Disj(p,q)) = Disj(nnf p, nnf q);

fun nnfpos (Atom a) = Atom a
  | nnfpos (Neg p) = nnfneg p
  | nnfpos (Conj(p,q)) = Conj(nnfpos p, nnfpos q)
  | nnfpos (Disj(p,q)) = Disj(nnfpos p, nnfpos q)
and nnfneg (Atom a) = Neg (Atom a)
  | nnfneg (Neg p) = nnfpos p
  | nnfneg (Conj(p,q)) = Disj(nnfneg p, nnfneg q)
  | nnfneg (Disj(p,q)) = Conj(nnfneg p, nnfneg q);
```

```
fun distrib (p, Conj(q,r)) = Conj(distrib(p,q), distrib(p,r))
  | distrib (Conj(q,r), p) = Conj(distrib(q,p), distrib(r,p))
  | distrib (p, q) = Disj(p,q)   (*no conjunctions*) ;

fun cnf (Conj(p,q)) = Conj (cnf p, cnf q)
  | cnf (Disj(p,q)) = distrib (cnf p, cnf q)
  | cnf p = p    (*a literal*) ;

exception NonCNF;

fun positives (Atom a)      = [a]
  | positives (Neg(Atom _)) = []
  | positives (Disj(p,q))   = positives p @ positives q
  | positives  _            = raise NonCNF;

fun negatives (Atom _)      = []
  | negatives (Neg(Atom a)) = [a]
  | negatives (Disj(p,q))   = negatives p @ negatives q
  | negatives  _            = raise NonCNF;

fun taut (Conj(p,q)) = taut p andalso taut q
  | taut p = not (null (inter (positives p, negatives p)));
```

10 Functionals

- Most powerful techniques of functional programming: treat functions as data.
- Like other values, functions can be arguments and results of other functions and may belong to pairs, lists, trees.
- In procedural languages the use of functions as values is restricted: functions can be arguments of other functions (e. g., giving the comparison operator as argument to a sorting-function).
- A function is **higher-order** or a **functional** if it operates on other functions. Example: general purpose functional `map` applies a function to every element of a list, creating a new list.
- With a suitable set of functionals, all functions can be expressed without variables!
- Applications (later sections): Parser construction, theorem proving strategies.
- Important concepts origin in Church's λ-calculus which we introduce in a later section.

10.1 Anonymous Functions

- If x is a variable of type σ and E is an expression of type τ, then `fn x => E;` denotes a function of type $\sigma \to \tau$ with argument x and body E.
- The expression `fn P1 => E1 | ... | Pn => En;` denotes the function defined by the patterns $P_1, \ldots, P_n$. It is equivalent to the let-expression:
  `let fun f(P1) = E1 | ... | f(Pn) = En in f end`
  provided that f does not appear in the expressions E_i.
- The `fn`-syntax cannot express recursion!
- An anonymous function can be applied to an argument:
  ```
  (fn n => n*2)(9);
  ```
- An anonymous function can be given a name by a `val`-declaration:
  ```
  val double = fn n => n*2;
  ```
- Many ML constructs are defined in terms of the `fn`-notation:
  ```
  if E then E1 else E2 == (fn true => E1 | false => E2) (E)
  ```

10.2 Curried Functions

- A function can only have one argument. Two possibilities to represent functions over multiple arguments: (1) use a tuple, (2) **Currying**.
- Curry and Schönfinkel: Use a function that returns another function as result.
- Function over pairs with type $(\sigma_1 \times \sigma_2) \to \tau$.
  Curried function with type $\sigma_1 \to (\sigma_2 \to \tau)$.

```
fun prefix (pre, post) = preˆpost;

fun prefix pre =
  let fun cat post = preˆpost
  in cat end;
> val prefix = fn : string -> (string -> string)

fn pre => (fn post => preˆpost);
```

10.2.1 Lexical Closures and Partial Application

```
- val prefix = fn pre => (fn post => preˆpost);
val prefix = fn : string -> string -> string
- prefix "Sir ";
val it = fn : string -> string
- it "James";
val it = "Sir James" : string
- (prefix "Sir ") "James";
val it = "Sir James" : string
```

- `prefix` behaves like a function with two arguments.
- It allows partial application: Applied to its first argument, it returns a function with the second argument as argument.
- The instantiation of the first argument defines the *lexical context*, also called a *lexical closure*.

```
val knightify = prefix "Sir ";
knightify "William";
knightify "Richard";
val dukify = prefix "The Duke of ";
dukify "Clarence";
```

Excursus: Lexical Closures in Lisp and Adapter-Classes in Java

- *Lexical Closure*: A function remembering its context, binding free variables with respect to this context.
- Example for Common Lisp:
  For the inner, nameless, function (λ-expression), paramter n is free and parameter m is bound.
  Result of `adder 3` is a function which adds 3 to its argument m.

```
[1]> (defun adder (n) (function (lambda (m) (+ m n))))
ADDER
[2]> (setq add3 (adder 3))
#<CLOSURE :LAMBDA (M) (+ M N)>
[3]> (funcall add3 4)
7
```

- In Java, adapter classes (nearly) imitate the concept of lexical closures.
  In `makeAdder` the `Adder`-Interface is adapted for a special demand. The anonymous inner class realizes the lexical closure.
- But: parameter n must be declared final. That is, it is not possible, that different functions share the same context (except if the context would be given in an object of the surrounding class).

```
interface Adder{ int add(int m); }

public Adder makeAdder(final int n) {
  return new Adder() {
    public int add (int m) {return m + n;}
};}
```

10.2.2 Syntax for Curried Functions

- A curried function can be defined using `fun` where arguments are given without parentheses: `fun prefix pre post = preˆpost;` is interpreted as `fn : string -> (string -> string)`.
- A function call has the form EE_1 where E is an expression that denotes a function.
- $EE_1E_2 \ldots E_n$ abbreviates $(\ldots((EE_1)E_2)\ldots)E_n$.
  Expressions are evaluated left to right.
  Symbol $\rightarrow$ associates to the right: $string \rightarrow (string \rightarrow string)$ can be written without parentheses.
- That is: `prefix "Sir " "James";` can be written without parentheses.
- Remark: Compare to two-dimensional and nested arrays `A[i, j]` (“tuple”) vs. `A[i][j]` (“currying”).

10.2.3 Recursion

- Curried functions may be recursive.

```
fun replist n x = if n=0 then [] else x :: replist (n-1) x;
> val replist = fn : int -> 'a -> 'a list
replist 3;
> val fn x => if 3 = 0 then [] else x :: replist(3-1) x;
it true;
> [true true true] : bool list;
```

Remark: This doesn't work with SML!

```
- fun replist n x = if n=0 then [] else x :: replist (n-1) x;
val replist = fn : int -> 'a -> 'a list
- replist 3;
stdIn:28.1-28.10 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val it = fn : ?.X1 -> ?.X1 list
- it true;
stdIn:29.1-29.8 Error: operator and operand don't agree [tycon mismatch]
  operator domain: ?.X1
  operand:         bool
  in expression:
    it true
```

But with Moscow ML!

```
- fun replist n x = if n=0 then [] else x::replist (n-1) x;
> val 'a replist = fn : int -> 'a -> 'a list
- replist 3;
! Warning: Value polymorphism:
! Free type variable(s) at top level in value identifier it
> val it = fn : 'a -> 'a list
- it true;
! Warning: the free type variable 'a has been instantiated to bool
> val it = [true, true, true] : bool list
```

10.2.4 Functions in Data Structures

- Pairs and lists may contain functions as components.
- Functions stored in a data structure can be extracted and applied.
- Functions stored in a data structure (or data type, such as tree) must have the same type. Type `exn` can be regarded as including all types (but redefining functions with different types based on exceptions is a hack!).

```
val titlefns = [dukify, lordify, knightify];
> val titlefns = [fn, fn, fn] : (string -> string) list
hd titlefns "Gloucester";
```

- Curried function call: `hd titlefuns` returns function `dukify`.
- The polymorphic function `hd` has here the type `(string -> string) list -> (string -> string)`.

10.3 Functions as Arguments and Results

```
fun insort lessequal =
    let fun ins (x, []) = [x]
          | ins (x, y::ys) =
              if lessequal(x,y) then x::y::ys
              else y :: ins (x,ys)
        fun sort [] = []
          | sort (x::xs) = ins (x, sort xs)
    in  sort  end;
> val insort = fn : ('a * 'a -> bool) -> 'a list -> 'a list
insort (op<=) [5, 3, 7, 5, 9, 8];
```

- For an argument $\tau \times \tau \rightarrow bool$ `insort` returns the function `sort` with type $\tau list \rightarrow \tau list$.

```
fun summation f m =
    let fun sum (i,z) : real =
            if  i=m  then  z  else  sum (i+1, z + (f i))
    in  sum(0, 0.0)  end;
```

calculates $\sum_{i=0}^{m-1} f(i)$.

11 General-purpose Functionals

11.1 Sections

- Section: an operator which leaves an operand unspecified.
- Examples: `"Sir "^` (function `knightify`), `/2.0` (function 'divide real by 2')
- Section can be added to ML (in a rather crude way) as higher-order functions.
- `secl` ist a function getting a value $x : \alpha$, a function $\alpha \times \beta \to \gamma$ and a second (unspecified) value $y : \beta$ and returns a value $f(x, y) : \gamma$.
- `secl` is a higher-order function because it takes a function as argument.

```
- fun secl x f y = f(x,y);
val secl = fn : 'a -> ('a * 'b -> 'c) -> 'b -> 'c
- val recip = (secl 1.0 op/);
val recip = fn : real -> real
- recip 5.0;
val it = 0.2 : real
- fun secr f y x = f(y,x);
val secr = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

11.2 Combinators

- The theory of λ-calculus is in part concerned with expressions known as combinators.
- Many combinators can be coded in ML as higher-order functions.
- Composition `0` is a combinator. (Remember FP and introduction to functions.)
- Identity `I` Constant Function `K`, and General Composition `S` are combinators.
- Every function in λ-calculus can be expressed using just S and K with no variables.
  This can be exploited for lazy evaluation: since no variables are involved, no mechanism is needed for binding their values.

```
infix o;
fun (f o g) x = f ( g x);
> val o = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

Now functions can be expressed without mentioning their arguments:

```
- ~ o Math.sqrt;
val it = fn : real -> real
- it 2.0;
val it = ~1.41421356237 : real
- (secl 2.0 op/) o (secr op- 1.0);
val it = fn : real -> real

fun I x = x;
> val I = fn : 'a -> 'a
fun K x y = x;
> val K = fn : 'a -> 'b -> 'a
fun S x y z = x z (y z);
> val S = fn : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

Examples:

```
- fun sum x y = x + y;
val sum = fn : int -> int -> int
- S sum I 1; (* 1 + 1 *)
val it = 2 : int
- S sum ~ 1;  (* 1 + -1 *)
val it = 0 : int
- S K K 17;   (* I = S K K *)
val 17 : int
```

11.3 List Functionals 'Map' and 'Filter'

$\boxed{map\ f[x_1, \ldots, x_n] = [fx_1, \ldots, fx_n]}$

$\boxed{filter\ p[x_1, \ldots, x_n] = [x_i \mid p(x_i) = true]}$

```
fun map f [] = []
  | map f (x::xs) = (f x) :: map f xs;
val map = fn : ('a -> 'b) -> 'a list -> 'b list
- fun filter pred [] = []
    | filter pred (x::xs) = if pred x then x :: filter pred xs
                            else filter pred xs;
val filter = fn : ('a -> bool) -> 'a list -> 'a list
- map size ["York", "Clarence", "Gloucester"];
val it = [4,8,10] : int list
- filter (fn a => size a = 4) ["hie","thee","to","hell","thou","cacodemon"
val it = ["thee","hell","thou"] : string list
```

- Map and filter are curried:
  `map` takes a function of type $\sigma \to \tau$ to one of type $\sigma list \to \tau list$.
  `filter` takes a function of type $\tau \to bool$ to one of type $\tau list \to \tau list$.
  Pattern-Matching works exactly as if the arguments were tuples.
- `map` and `filter` can work for lists of lists: $map(mapf)[l_1, \ldots, l_n]$ applies $mapf$ to each list l_i.

```
- fun double x = x*2;
val double = fn : int -> int
- map(map double) [[1],[2,3], [4,5,6]];
val it = [[2],[4,6],[8,10,12]] : int list list
- map (filter (secr op< "m"))
= [["my", "hair", "doth", "stand", "on", "end"],
= ["to", "hear", "her", "curses"]];
val it = [["my","stand","on"],["to"]] : string list list
```

Many functions can be now coded trivially:

```
fun transp ([] :: _) = []
  | transp rows = map hd rows :: transp (map tl rows);
fun inter (xs, ys) = filter (secr (op mem) ys) xs;
```

11.4 List Functionals ‘foldl’ and ‘foldr’

- Also known as *reduce*.

$$\boxed{foldl\ f\ e\ [x_1, \ldots, x_n] = f(x_n, f(x_{n-1}, \ldots, f(x_2, f(x_1, e))\ldots))}$$

$$\boxed{foldr\ f\ e\ [x_1, \ldots, x_n] = f(x_1, f(x_2, \ldots, f(x_{n-1}, f(x_n, e))\ldots))}$$

```
fun foldl f e [] = e   (* e is the identity element *)
  | foldl f e (x::xs) = foldl f (f(x, e)) xs;
fun foldr f e [] = e
  | foldr f e (x::xs) = f(x, foldr f e xs);
```

Examples:

```
val sum = foldl op+ 0;  (* summation of ints with e=0 *)
sum [1,2,3,4];
> 10 : int

foldl (fn (_, n) => n+1) 0 (explode "Margaret");
> 8 : int

foldr op@ [] [[1], [2,3], [4,5,6]];
> [1,2,3,4,5,6] : int list
```

Expressing `map` with `foldr`:

```
fun map f = foldr (fn(x,l) => f x :: l) [];
> val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

Cartesian Product of Lists:

```
fun cartprod (xs, ys) =
    foldr (fn (x, pairs) =>
                foldr (fn (y,l) => (x,y)::l) pairs ys)
            [] xs;
```

Remark: Cartesian Products can be calculated more clearly using `map` and `List.concat`:

```
- fun pair x y = (x,y);
- map (fn a => map (pair a) ["Hastings", "Stanley"]) ["Lord", "Lady"];
val it =
  [[("Lord","Hastings"),("Lord","Stanley")],
   [("Lady","Hastings"),("Lady","Stanley")]]
  : (string * string) list list
- List.concat it;
val it =
  [("Lord","Hastings"),("Lord","Stanley"),("Lady","Hastings"),
   ("Lady","Stanley")] : (string * string) list
```

11.5 The List Functionals ‘Exists’ and ‘All’

```
fun exists pred [] = false
  | exists pred (x::xs) = (pred x) orelse exists pred xs;
fun all pred [] = true
  | all pred (x::xs) = (pred x) andalso all pred xs;
```

Elegant member-test:

```
infix mem;
fun x mem xs = exists (secr op= x) xs;
```

Disjoint Test:

```
fun disjoint (xs, ys) = all (fn x => all (fn y => x <> y) ys) xs;
```

11.6 Functionals in the Standard Library

- o and list functionals `map`, `foldl`, `foldr` are available top-level.
- Components of structure `List`: `map`, `foldl`, `foldr`, `filter`, `exists`, `all`.
- Components of structure `ListPair`: variants of `map`, `exists`, `all`.

11.7 Further Useful Functionals

```
fun takewhile pred [] = []
  | takewhile pred (x::xs) = if pred x then x :: takewhile pred xs
                             else [];

fun dropwhile pred [] = []
  | dropwhile pred (x::xs) = if pred x then dropwhile pred xs
                             else x::xs;

(* map(takewhile pred) returns a list of initial segments *)
(* takewhile (all pred) works on lists of lists *)
```

Powers of a Function

```
fun repeat f n x =
    if n>0  then  repeat f (n-1) (f x)
            else  x;
repeat (fn t =>Br("No", t, t)) 3 Lf;
(* complete binary tree with constant label *)
```

Handling Trees

```
fun treefold f e Lf = e
  | treefold f e (Br(u,t1,t2)) = f(u, treefold f e t1, treefold f e t2);
val size = treefold (fn(_, c1,c2) => 1+c1+c2) 0;
val depth = treefold (fn(_,d1,d2) => 1 + Int.max(d1,d2) 0;
val preordlist = treefold(fn(u,l1,l2) => [u] @ l1 @ l2) [];
```

Operations on Terms

```
datatype term = Var of string | Fun of string * term list;

(* (x+u) - (y*x) *)
val tm = Fun("-", [Fun("+", [Var "x", Var "u"]),
                   Fun("*", [Var "y", Var "x"]),

(* Substitution *)
fun subst f (Var a) = f a
  | subst f (Fun (a, args)) = Fun(a, map (subst f) args);
(* List of Variables *)
fun vars (Var a) = [a]
  | vars (Fun(_,args)) = List.concat (map vars args);
```

List functionals are patterns (schemes) for many standard functions:

- Examples for *map*: all functions which change every element of an input list (such as square each number in a list).
- Examples for *filter*: return all odd-numbers of a list, delete an element in a list, ...
- Examples for *reduce* (fold): calculate the sum of a list of numbers, ...

12 Infinite (Lazy) Lists – Sequences

- Lazy lists are one of the most celebrated features of functional programming.
- The elements of a lazy list are not evaluated until their values are required by the rest of the program, thus a lazy list may be infinite.
- In the purely functional language Haskell, which is based on lazy evaluation, infinite lists are commonplace and can be used to express list comprehensions, such as `[square x | x <- [0 ..], square x < 10]`, where Haskell returns a partial list `0 : 1 : 4 : 9 :` $\perp$ (the system does not terminate, it expands the list "forever", searching for further elements whose square is less than 10).
- For lazy lists, recursions can go on forever. Instead of termination of a program, we can only ask whether the program generates each finite part of its result in finite time.
- In ML infinite lists are called **sequences** (sequences in Lisp are not infinite lists). Traditionally, infinite lists are called *streams* (which today typically denote input/output channels).
- Typically (Haskell, Lisp, ML) for all list functionals there are corresponding functionals for sequences.
- In ML lazy lists can be expressed, but treatment of lazy lists is a bit problematic, because ML uses strict evaluation. Lazy lists are realized by representing the tail of a list by a function in order to delay evaluation.

```
datatype 'a seq = Nil
                | Cons of 'a * (unit -> 'a seq);
```

- Like a list, a sequence either is empty or contains a head and a tail. The empty sequence is `Nil` and a non-empty sequence has the form `Cons(x,xf)`, where `x` is the head and `xf` is a function to compute the tail!

12.1 The Type ‘seq’ and Its Primitive Functions

```
datatype 'a seq = Nil
                | Cons of 'a * (unit -> 'a seq);

exception Empty;

  fun hd (Cons(x,xf)) = x
    | hd Nil = raise Empty;

  fun tl (Cons(x,xf)) = xf()
    | tl Nil = raise Empty;

  fun cons(x,xq) = Cons(x, fn()=>xq);

  fun fromList l = foldr cons Nil l;
```

- `cons(x, E)` is not evaluated lazily: ML evaluates E, yielding a result xq, and returns `Cons(x, fn() => xq)`. The `fn` inside `cons` does not delay evaluation of the tail. Only use `cons` where lazy evaluation is not required, for example in `fromlist` which converts a list into a sequence.

- Delay of evaluation is realized with `Cons(x, fn() => E)`.

```
- fun from k = Cons(k, fn()=> from(k+1));
> val from = fn : int -> int seq
- from 1;
> val it = Cons(1, fn) : int seq
- tl it;
> val it = Cons(2, fn) : int seq
- tl it;
> val it = Cons(3, fn) : int seq
- tl it;
> val it = Cons(4, fn) : int seq
```

Function `take` returns the first n elements of a sequence xq as a list:

```
fun take (xq, 0) = []
    | take (Nil, n) = raise Subscript
    | take (Cons(x,xf), n) = x :: take (xf(), n-1);
```

Evaluation of `take(from 30, 2)`:

```
take(from 30,2)
=> take(Cons(30, fn()=>from(30+1)), 2)
=> 30 :: take(from(30+1), 1)
=> 30 :: take(Cons(31, fn()=>from(31+1)), 1)
=> 30 :: 31 :: take(from(31+1),0)
=> 30 :: 31 :: take(Cons(32,fn()=>from(32+1)),0)
=> 30 :: 31 :: []
=> [30,31]
```

- Datatype α `seq` is not really lazy in ML:
  the head of a non-empty sequence is always computed (such as 32 in the example above, which is not used);
  inspecting the tail repeatedly evaluates it repeatedly

12.2 Elementary Sequence Processing

- For a function on sequences to be computable, each finite part of the output must depend on a finite part of the input.
- Example: Squaring a sequence of integers one by one: the tail of the output, when evaluated, applies function *squares* to the tail of the input.
- Other examples: add pairs of elements of two sequences, append, interleaving two sequences, ...

```
- fun squares Nil : int seq = Nil
   | squares (Cons(x, xf)) = Cons(x*x, fn()=> squares(xf()));
> val squares = fn : int seq -> int seq
- squares (from 1);
> Cons(1, fn) : int seq
- take(it, 10);
> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100] : int list
```

12.3 Functionals on Sequences

- List functional can be generalized to sequences.
- `squares` is an instance of functional *map*.
- `from` is an instance of the functional *iterates* which generates sequences of the form $[x, f(x), f(f(x)), \ldots, f^k(x), \ldots]$.
- *filter* successively calls the tail function until an element is found which satisfies the given predicate. If no such element exists the function will not terminate!

```
fun map f Nil  = Nil
  | map f (Cons(x,xf)) = Cons(f x, fn()=> map f (xf()));

fun filter pred Nil = Nil
  | filter pred (Cons(x,xf)) =
        if pred x then Cons(x, fn()=> filter pred (xf()))
                  else filter pred (xf());

fun iterates f x = Cons(x, fn()=> iterates f (f x));

filter (fn n => n mod 10 = 7) (from 50);
> Cons(57, fn) : int seq
take (it, 8);
> [57, 67, 77, 87, 97, 107, 117, 127] : int list

iterates(secr op/ 2.0) 1.0;
> Cons(1.0, fn) : real seq
take(it, 5);
> [1.0, 0.5, 0.25, 0.125, 0.0625] : real list
```

12.4 A Structure for Sequences

```
datatype 'a seq = Nil
                | Cons of 'a * (unit -> 'a seq);

(*Type seq is free in this signature!*)
signature SEQUENCE =
  sig
  exception Empty
  val cons : 'a * 'a seq -> 'a seq
  val null : 'a seq -> bool
  val hd : 'a seq -> 'a
  val tl : 'a seq -> 'a seq
  val fromList  : 'a list -> 'a seq
  val toList  : 'a seq -> 'a list
  val take : 'a seq * int -> 'a list
  val drop : 'a seq * int -> 'a seq
  val @  : 'a seq * 'a seq -> 'a seq
  val interleave : 'a seq * 'a seq -> 'a seq
  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val filter : ('a -> bool) -> 'a seq -> 'a seq
  val iterates : ('a -> 'a) -> 'a -> 'a seq
  val from : int -> int seq
  end;

structure Seq : SEQUENCE =
  struct
  exception Empty;
  fun hd (Cons(x,xf)) = x
    | hd Nil = raise Empty;
  fun tl (Cons(x,xf)) = xf()
    | tl Nil = raise Empty;
  fun cons(x,xq) = Cons(x, fn()=>xq);
  fun null (Cons _) = false
    | null Nil      = true;
  fun fromList l = foldr cons Nil l;
  fun toList Nil = []
    | toList (Cons(x,xf)) = x :: toList (xf());
```

```
fun take (xq, 0) = []
  | take (Nil, n) = raise Subscript
  | take (Cons(x,xf), n) = x :: take (xf(), n-1);
fun drop (xq, 0) = xq
  | drop (Nil, n) = raise Subscript
  | drop (Cons(x,xf), n) = drop (xf(), n-1);
fun Nil @ yq = yq
  | (Cons(x,xf)) @ yq = Cons(x, fn()=> (xf()) @ yq);
fun interleave (Nil,    yq) = yq
  | interleave (Cons(x,xf), yq) =
        Cons(x, fn()=> interleave(yq, xf()));
(** functionals for sequences **)
fun map f Nil  = Nil
  | map f (Cons(x,xf)) = Cons(f x, fn()=> map f (xf()));
fun filter pred Nil = Nil
  | filter pred (Cons(x,xf)) =
        if pred x then Cons(x, fn()=> filter pred (xf()))
                  else filter pred (xf());
fun iterates f x = Cons(x, fn()=> iterates f (f x));
fun from k = Cons(k, fn()=> from(k+1));
end;
```

12.5 Elementary Applications of Sequences

Example: Calculating Primes using the Sieve of Eratosthenes

- Start with the sequence [2, 3, 4, 5, 6, ...]
- Take 2 as a prime. Delete all multiples of 2, since they cannot be prime. This leaves sequence [3, 5, 7, 9, 11, ...]
- Take 3 as a prime and delete its multiples. This leaves the sequence [5, 7, 11, 13, 17, ...]
- Take 5 as a prime ...
- At each stage, the sequence contains those numbers not divisible by any of the primes generated so far. Therefore, its head is a prime, and the process can continue indefinitely.
- `sift` deletes multiples from a sequence and `sieve` repeatedly sifts a sequence.

```
fun sift p = Seq.filter (fn n => n mod p <> 0);
fun sieve (Cons(p,nf)) = Cons(p, fn()=> sieve (sift p (nf())));
val primes = sieve (Seq.from 2);
Seq.take (primes, 25);
```

12.6 Search Strategies on Infinite Lists

- Each search strategy is a curried function, getting a function `next: 'a -> ' a list` which returns the next list of subtrees for a node x.
- The most primitive algorithms just enumerate all nodes in some order. Solutions can be identified using functional `Seq.filter` with a suitable predicate on nodes. (generate and test)
- filter all *legal* solutions; optional: find all/an *optimal* solution(s).
- More efficient versions of search can include a predicate to recognize solutions and terminate after a solution is found.
- A strategie which combines depth-first and breadth-first is iterative deepening: It searches depth first to a depth d, returning all solutions, then to depth $2d$, then to depth $3d$, and so on.
- Heuristic search strategies as best-first search or A^* incorporate estimations about the distance of a node from the desired goal state.

```
fun depthFirst next x =
    let fun dfs [] = Nil
          | dfs(y::ys) = Cons(y, fn()=> dfs(next y @ ys))
    in  dfs [x]  end;

fun breadthFirst next x =
    let fun bfs [] = Nil
          | bfs(y::ys) = Cons(y, fn()=> bfs(ys @ next y))
    in  bfs [x]  end;
```

13 Reasoning about Functional Programs

- Software crisis in the 70's (and after): hard to cope with increasingly complex software projects; delayment (cancellation) of systems, cost explosion. Several methodologies were proposed to deal with the software crisis:
  - **Structured Programming**: Organization of programs in simple parts with simple interfaces. Using abstract datatypes/modules. (see later section)
  - **Declarative Programming**: (functional or logic) Expressing computations directly in mathematics, making the machine state invisible, understanding/testing one expression at a time. (that's what we do in this lecture)
  - **Program Correctness Proofs**: Verification oder Derivation (synthesis) of programs with respect to a specification. (our topic)

13.1 Functional Programs and Mathematics

- Remember FP: Proofs on simple programs can be performed by rewriting, applying the definitions of the built-in functions and simple mathematical laws (as associativity, commutativity). Theorems about recursive programs can be proved using induction.
- With different kinds of induction we can verify programs over simple types (mathematical or complete induction), over lists and trees (structural induction), for well-founded recursion (well-founded induction).
- Equality of syntactically different forms of functions can be shown either by rewriting, applying mathematical laws (extensional equality) or by calculating normal forms (intensional equality). Equality cannot generally be shown.
- Properties of whole classes of functions (recursive program schemes) can be proven with well-founded induction.
- The (denotational) semantics of recursive functions can be proved using domain theory and fixed point theorems.

For the most part, we will restrict the proofs to functions which are assumed to terminate (no proofs about infinite lists, no lazy evaluation). Reasoning about programs which do not always terminate involve domain theory.

13.2 Limitations and Advantages of Verification

- Verification assumes that the hardware is infallible. Typically, questions about arithmetic overflow, rounding errors, running out of store are ignored (but can be included).
- A specification may be incomplete or wrong because *design requirements* are hard to formalize. Satisfying a faulty specification will not satisfy the customer. Verification: Did we build the product right? versus Validation: Did we build the right product?
- Proofs may contain errors: Automated theorem proving can reduce but not eliminate the likelihood of errors. All human efforts may have flaws, many errors have been discovered in theorem provers, in proof rules, in published proofs.
- Formal Proof is tedious: one must take great pains, even to prove very elementary things. Imagine verifying a compiler!
- But: Writing formal specifications reveal ambiguities and inconsistencies in the design requirements. The painstaking work of verification yields rewards: an attempted proof often can pinpoint the error in a program.
- Specification and verification yield a fuller knowledge of the program and its task.
- Additionally, testing (validation) is always necessary. It is the only way to investigate whether the computational model and the formal specification reflect the real world. However, while testing can detect errors, it cannot guarantee success (testing is necessarily incomplete).
- Alternative effort: Synthesis a program automatically from a specification (KIDS, D. Smith).
- Correspondence between proofs and programs: (Curry-Howard Isomorphism) If we can extract programs from proofs, then by constructive theorem proving we obtain verified programs. $\forall i \exists o[p(i) \Rightarrow \phi(i,o)]$ written constructively $\forall i[p(i) \Rightarrow \phi(i, f(i))]$ where i stands for input values, o for output values, p for preconditions on i, ϕ for the desired input/output relation and f (introduced by Skolemization) is the searched for program. See for example: S. Thompson (1991). Type Theory and Functional Programming. Addison-Wesley.

13.3 Mathematical Induction and Complete Induction

- Mathematical Induction: Reduction of a problem $\phi(k)$ to $\phi(k-1)$. Complete Induction: Reduction of a problem $\phi(k)$ to the k subproblems $\phi(0)$, $\phi(1)$, ..., $\phi(k-1)$. (includes math. ind. as special case)

13.3.1 Mathematical Induction

Induction Scheme:

$$\frac{\begin{array}{cc} [\phi(k)] & (induction\ hypothesis) \\ \phi(0)\ \ \phi(k+1) & (base\ case\ \ and\ \ induction\ step) \end{array}}{\phi(n)}$$

- Above the line: premises, below: conclusion.
- The induction hypothesis can be assumed true while proving the induction step.
- For more complex proofs: k must not appear in any other induction hypothesis, it is a new variable, which stands for an arbitrary value.

Example:

Theorem: Every natural number has the form $2m$ or $2m+1$ for some natural number m.
Formally: $\forall n\ \exists m\ n = 2m \vee n = 2m + 1$.

Proof:

- **Induction Hypothesis**: $\exists m\ k = 2m \vee k = 2m + 1$
- **Base Case** ($k = 0$): $\exists m\ 0 = 2m \vee 0 = 2m + 1$
  holds with $m = 0$
- **Induction Step**: ($k \to k + 1$): $\exists m'\ k + 1 = 2m' \vee k + 1 = 2m' + 1$
  If $k = 2m$ then $k + 1 = 2m + 1$, so $m = m'$.
  If $k = 2m + 1$ then $k + 1 = 2m + 2 = 2(m + 1)$, so $m' = m + 1$. □

```
datatype evenodd = Even | Odd;
fun half 0 = (Even, 0)
  | half n = (case half (n-1) of (Even, m) => (Odd, m)
                               | (Odd, m) => (Even, m+1));
```

13.3.2 Complete Induction

Induction Scheme:

$$\frac{\begin{array}{ll}[\forall i < k \;\; \phi(i)] & (induction\ hypothesis)\\ \phi(k) & (induction\ step)\end{array}}{\phi(n)}$$

Theorem: Every natural number $n \geq 2$ can be written as a product of prime numbers, $n = p_1 \cdots p_k$.

Proof:

- **Case 1:** n is prime. Then $k = 1$ (n = n).
- **Case 2:** n is not prime.
  n is divisible by some natural number m such that $1 < m < n$.
  Since $m < n$ and $n/m < n$, the induction hypothesis can be used twice:
  $m = p_1 \cdots p_k$ and $n/m = q_1 \cdots q_l$.
  Now $n = m \times (n/m) = p_1 \cdots p_k q_1 \cdots q_l$. □

13.3.3 Program Verification with Mathematical Induction

Theorem: For every natural number n, $facti(n, 1) = n!$.
Relation between a ML function and mathematics.

```
fun facti(n, p) = if n=0 then p else facti(n-1, n*p);
```

Proving the more general formula $\forall n \forall p \; facti(n, p) = n! \times p$ and setting $p = 1$.

- **Base Case:** $\forall p \; facti(0, p) = 0! \times p$.
  holds because: $facti(0, p) = p = 1 \times p = 0! \times p$.
- **Induction Step:** Show $\forall p \; facti(n + 1, p) = (n + 1)! \times p$
  $facti(n + 1, p) = facti(n, (n + 1) \times p)$ (unfolding $facti$)
  $= n! \times ((n + 1) \times p)$ (induction hyp.)
  $= (n! \times (n + 1)) \times p$ (associativity)
  $= (n + 1)! \times p$ (factorial). □

Comments:

- This proof helps us to understand the role of p (accumulator) in $facti(n, p)$.
- The induction formula is analogous to a loop invariant in procedural program verification (Hoare calculus).
- Since the proof depends on associativity for multiplication, it suggests that $facti(n, 1)$ computes $n!$ by multiplying the number in the order $(1 \times (2 \times (3 \ldots (n \times 1) \ldots)))$.
- Later we shall generalize this to a general theorem about transforming recursive functions into iterative functions!

13.4 Structural Induction

Induction Scheme for Lists:

$$\frac{\begin{array}{cc}[\phi(ys)] & (induction\ hypothesis)\\ \phi([\,])\ \ \phi(y :: ys) & (base\ case\ \&\ \ ind.\ step)\end{array}}{\phi(xs)}$$

Induction Scheme for Trees:

$$\frac{\begin{array}{cc}[\phi(t_1), \phi(t_2)] & (induction\ hypothesis)\\ \phi(Lf)\ \ \phi(Br(x, t_1, t_2)) & (base\ case\ \ \&\ \ ind.\ step)\end{array}}{\phi(t)}$$

- Structural induction is a generalization of mathematical induction to lists and trees.
- Mathematical Induction is based on the datatype of natural numbers: `datatype num = 0 | succ of num;`.
  Base: 0 is a number; Step: If k is a number, so is $succ(k)$.
- Analogy for lists and trees.

Example:

```
fun nlength [] = 0
  | nlength (x::xs) = 1 + nlength xs;
fun [] @ ys = ys
  | (x::xs) @ ys = x :: (xs @ ys);
```

Theorem: For all lists xs and ys, *nlength(xs @ ys) = nlength(xs) + nlength(ys)*.

- **Base Case:** *nlength([] @ ys) = nlength[] + nlength ys*.
  nlength([] @ ys) = nlength ys (append)
  = 0 + nlength ys (arithmetic)
  = nlength[] + nlength ys (nlength). □
- **Induction Step:** Show for all x and xs that
  nlength((x::xs) @ ys) = nlength(x::xs)+nlength(ys).

  nlength((x::xs) @ ys)
  = nlength(x::(xs @ ys)) (append)
  = 1 + nlength(xs @ ys) (nlength)
  = 1 + (nlength xs + nlength ys) (ind. hyp.)
  = (1 + nlength xs) + nlength ys (associativity)
  = nlength(x::xs) + nlength ys (nlength). □

- The proof brings out the correspondence between inserting list elements and counting them.
- Induction on xs works because base case and induction step can be simplified using function definition. Induction on ys leads nowhere!!
- Application of the definition of a function: replacing head (function call) by body with app. substitution of parameters is called **unfolding**, replacing the body by the app. function call is called **folding** (Burstall and Darlington).

Example:

```
fun reflect Lf = Lf
  | reflect (Br(v, t1, t2)) = Br(v, reflect t2, reflect t1);
```

Theorem: For every binary tree t, $reflect(reflect\ t) = t$.
Proof:

- **Base Case:** $reflect(reflect\ Lf) = Lf$
  holds by definition $reflect(reflect\ Lf) = reflect\ Lf = Lf$.
- **Induction Step:**
  Two induction hypotheses: $reflect(reflect\ \ t_1) = t_1$ and $reflect(reflect\ t_2) = t_2$.

  To prove: $reflect(reflect(Br(x, t_1, t_2))) = Br(x, t_1, t_2)$
  $reflect(reflect(Br(x, t_1, t_2)))$
  $= reflect(Br(x, reflect\ t_2, reflect\ t_1))$ (reflect)
  $= Br(x, reflect(reflect\ t_1), reflect(reflect\ t_2))$ (reflect)
  $= Br(x, t_1, reflect(reflect\ t_2))$ (ind. hyp.)
  $= Br(x, t_1, t_2)$ (ind. hyp.) □

Remark: For proving theorems about data types involving functions, this set-theoretical way of proofs is not sufficient. We need domain theory, where we can prove theorems about the domain $D \to D$ of continuous functions!

13.5 Equality of Functions and Theorems about Functionals

- We use the notion of functions as values and the set theoretical view of functions.
- **Law of extensionality:** Two functions f and g are equal if $f(x) = g(x)$ for all x (of suitable type) and f and g must have identical domains.
  - There is no general computable method of testing whether two functions are extensionally equal. In ML there does not exist an equality predicate for functions (in Lisp intensional equality is used).
  - Extensional equality can be shown if replacing f and g does not affect the value of any application of f.
  - This law requires that functions terminate. Generally, $\perp$ denotes an undefined function value and $\lambda x.\perp$ denotes a function which never terminates. In effect, in both cases the result is $\perp$.
  - Example: `fun double1(n) = 2*n;`, `fun double2(n) = n*2;`, `fun double3(n) = n + n;`
- Intensional Equality: Two functions are equal iff their definitions are identical. (If all functions can be rewritten in a normal form, often – not always – semantically identical functions have normal forms which are equal up to variable renaming.)

13.5.1 Theorems about Functionals

Theorem: For all functions f, g, h (of appropriate type), $(f \circ g) \circ h = f \circ (g \circ h)$.
Proof: By the law of extensionality it is enough to show:
$((f \circ g) \circ h)x = (f \circ (g \circ h))x$ for all x.
$((f \circ g) \circ h)x$
$= (f \circ g)(hx)$
$= f(g(hx))$
$= f((g \circ h)x)$
$= (f \circ (g \circ h))x$.
Each step holds by the definition of composition: $(f \circ g)x \equiv f(gx)$.

Theorem: For all functions f and g, $(map\ f) \circ (map\ g) = map(f \circ g)$. (This theorem can be used to avoid computing intermediate lists.)
Proof: By the law of extensionality, this equality holds if $(map\ f) \circ (map\ g)xs = map(f \circ g)xs$ for all lists xs. Simplification (def. of $\circ$): $map\ f(map\ g\ xs) = map(f \circ g)xs$.
Proof by structural induction:

- Base Case: $map\ f(map\ g\ []) = map\ f\ [] = [] = map(f \circ g)[]$.
- Induction Step: $map\ f(map\ g\ (x :: xs)) = map(f \circ g)(x :: xs)$.
  $map\ f(map\ g\ (x :: xs))$
  $= map\ f((g\ x) :: (map\ g\ xs))$ (map)
  $= f(g\ x) :: (map\ f(map\ g\ xs))$ (map)
  $= f(g\ x) :: (map(f \circ g)xs)$ (ind. hyp.)
  $= (f \circ g)(x) :: (map(f \circ g)xs)$ (comp.)
  $= map(f \circ g)(x :: xs)$ (map). □

13.6 Well-Founded Induction and Recursion

- For all recursive functions on lists/trees which make recursive calls on the tail/sub-trees of their argument, the type of recursion is called *structural recursion* and proofs can be based on structural induction.
- But, recursive functions can shorten lists in other ways: *maxl*, *quick sort*, *merge sort*, *transpose*, ...

```
fun maxl [m] : int  = m
  | maxl (m::n::ns) = if m>n then maxl(m::ns) else maxl(n::ns);
```

  Here, not the plain list *structure* but the *content* (attribute of the elements) must be taken into account! (Remark: These are the problematic cases for program synthesis!)
  The same is true for functions on trees: *nnf* (conversion of a proposition in negation normal form) is not structurally recursive.

- Well-founded induction is a powerful generalization of complete induction and includes most other induction rules. A special case of well-founded induction is induction on size.

Scheme for Well-founded Induction:

$$\frac{\begin{array}{ll}[\forall y' \prec y.\phi(y')] & (induction\ hypothesis)\\ \phi(y) & (ind.\ step)\end{array}}{\phi(x)}$$

- Here $\prec$ is a **well-founded order relation** over some type τ:
  The relation $\prec$ is well-founded if there exist no infinite decreasing chains.
- Examples: less-than ($<$) on natural numbers is well-founded, less-than on integers is not well-founded.
  Lexicographic ordering of pairs of natural numbers: *(i', j')* $\prec_{lex}$ *(i, j) iff i' < i* $\vee$ *(i'= i* $\wedge$ *j' < j)*.

- Recursive functions over arguments with a well-founded order relation are called **well-founded recursive functions**.
- The Boyer-Moore (1975, 1988) theorem prover is based on well-founded recursion.

13.7 Recursive Program Schemes

- Well-founded relations permit reasoning about program schemes.
- A program scheme is an 'abstract' program structure defined over classes of functions and operators.
- Example: Schemes for linear recursion and tail recursion (with $\oplus$ as associative operator and identity e):

$$f_1(x) = \ if\ p(x)\ then\ e\ else\ f_1(g\ x) \oplus x$$

$$f_2(x, y) = \ if\ p(x)\ then\ y\ else\ f_2(g\ x, x \oplus y)$$

- Well-founded relation $g(x) \prec x$ for all x with $p(x) = false$.
  $\hookrightarrow$ f_1 and f_2 terminate.

Theorem: Suppose $\oplus$ is an infix operator that is associative and has identity e, that is, for all x, y, z:
$x \oplus (y \oplus z) = (x \oplus y) \oplus z$
$e \oplus x = x = x \oplus e$.
Then, for all x and e holds $f_2(x, e) = f_1(x)$.
Proof: $\forall y. f_2(x, y) = f_1(x) \oplus y$. Then set $y = e$.
This holds by well-founded induction over $\prec$. There are two cases:

1. If $p(x) = true$ then:
   $f_2(x, y) = y$ (def. of f_2)
   $= e \oplus y$ (identity)
   $= f_1(x) \oplus y$ (def. of f_1).

2. If $p(x) = false$ then:
   $f_2(x, y) = f_2(g\ x, x \oplus y)$ (def. of f_2)
   $= f_1(g\ x) \oplus x \oplus y$ (ind.hyp.)
   $= f_1(x) \oplus y$ (def. of f_1). □

- The induction hypothesis applies because $g(x) \prec x$. We have implicitly used associativity of $\oplus$.
- Thus, we can transform a linear recursive function into a tail recursive function with an accumulator!
- The theorem applies to the computation of factorials:
  $e = 1$, $\oplus = \times$, $g(x) = x - 1$, $p(x) = (x = 0)$, $\prec = <$
- In general, such proofs involve domain theory.

14 Domain Theory and Fixpoint Semantics

14.1 Concept for Semantics of Programming Languages

- Like every (natural) language, a programming language has a syntax (a grammar, specified in BNF, to construct well-formed expressions), a semantic (the meaning of the primitives of a language and of the user-defined programs built over them; often only given informally), and pragmatics (how the various features of a language are intended to be used, the implementation of the language, etc.).
- Formal, mathematical semantics allows machine independent specification of behavior, reasoning about programs and can help to bring to light abiguities and unforseen subtleties in programming language constructs.
- **Axiomatic Semantics:** Meanings for program phrases defined indirectly via the axioms and rules of some logic of program properties (e.g., Hoare calculus).
- **Operational Semantics:** Meanings for program phrases defined in terms of the steps of computation they can take during program execution. (e.g., Backus Algebra of FP; evaluation rule for ML, ...)
- **Denotational Semantics:** Meanings for program phrases defined abstractly as elements of some suitbale mathematical structure.
- Analogy to natural language semantics:
  "A chair is to sit on", the meaning of "x is left of y" is the knowledge needed to reach x starting from y (operational, cf. "procedural semantics for mental models", Johnson-Laird).
  The meaning of chairs is the set of all chairs, the meaning of "the chair is red" is the intersection of the set of red objects and the set of chairs (denotational, cf. model-theoretic semantics, Tarski).

Sets and Domains: (Field & Harrison (1986), Appendix B)

- Set-Theory: e. g. the set of integers $\mathcal{N}$, the set of booelan values $Bool$.
  Domain-Theory: introduce an additional, unique element $\perp$ which represents the undefined value or non-termination. $\mathcal{N}_\perp$ and $Bool_\perp$ are domains.

14.2 Semantics of Backus FFP

14.2.1 Syntax

- Predefined set of atoms A with T (true), F (false), ϕ (empty set), numbers, ... as elements of A. (A also contains primitive functions.)
- Objects $\mathcal{O}$:
  - $\perp$ is an object.
  - Every atom is an object.
- Expressions $\mathcal{E}$:
  - An object is an expression which has no application as a sub-expression.
  - If $x_1, \ldots, x_n$ are expressions then sequence $\langle x_1, \ldots, x_n \rangle$ is an expression. (ϕ is atom and sequence; if some x_i in a sequence is $\perp$, then the sequence is $\perp$).
  - If x and y are expressions, then application $x : y$ is an expression.
- All objects and expressions are formed by finite use of the above rules.

Depending on the set of A, different functional programming languages can be created. The abstract language is called FFP with FP as a special case.

14.2.2 Differences between FP and FFP

- FFP allows user-defined functionals (as well as functions).
- In FFP operators are objects, therefore, a function as 'apply' can be defined as:
  `apply:⟨x,y⟩ = (x:y)` where x is an object as well as an expression (for example the primitive function `NULL`).

14.2.3 Meaning of Expressions

- Every FFP expression e has a meaning μe which is always an object.
- μe is found by repeatedly replacing each innermost application in e by its meaning.
- If this process is non-terminating, the meaning of e is $\perp$.
- The association between objects and the functions they represent is given by an representation function ρ.
- μ and ρ belong not to FFP but to the description of FFP.

- $\mu \in [expressions \rightarrow objects]$.
- If x is an object, $\mu x = x$.
- If e is an expression and $e = \langle e_1, \ldots, e_n \rangle$, then $\mu e = \langle \mu e_1, \ldots, \mu e_n \rangle$.
- $\rho \in [expressions[expressions \rightarrow expressions]]$.
- For any expression e $\rho e = \rho(\mu e)$.
- If x is an object and e an expression, then $\rho x : e = \rho x : (\mu e)$.
- If x and y are objects, then $\mu(x : y) - \mu(\rho x : y)$.
  The meaning $\mu(x : y)$ of an FFP application $(x : y)$ is found by applying ρx, the function represented by x, to y and then finding the meaning of the resulting expression (which is usally an object). The meaning of an object is the object itself.

Meta-Composition Rule: $(\rho\langle x_1, \ldots, x_n \rangle) : y = (\rho x_1) : \langle \langle x_1, \ldots, x_n \rangle, y \rangle$.
(x_1 is a functional form and $x_2, \ldots, x_n$ are its parameters.)
Example: $Def\ \rho CONST = \equiv 2 \circ 1$
$(\rho\langle CONST, x \rangle) : y = (\rho CONST) : \langle \langle CONST, x \rangle, y \rangle = 2 \circ 1 : \langle \langle CONST, x \rangle, y \rangle$
$= x$.
Meta-composition is introduced to avoid recursive function definitions.

Example: $\rho MLAST \equiv null \circ tl \circ 2 \rightarrow 1 \circ 2; apply \circ [1, tl \circ 2]$

$\mu(\langle MLAST \rangle : \langle A, B \rangle)$
$= \mu(\rho MLAST : \langle \langle MLAST \rangle, \langle A, B \rangle \rangle)$
$= \mu(apply \circ [1, tl \circ 2] : \langle \langle MLAST \rangle, \langle A, B \rangle \rangle)$
$= \mu(apply : \langle \langle MLAST \rangle, \langle B \rangle \rangle)$
$= \mu(\langle MLAST \rangle : \langle B \rangle)$
$= \mu(\rho MLAST : \langle \langle MLAST \rangle, \langle B \rangle \rangle)$
$= \mu(1 \circ 2 : \langle \langle MLAST \rangle, \langle B \rangle \rangle)$
$= B.$
$\hookrightarrow$ denotational semantics

14.3 Semantics of Recursive Functions

What is the meaning of $f = E(f)$? (a rose is a rose is a rose)

Basic idea of fixpoint semantics:

- Use (linear) expansion and define a sequence of partial unfoldings; in the limit, the interpretation of these unfoldings represent the meaning of the function.

Example: *factorial*
$f^0 = \bot$
$f^1 = \ if\ (x = 0)\ then\ 1\ else\ x \times \bot$
$f^2 = \ if\ (x = 0)\ then\ 1\ else\ x \times \ if\ (x - 1 = 0)\ then\ 1\ else\ (x - 1) \times \bot$
$f^3 = \ if\ (x = 0)\ then\ 1\ else\ x \times \ if\ (x - 1 = 0)\ then\ 1\ else\ (x - 1) \times \ if\ (x - 1 - 1 = 0)\ then\ 1\ else\ (x - 1 - 1) \times \bot$
...
$f^n = \ if\ (x = 0)\ then\ 1\ else\ ...(x - \underbrace{1 - \ldots - 1}_{n-times})...$

- The expansion f^k is defined for the first k inputs, that is, $i = \{0, 1, \ldots, k-1\}$. In the limit $k \to \infty$ the non-recursive function definition is given for all possible inputs.
- In domain theory we consider the Kleene-sequence of such partial mappings. We must show that for the strict function f there exists a complete partial order and that a mapping Φ from strict functions to strict functions is order-preserving and strict.

14.4 Fixpoint Semantics

Fixpoint theory in general concerns the following question: Given an ordered set P and an order-preserving map $\Phi : P \to P$, does there exist a fixpoint for Φ? A point $x \in P$ is called fixpoint if $\Phi(x) = x$.
The following introduction of fixpoint semantics closely follows Davey and Priestly (1990). We first introduce some basic concepts and notations. Then, we present the fixpoint theorem for complete partial orders. Finally, we give an illustration for the *factorial* function.

Definition 1 (Map) *Let X be a set and consider a map $f : X \to X$. f assigns a member $f(x) \in X$ to each member $x \in X$ and is determined by its graph, $\mathbf{graph} f = \{(x, f(x)) \mid x \in X\}$ with $\mathbf{graph} f \subseteq X \times X$.*
A partial map *is a map $\sigma : S \to X$ where $S \subset X$. With $\mathbf{dom}\ \sigma = S$ we denote the domain of σ. The set of partial maps on X is denoted $(X \stackrel{\circ}{\to} X)$ and is ordered in the following way: given $\sigma, \tau \in (X \stackrel{\circ}{\to} X)$, define $\sigma \leq \tau$ iff $\mathbf{dom}\ \sigma \subseteq \mathbf{dom}\ \tau$ and $\sigma(x) = \tau(x)$ for all $x \in \mathbf{dom}\ \sigma$. A subset G of $X \times X$ is the graph of a partial map iff $\forall s \in X : (s, x) \in G$ and $(s, x') \in G \Rightarrow x = x'$.*

Definition 2 (Order-Preserving Map) *Let P and Q be ordered sets. A map $\phi : P \to Q$ is order-preserving (or monotone) if $x \leq y$ in P implies $\phi(x) \leq \phi(y)$ in Q.*

Definition 3 (Bottom Element) *Let P be an ordered set. The least element of P, if such exists, is called the* bottom element *and denoted $\perp$.*
Given an ordered set P (with or without $\perp$), we form $P_\perp$ (called "P lifted") as follows: Take an element $\mathbf{0} \notin P$, construct $P_\perp = P \cup \{\mathbf{0}\}$ and define $\leq$ on $P_\perp$ by $x \leq y$ iff $x = \mathbf{0}$ or $x \leq y$ in P.

Definition 4 (Order-Isomorphism between Partial and Strict Maps) *With each $\pi \in (S \stackrel{\circ}{\to} S)$ we associate a map $\psi(\pi) : S \to S_\perp$, given by $\psi(\pi) = \pi_\perp$ where*

$$\pi_\perp(x) = \begin{cases} \pi(x) & if\ x \in \mathbf{dom}\ \pi \\ \mathbf{0} & otherwise. \end{cases}$$

Thus ψ is a map from $(S \stackrel{\circ}{\to} S)$ to $(S \to S_\perp)$. We have used the extra element $\mathbf{0}$ to convert a partial map on S into a total map. ψ sets up an order-isomorphism between $(S \stackrel{\circ}{\to} S)$ and $(S \to S_\perp)$.

Definition 5 (Supremum) *Let P be an ordered set and let $S \subseteq P$. An element $x \in P$ is an* upper bound *of S if $s \leq x$ for all $s \in S$. x is called* least upper bound *of S if*

1. *x is an upper bound of S, and*
2. *$x \leq y$ for all upper bounds y of S.*

Since least elements are unique, the least upper bound is unique if it exists. The least upper bound is also called the supremum *of S and is denoted $\mathbf{sup}S$, or $\bigvee S$ ("join of S"), or $\bigsqcup S$ if S is directed.*

Definition 6 (CPO) *An ordered set P is a CPO (complete partial order) if*

1. *P has a bottom element $\bot$,*
2. *$\mathbf{sup}D$ exists for each directed sub-set D of P.*

The simplest example of a directed set is a chain, such as $0 \leq succ(0) \leq succ(succ(0)) \leq succ(succ(succ(0))) \leq \ldots$. Fixpoint theorems are based on ascending chains.

Definition 7 (Continuous and Strict Maps) *A map $\phi : P \to Q$ is continuous, if for every directed set D in P: $\phi(\bigsqcup D) = \bigsqcup(\phi(D))$. Every continuous map is order preserving.*
A map $\phi : P \to Q$ such that $\phi(\bot) = \bot$ is called strict.

Definition 8 (Fixpoint) *Let P be an ordered set and let $\Phi : P \to P$ be a map. We say $x \in P$ is a* fixpoint *of Φ if $\Phi(x) = x$. The set of all fixpoints of Φ is denoted by $\mathbf{fix}(\Phi)$; it carries the induced order. The least element of $\mathbf{fix}(\Phi)$, when it exists, is denoted by $\mu(\Phi)$.*
The n-fold composite, Φ^n, of a map $\Phi : P \to P$ is defined as follows: Φ^n is the identity map if $n = 0$ and $\Phi^n = \Phi \circ \Phi^{n-1}$ for $n \geq 1$. If Φ is order preserving, so is Φ^n.

Theorem 1 (CPO Fixpoint)
Let P be a complete partial order (CPO), let $\Phi : P \to P$ be an order-preserving map and define $\alpha = \bigsqcup_{n \geq 0} \Phi^n(\bot)$.

1. *If $\alpha \in fix(\Phi)$, then $\alpha = \mu(\Phi)$.*

2. *If Φ is continuous then the least fixpoint $\mu(\Phi)$ exists and equals α.*

Proof 1 (CPO Fixpoint)

1. *Certainly $\bot \leq \Phi(\bot)$. Applying the order preserving map Φ^n, we have $\Phi^n(\bot) \leq \Phi^{n+1}(\bot)$ for all n. Hence we have a chain $\bot \leq \Phi(\bot) \leq \ldots \leq \Phi^n(\bot) \leq \Phi^{n+1}(\bot) \leq \ldots$ in P. Since P is a CPO, $\alpha = \bigsqcup_{n\geq 0} \Phi^n(\bot)$ exists. Let β be any fixpoint of Φ. By induction, $\Phi^n(\beta) = \beta$ for all n. We have $\bot \leq \beta$, hence we obtain $\Phi^n(\bot) \leq \Phi^n(\beta) = \beta$ by applying Φ^n. The definition of α forces $\alpha \leq \beta$. Hence if α os a fixpoint then it is the least fixpoint.*
2. *It will be enough to show that $\alpha \in \mathbf{fix}(\Phi)$. We have*

$$\begin{array}{lll} \Phi(\bigsqcup_{n\geq 0} \Phi^n(\bot)) & = \bigsqcup_{n\geq 0} \Phi(\Phi^n(\bot)) & (since\ \Phi\ is\ continuous) \\ & = \bigsqcup_{n\geq 1} \Phi^n(\bot) & \\ & = \bigsqcup_{n\geq 0} \Phi^n(\bot) & (since \bot \leq \Phi^n(\bot)\ for\ all\ n). \end{array}$$

Consider the factorial function $\mathbf{facu}(x) = x!$ with its recursive definition:

$$\mathbf{facu}(x) = \begin{cases} 1 & if\ x = 0 \\ x \cdot \mathbf{facu}(x-1) & if\ x \geq 1. \end{cases}$$

To each map $f : \mathcal{N}_0 \to \mathcal{N}_0$ we may associate a new map $\bar{f}$ given by:

$$\bar{f}(x) = \begin{cases} 1 & if\ x = 0 \\ x \cdot f(x-1) & if\ x \geq 1. \end{cases}$$

The equation satisfied by $\mathbf{facu}$ can be reformulated as $\Phi(f) = f$, where $\Phi(f) = \bar{f}$. The *entire* factorial function cannot be unwound from its recursive specification in a finite number of steps. However we can, for each $n = 0, 1, \ldots$, determine in a finite number of steps the *partial* map f_n which is a restriction of $\mathbf{facu}$ to $\{0, 1, \ldots, n\}$. The graph of f_n is $\{(0, 1), (1, 1), \ldots, (n, n!)\}$. Therefore, to accommodate approximations of $\mathbf{facu}$ we can consider all partial maps on $\mathcal{N}_0$ and regard $\bar{f}$ as having the domain $\{0\} \cup \{k \mid k-1 \in \mathbf{dom} f\}$. Similarly, we can work with all maps from $\mathcal{N}_0$ to $(\mathcal{N}_0)_\bot$ and take $\bar{f} = \bot$ precisely when $f(k-1) = \bot$. The factorial function can be regarded as a solution of a recursive equation $\Phi(f) = f$,

where $f \in (\mathcal{N}_0 \stackrel{\circ}{\to} \mathcal{N}_0)$ or equivalently can be regarded as a solution of a recursive equation $\Phi(f) = f$, where $f \in (\mathcal{N}_0 \to (\mathcal{N}_0)_\perp)$:

$$\Phi(f)(x) = \begin{cases} 1 & if\ x = 0 \\ x \cdot f(x-1) & if\ x \geq 1. \end{cases}$$

The domain P of Φ as given in the fixpoint theorem above, is $(\mathcal{N}_0 \stackrel{\circ}{\to} \mathcal{N}_0)$. That is, Φ maps partial maps on partial maps. The bottom element of $(\mathcal{N}_0 \stackrel{\circ}{\to} \mathcal{N}_0)$ is $\emptyset$. This corresponds to that map in $(\mathcal{N}_0 \to (\mathcal{N}_0)_\perp)$ which sends every element to $\perp$. We denote this map by $\perp$.
It is obvious that Φ is order preserving: We have graph $\Phi(\perp) = \{(0,1)\}$, graph $\Phi^2(\perp) = \{(0,1),(1,1)\}$ and so on. An easy induction confirms that $f_n = \Phi^n(\perp)$ for all n where $\{f_n\}$ is the sequence of partial maps ("Kleene sequence") which approximate $\mathbf{facu}$. Taking the union of all graphs, we see that $\bigsqcup_{n\geq 0} \Phi^n(\perp)$ is the map $\mathbf{facu} : x \to x!$ on $\mathcal{N}_0$. This is the least fixpoint of Φ in $(\mathcal{N}_0 \stackrel{\circ}{\to} \mathcal{N}_0)$.

15 Abstract Types and Functors

Remember:

- Modules in ML are structures (implementation modules) and signatures (definition modules).
- A structure can be used to pack definitions of related types, values and functions.
- A signature gives a specification for a class of structures by listing names and types of each component.
- If a structure is declared (without signature constraint), its signature is inferred, consisting of all components of a structure. That is, everything is visible and usable outside the structure.
- If a structure is constrained to a signature, then only the components given in the signature are visible outside, implementation details are hidden.

```
signature ARITH =
  sig
  type t;
  val zero : t;
  val sum : t * t -> t;
  ...
  end;

structure Rational : ARITH =
  struct
  type t = int * int;
  val zero = (0,1);
  ...
  end;
```

15.1 Transparent and Opaque Signature Constraints

- Declaring structures constrained to a signature using a colon (`:`) is called a **transparent signature constraint**: components may be hidden, but they are still present.
- Alternatively, **opaque signature constraints** can be defined using (`:>`). Now all information about the new structure is hidden, except its signature.
- In the example below, for `TAs`, the implementation of `'a t` as `'a list` is still transparent, but not in `OAs`;

```
signature A =
  sig
  type 'a t;
  val bottom: 'a t;
  val next: 'a t -> 'a t;
  end;

structure As =
  struct
  type 'a t = int list;
  val bottom =  [];
  fun next (x) = 1::x;
  end;

- structure TAs : A = As;
> structure TAs :
  {type 'a t = int list,
   val 'a bottom : int list,
   val 'a next : int list -> int list}
- TAs.next(1::[2]);
> val it = [1, 1, 2] : int list
- structure OAs :> A = As;
> New type names: t/1
  structure OAs :
  {type 'a t = 'a t/1, val 'a bottom : 'a t/1,
   val 'a next : 'a t/1 -> 'a t/1}
- OAs.next(1::[2]);
! Toplevel input:
! OAs.next(1::[2]);
!          ^^^^^^
! Type clash: expression of type
!   'a list
! cannot have type
!   'b t/1
- OAs.bottom;
> val 'b it = <t> : 'b t/1
```

15.2 Abstract Datatypes

- While transparent constraints might hide too few implementation details, opaque constraints generate completely abstract types.
- A “middleway” is to declare an abstract datatype: `abstype <Datatype Binding> with <Declatation> end;`
- Datatype binding: type name followed by constructor descriptions (as in datatype declarations)
- The constructors are visible in the declaration and must be used in the declaration to implement all operations associated with the abstract type.
- The contructors are invisible outside, all identifiers declared are visible.

```
abstype 'a ast = Bottom | Cons of 'a * 'a ast
with
val bottom = Bottom;
val cons = Cons;
val el = 1;
fun  next(Cons(x,y)) = Cons(x,Cons(x,y));
fun fst(Cons(x,y)) = x;
end;
> New type names: ast/3
  type 'a ast = 'a ast/3
  val 'a bottom = <ast> : 'a ast/3
  val 'a cons = fn : 'a * 'a ast/3 -> 'a ast/3
  val el = 1 : int
  val 'a next = fn : 'a ast/3 -> 'a ast/3
  val 'a fst = fn : 'a ast/3 -> 'a
> val it = () : unit
- cons(1,bottom);
> val it = <ast> : int ast/3
- next it;
> val it = <ast> : int ast/3
- fst it;
> val it = 1 : int
```

15.3 Functors

- An ML functor is a structure that takes other structures as parameters.
- Applying it substitutes argument structures for the parameters.
- The bindings that arise from the resulting structure are returned.
- A functor can only be applied to aguments that match the correct signature.
- Functors let us write program units that can be combined in different ways.
- Functors can be used to express generic algorithms.
- Analogy:
  structure – value, signature – type, functor – function

```
functor TestA (At : A) =
struct
fun tst a = At.next a;
end;

structure TestAs = TestA (As);

> functor TestA :
  !t.
   {type 'a t = 'a t, val 'a bottom : 'a t,
    val 'a next : 'a t -> 'a t}->
       {val 'a tst : 'a t -> 'a t}
> structure TestAs : {val 'a tst : int list -> int list}
```

15.4 Example: Matrix Operations

```
(*** Matrix operations ***)
signature ZSP =
   sig
   type t
   val zero : t
   val sum  : t * t -> t
   val prod : t * t -> t
   end;

functor MatrixZSP (Z: ZSP) : ZSP =
  struct
  type t = Z.t list list;
  val zero = [];

  fun sum (rowsA,[]) = rowsA
    | sum ([],rowsB) = rowsB
    | sum (rowsA,rowsB) = ListPair.map (ListPair.map Z.sum) (rowsA,rowsB);

  fun dotprod pairs = foldl Z.sum Z.zero (ListPair.map Z.prod pairs);

  fun transp ([]::_) = []
    | transp rows = map hd rows :: transp (map tl rows);

  fun prod (rowsA,[]) = []
    | prod (rowsA,rowsB) =
        let val colsB = transp rowsB
        in  map (fn row => map (fn col => dotprod(row,col)) colsB) rowsA
        end;
  end;

structure IntZSP =
  struct
  type t = int;
  val zero = 0;
  fun sum   (x,y) = x+y: t;
  fun prod  (x,y) = x*y: t;
  end;
```

```
structure BoolZSP =
  struct
  type t = bool;
  val zero = false;
  fun sum   (x,y) = x orelse y;
  fun prod  (x,y) = x andalso y;
  end;

(** All-Pairs Shortest Paths -- Chapter 26 of Cormen et al. **)

(*Requires a maximum integer for INFINITY*)
structure PathZSP =
  struct
  type t = int;
  val SOME zero = Int.maxInt;
  val sum  = Int.min
  fun prod(m,n) = if m=zero orelse n=zero then zero
                  else m+n;
  end;

structure PathMatrix = MatrixZSP (PathZSP);

fun fast_paths mat =
  let val n = length mat
      fun f (m, mat) = if n-1 <= m then mat
                       else f(2*m, PathMatrix.prod(mat,mat))
  in  f (1, mat)  end;

val zz = PathZSP.zero;

val mat = [[0,  3,  8,  zz, ~4],
           [zz, 0,  zz, 1,   7],
           [zz, 4,  0,  zz,  zz],
           [2,  zz, ~5, 0,   zz],
           [zz, zz, zz, 6,   0]];
```

15.5 Example: Queues

```
(*** Three representations of queues ***)

(** Queues as lists **)
structure Queue1 =
  struct
  type 'a t = 'a list;
  exception E;

  val empty = [];
  fun enq(q,x) = q @ [x];
  fun null(x::q) = false
    | null _ = true;
  fun hd(x::q) = x
    | hd [] = raise E;
  fun deq(x::q) = q
    | deq [] = raise E;
  end;

(** Queues as a new datatype **)
structure Queue2 =
  struct
  datatype 'a t = empty
                | enq of 'a t * 'a;
  exception E;

  fun null (enq _) = false
    | null empty = true;
  fun hd (enq(empty,x)) = x
    | hd (enq(q,x)) = hd q
    | hd empty = raise E;
  fun deq (enq(empty,x)) = empty
    | deq (enq(q,x)) = enq(deq q, x)
    | deq empty = raise E;
  end;
```

```
(*Applicative queues represented by (heads, reversed tails).*)
structure Queue3 =
  struct
  datatype 'a t = Queue of ('a list * 'a list);
  exception E;

  val empty = Queue([],[]);
  (*Normalized queue, if nonempty, has nonempty heads list*)
  fun norm (Queue([],tails)) = Queue(rev tails, [])
    | norm q = q;
  (*norm has an effect if input queue is empty*)
  fun enq(Queue(heads,tails), x) = norm(Queue(heads, x::tails));
  fun null(Queue([],[])) = true
    | null _ = false;
  fun hd(Queue(x::_,_)) = x
    | hd(Queue([],_)) = raise E;
  (*normalize in case heads become empty*)
  fun deq(Queue(x::heads,tails)) = norm(Queue(heads,tails))
    | deq(Queue([],_)) = raise E;
  end;

signature QUEUE =
  sig
  type 'a t                      (*type of queues*)
  exception E                    (*for errors in hd, deq*)
  val empty: 'a t                (*the empty queue*)
  val enq: 'a t * 'a -> 'a t     (*add to end*)
  val null: 'a t -> bool         (*test for empty queue*)
  val hd: 'a t -> 'a             (*return front element*)
  val deq: 'a t -> 'a t          (*remove element from front*)
  end;

structure Queue : QUEUE =
  struct
  abstype 'a t = Queue of ('a list * 'a list)
    with
    exception E;
```

```
    val empty = Queue([],[]);
    (*Normalized queue, if nonempty, has nonempty heads list*)
    fun norm (Queue([],tails)) = Queue(rev tails, [])
      | norm q = q;
    (*norm has an effect if input queue is empty*)
    fun enq(Queue(heads,tails), x) = norm(Queue(heads, x::tails));
    fun null(Queue([],[])) = true
      | null _ = false;
    fun hd(Queue(x::_,_)) = x
      | hd(Queue([],_)) = raise E;
    (*normalize in case heads become empty*)
    fun deq(Queue(x::heads,tails)) = norm(Queue(heads,tails))
      | deq(Queue([],_)) = raise E;
    end
  end;

(*** Abstype declarations ***)

(** Queues as lists **)
abstype 'a queue1 = Q1 of 'a list
  with
  val empty = Q1 [];
  fun enq(Q1 q, x) = Q1 (q @ [x]);
  fun qnull(Q1(x::q)) = false
    | qnull _ = true;
  fun qhd(Q1(x::q)) = x;
  fun deq(Q1(x::q)) = Q1 q;
  end;

(** Queues as a new datatype **)
abstype 'a queue2 = Empty
                  | Enq of 'a queue2 * 'a
  with
  val empty = Empty
  and enq   = Enq
  fun qnull (Enq _) = false
```

```
    | qnull Empty = true;
  fun qhd (Enq(Empty,x)) = x
    | qhd (Enq(q,x)) = qhd q;
  fun deq (Enq(Empty,x)) = Empty
    | deq (Enq(q,x)) = Enq(deq q, x);
  end;

(**** Functors ****)
(** Testing/benchmarking of queues **)

functor TestQueue (Q: QUEUE) =
  struct
  fun fromList l = foldl (fn (x,q) => Q.enq(q,x)) Q.empty l;
  fun toList q = if Q.null q then []
                 else Q.hd q :: toList (Q.deq q);
  end;

(** Queues and Breadth-First Search **)

functor BreadthFirst (Q: QUEUE) =
  struct
  fun enqlist q xs = foldl (fn (x,q) => Q.enq(q,x)) q xs;

  fun search next x =
    let fun bfs q =
          if Q.null q then Nil
          else let val y = Q.hd q
               in  Cons(y, fn()=> bfs (enqlist (Q.deq q) (next y)))
               end
    in  bfs (Q.enq(Q.empty, x))  end;
  end;

structure Breadth = BreadthFirst (Queue);
fun brQueen n = Seq.filter (isFull n) (Breadth.search (nextQueen n) []);

structure Breadth1 = BreadthFirst (Queue1);
fun brQueen1 n = Seq.filter (isFull n) (Breadth1.search (nextQueen n) []);
```

16 Modules

- A large system should be organized in small structures.
- The organization should be hierarchical: major subsystems should be implemented as structures whose components are structures of the layer below.
- A well organized system will have many small signatures.
  Component specifications will obey strict naming conventions. In a group project, team members will have to agree upon each signature. Subsequent changes to signatures must be controlled rigorously.
- The system will include some functors. If the major subsystems are implemented independently, they will all have to be functors!

16.1 Functors with Multiple Arguments

- Multiple arguments of functions: packaged in a tuple or a record.
- Multiple arguments of functionals (higher-order functions): currying
- Multiple arguments of functors: packaged in a structure! (analogy to records as arguments of functions)

Remarks:

- Like with `val` declarations the keyword `and` can be used: `structure O1: ORDER and O2: ORDER`.
- For single argument functors both of the following kinds of syntax can be used:
  ```
  functor F (S: MYS) ...
  functor F (structure S: MYS) ...
  ```
- If datatype ‘order’ is declared by the user this results in an error because ‘order’ in ORDER is the user-declared datatype and ‘String.compare’ expects the ‘order’ type from the Standard Library.
  Obviously types, structurs and functors are equal by content but datatypes are equal by name!!! (see example next page)

```
(* from the standard lib:
   datatype order = LESS | EQUAL | GREATER;

   introducing type order = order works ok *)

signature ORDER = (* linearly ordered types *)
  sig
  type t
  val compare: t*t -> order
  end;

functor LexOrder (structure O1: ORDER
                  structure O2: ORDER) : ORDER =
  struct
  type t = O1.t * O2.t;
  fun compare ((x1,y1), (x2,y2)) =
    (case O1.compare (x1,x2) of
    EQUAL => O2.compare (y1,y2)
  | ord => ord)
  end;

structure StringOrder: ORDER =
  struct
  type t = string;
  val compare = String.compare
  end;

structure IntegerOrder: ORDER =
  struct
  type t = int;
  val compare = Int.compare
  end;

structure StringIntOrd = LexOrder(structure O1=StringOrder
                                  structure O2=IntegerOrder);
```

16.2 Further Concepts for Modules

16.2.1 Functors with No Arguments

- Functors without arguments have the empty signature or structure as an argument.
  Such functors are structures.
- `functor F () = ...`

16.2.2 Sharing Constraints

- When modules are combined to form a large one, special care is needed to ensure that the components fit together.
- Checking for type compatibility is facilitated by sharing constraints which compel types to agree!

```
functor Join2 (structure PQueue: PRIORITY_QUEUE
               structure Dict: DICTIONARY
               sharing type PQueue.Item.t = Dict.key) =
  struct
  fun lookmin(dict, pq) = Dict.lookup(dict, PQueue.min pq);
  end;
```

Example: Sharing Constraints on Structures

```
signature IN =
  sig
  structure PQueue: PRIORITY_QUEUE
  type problem
  val goals: problem -> PQueue.t
  end;

signature OUT =
  sig
  structure PQueue: PRIORITY_QUEUE
  type solution
  val solve: PQueue.t -> solution
  end;

functor MainFunctor (structure In: IN and Out: OUT
                     sharing In.PQueue = Out.PQueue) =
  struct
  fun tackle(p) = Out.solve(In.goals p)
  end;
```

16.2.3 Fully Functional Programming

- Typical pragmatic rule for program design: never declare a procedure that is called only once
- But: declaring procedures is regarded as good style.
  In ML: good reasons for declaring more functors than are strictly necessary.
- If all program units are coded as functors then they can be written and compiled seperately (good for program development), several programmers can code their functors independently, functors may be coded in any order (each functor refers to signatures but not to structures and other functors).
- Functors are linked together at the end: declaration of structures by applying a functor to previously declared structures.
- **Self-contained signatures**: predefined names (such as `int` are called pervasive (visible everywhere); a name occuring in a signature which has not yet been specified is said to be **free in that signature**.

16.2.4 The ‘open’ Declaration

- To omit using compound names when structures are nested, structures can be opend locally.
- Alternatively, structures can be introduced locally using `let`.
- `Open` can be used selectively.

```
functor FlexArray (Braun: BRAUN): FLEXARRAY =
  struct
  local open Braun Braun.Tree
    in
    ...
    end
  end;

functor QuadOrder (O: ORDER) : ORDER =
  let structure OO = LexOrder (structure O1 = O and O2 = O)
  in LexOrder(structure O1 = OO structure O2 = OO)
  end;

structure Tree =
  struct
     structure Export =
     struct
     ...
     end;
  open Export;
  ...
  end;
```

16.2.5 Sharing Constraints in a Signature

```
signature BRAUNPAIR2 =
  sig
    structure Braun: BRAUN
    type 'a tree
    sharing type tree = Braun.Tree.tree
    val zip: 'a tree * 'b tree -> ('a * 'b) tree
    ...
    end;
```

16.2.6 'Include' Specification

```
signature BRAUNPAIR4 =
  sig
    include BRAUN
    val zip: 'a Tree.tree * 'b Tree.tree -> ('a * 'b) Tree.tree
  end;
```

Writing the contents of the signature without the surrounding `sig .. end;`

16.3 A Complex Example: Dictionaries

```
(*** Dictionaries -- as a functor ***)

(** Linearly ordered types **)

signature ORDER =
  sig
  type t
  val compare: t*t -> order
  end;

structure StringOrder: ORDER =
  struct
  type t = string;
  val compare = String.compare
  end;



functor Dictionary (Key: ORDER) : DICTIONARY =
  struct

  type key = Key.t;

  abstype 'a t = Leaf
               | Bran of key * 'a * 'a t * 'a t
    with

    exception E of key;

    val empty = Leaf;

    fun lookup (Bran(a,x,t1,t2), b) =
          (case Key.compare(a,b) of
               GREATER => lookup(t1, b)
             | EQUAL   => x
             | LESS    => lookup(t2, b))
```

```
      | lookup (Leaf, b) = raise E b;

    fun insert (Leaf, b, y) = Bran(b, y, Leaf, Leaf)
      | insert (Bran(a,x,t1,t2), b, y) =
          (case Key.compare(a,b) of
               GREATER => Bran(a, x, insert(t1,b,y), t2)
             | EQUAL   => raise E b
             | LESS    => Bran(a, x, t1, insert(t2,b,y)));

    fun update (Leaf, b, y) = Bran(b, y, Leaf, Leaf)
      | update (Bran(a,x,t1,t2), b, y) =
          (case Key.compare(a,b) of
               GREATER => Bran(a, x, update(t1,b,y), t2)
             | EQUAL   => Bran(a, y, t1, t2)
             | LESS    => Bran(a, x, t1, update(t2,b,y)));
    end
  end;


(*This instance is required by sample9.sml and sample10.sml*)
structure StringDict = Dictionary (StringOrder);


(*Differs from the other PRIORITY_QUEUE by having a substructure*)
signature PRIORITY_QUEUE =
  sig
  structure Item : ORDER
  type t
  val empty    : t
  val null     : t -> bool
  val insert   : Item.t * t -> t
  val min      : t -> Item.t
  val delmin   : t -> t
  val fromList : Item.t list -> t
  val toList   : t -> Item.t list
  val sort     : Item.t list -> Item.t list
  end;
```

```
(**** Building large systems using modules ****)

(** Association list implementation of Dictionaries
     Illustrates eqtype and functor syntax **)

functor AssocList (eqtype key) : DICTIONARY =
  struct
  type key = key;
  type 'a t = (key * 'a) list;
  exception E of key;

  val empty = [];

  fun lookup ((a,x)::pairs, b) =
        if a=b then  x  else  lookup(pairs, b)
    | lookup ([], b) = raise E b;

  fun insert ((a,x)::pairs, b, y) =
        if a=b then  raise E b  else  (a,x)::insert(pairs, b, y)
    | insert ([], b, y) = [(b,y)];

  fun update (pairs, b, y) = (b,y)::pairs;

  end;


signature TREE =
  sig
  datatype 'a tree = Lf  |  Br of 'a * 'a tree * 'a tree
  val size: 'a tree -> int
  val depth: 'a tree -> int
  val reflect: 'a tree -> 'a tree
  end;
```

```
signature BRAUN =
  sig
  structure Tree: TREE
  val sub:    'a Tree.tree * int -> 'a
  val update: 'a Tree.tree * int * 'a -> 'a Tree.tree
  val delete: 'a Tree.tree * int -> 'a Tree.tree
  val loext:  'a Tree.tree * 'a -> 'a Tree.tree
  val lorem:  'a Tree.tree -> 'a Tree.tree
  end;

functor PriorityQueue(structure Item: ORDER
                      and       Tree: TREE) : PRIORITY_QUEUE =
  let open Tree
  in
    struct
    structure Item = Item;
    fun x <= y = (Item.compare(x,y) <> GREATER);

    abstype t = PQ of Item.t tree
    with

      val empty = PQ Lf;

      fun null (PQ Lf) = true
        | null  _      = false;

      fun min (PQ(Br(v,_,_))) = v;

      fun insert'(w, Lf) = Br(w, Lf, Lf)
        | insert'(w, Br(v, t1, t2)) =
            if w <= v then Br(w, insert'(v, t2), t1)
                      else Br(v, insert'(w, t2), t1);

      fun insert (w, PQ t) = PQ (insert' (w, t));

      fun leftrem (Br(v,Lf,_)) = (v, Lf)
        | leftrem (Br(v,t1,t2)) =
```

```
            let val (w, t) = leftrem t1
            in  (w, Br(v,t2,t))  end;

      fun siftdown (w, Lf, _) = Br(w,Lf,Lf)
        | siftdown (w, t as Br(v,_,_), Lf) =
            if w <= v then Br(w, t, Lf)
                      else Br(v, Br(w,Lf,Lf), Lf)
        | siftdown (w, t1 as Br(v1,p1,q1), t2 as Br(v2,p2,q2)) =
            if w <= v1 andalso w <= v2 then Br(w,t1,t2)
            else if v1 <= v2 then Br(v1, siftdown(w,p1,q1), t2)
               (* v2 < v1 *) else Br(v2, t1, siftdown(w,p2,q2));

      fun delmin (PQ Lf) = raise Size
        | delmin (PQ (Br(v,Lf,_))) = PQ Lf
        | delmin (PQ (Br(v,t1,t2))) =
            let val (w, t) = leftrem t1
            in  PQ (siftdown (w,t2,t))  end;

      fun heapify (0, vs) = (Lf, vs)
        | heapify (n, v::vs) =
            let val (t1, vs1) = heapify (n div 2, vs)
                val (t2, vs2) = heapify ((n-1) div 2, vs1)
            in  (siftdown (v,t1,t2), vs2)  end;

      fun fromList vs = PQ (#1 (heapify (length vs, vs)));

      fun toList (d as PQ (Br(v,_,_))) = v :: toList(delmin d)
        | toList _ = [];

      fun sort vs = toList (fromList vs);

      end
    end
  end;


functor FlexArray (Braun: BRAUN) : FLEXARRAY =
```

```
  struct
  datatype 'a array = Array of 'a Braun.Tree.tree * int;

  val empty = Array(Braun.Tree.Lf,0);

  fun length (Array(_,n)) = n;

  fun sub (Array(t,n), k) =
      if 0<=k andalso k<n then Braun.sub(t,k+1)
      else raise Subscript;

  fun update (Array(t,n), k, w) =
      if 0<=k andalso k<n then Array(Braun.update(t,k+1,w), n)
      else raise Subscript;

  fun loext (Array(t,n), w) = Array(Braun.loext(t,w), n+1);

  fun lorem (Array(t,n)) =
      if n>0 then Array(Braun.lorem t, n-1)
      else raise Size;

  fun hiext (Array(t,n), w) = Array(Braun.update(t,n+1,w), n+1);

  fun hirem (Array(t,n)) =
      if n>0 then Array(Braun.delete(t,n) , n-1)
      else raise Size;

  end;


functor BraunFunctor (Tree: TREE) : BRAUN =
  let open Tree in
    struct
    structure Tree = Tree;

    fun sub (Lf, _) = raise Subscript
      | sub (Br(v,t1,t2), k) =
```

```
          if k = 1 then v
          else if k mod 2 = 0
               then sub (t1, k div 2)
               else sub (t2, k div 2);

    fun update (Lf, k, w) =
          if k = 1 then Br (w, Lf, Lf)
          else raise Subscript
      | update (Br(v,t1,t2), k, w) =
          if k = 1 then Br (w, t1, t2)
          else if k mod 2 = 0
               then Br (v,  update(t1, k div 2, w),  t2)
               else Br (v,  t1,  update(t2, k div 2, w));

    fun delete (Lf, n) = raise Subscript
      | delete (Br(v,t1,t2), n) =
          if n = 1 then Lf
          else if n mod 2 = 0
               then Br (v,  delete(t1, n div 2),  t2)
               else Br (v,  t1,  delete(t2, n div 2));

    fun loext (Lf, w) = Br(w, Lf, Lf)
      | loext (Br(v,t1,t2), w) = Br(w, loext(t2,v), t1);

    fun lorem Lf = raise Size
      | lorem (Br(_,Lf,_)) = Lf         (*No evens, therefore no odds eith
      | lorem (Br(_, t1 as Br(v,_,_), t2)) = Br(v, t2, lorem t1);

    end
  end;


functor TreeFunctor () : TREE =
  struct
  datatype 'a tree = Lf  |  Br of 'a * 'a tree * 'a tree;

  fun size Lf = 0
```

```
  | size (Br(v,t1,t2)) =  1 + size t1 + size t2;

fun depth Lf = 0
  | depth (Br(v,t1,t2)) =  1 + Int.max (depth t1, depth t2);

fun reflect Lf = Lf
  | reflect (Br(v,t1,t2)) =
        Br(v, reflect t2, reflect t1);

fun preord (Lf, vs) = vs
  | preord (Br(v,t1,t2), vs) =
        v :: preord (t1, preord (t2, vs));

fun inord (Lf, vs) = vs
  | inord (Br(v,t1,t2), vs) =
        inord (t1, v::inord (t2, vs));

fun postord (Lf, vs) = vs
  | postord (Br(v,t1,t2), vs) =
        postord (t1, postord (t2, v::vs));

fun balpre  []    = Lf
  | balpre(x::xs) =
      let val k = length xs div 2
      in  Br(x, balpre(List.take(xs,k)),  balpre(List.drop(xs,k)))
      end;

fun balin [] = Lf
  | balin xs =
      let val k = length xs div 2
          val y::ys = List.drop(xs,k)
      in  Br(y, balin (List.take(xs,k)), balin ys)
      end;

fun balpost xs = reflect (balpre (rev xs));
end;
```

```
(** Application of the functors **)

structure Tree = TreeFunctor();
structure Braun = BraunFunctor (Tree);
structure Flex = FlexArray (Braun);
structure StringPQueue =
   PriorityQueue(structure Item = StringOrder
                 and       Tree = Tree);
```

17 Imperative Programming

- Remember: imperative programming is based on commands which change states of the machine store.
- For some aspects of programming, an imperative style is the most natural way (input/output) and/or the most efficient way (some data structures, such as hash tables and ring buffers).
- Imperative features: (variables, functions), references (pointers), arrays, loop constructs, commands for input/output.
- ML includes imperative features and supports imperative programming in full generality.
- Programs which are mostly concerned with managing states are best coded imperative.

17.1 Control Structures

- ML does not distinguish commands from expressions.
- A command is an expression that updates a state when evaluated. Remember: Assignment is a command/statement, the right-hand side of an assignment is an expression.
- Most commands have type *unit* and return *unit*.
- Expressions can be viewed as commands. E. g., the conditional expression and the case-expression can serve as control structure.
  `if E then E1 else E2`
  `case E of P1 => E1 | ... | Pn => En`
  First E is evaluated, perhaps changing the state. One of E_i is selected and evaluated. The resulting value (which can be *unit*) is returned.
- A series of commands can be executed by the expression
  `(E1; E2; ...; En)`
  The result is the value of E_n, the other results are discarded.
  (Because of the use of semicolon, this construct must always be enclosed in parentheses, unless it forms the body of a `let`-expression.)

17.1.1 Assignment

- The assignment `E1 := E2` evaluates E_1, which must return a reference p, and evaluates E_2. The value of E_2 is stored at address p.
- Syntactically, `:=` is a function and `E1 := E2` is an expression, even though it updates the store.
- Like most functions that change the machine's state, it returns `()` of type *unit*.

17.1.2 While-Command

- For iteration ML has a `while`-command:
  `while E1 do E2`
  If E_1 evaluates to *false*, then the `while` is finished; if E_1 evaluates to *true* then E_2 is evaluated and the `while` is executed again.
  The `while` command satisfies the following recursion:

  ```
  while E1 do E2 ==
  if E1 then (E2; while E1 do E2) else ()
  ```

  The return-value is `()`, E_2 is evaluated just for its effect on the state.

Examples for the use of assignment and while-commands is given during the introduction of reference types.

17.2 Reference Types

- All values computed during the execution of an ML program reside for some time in the machine store.
- To functional programmers, the store is nothing but a device inside the computer. They never have to think about the store until they run out of it.
- With imperative programming, the store is visible.
- An ML reference denotes the address of a location in the store.
  Each location contains a value, which can be replaced using an assignement.
- A reference is itself a value: if x has type τ, then a reference to x is written `ref x` and has type τ `ref`.
- The constructor `ref` creates references.
  When applied to a value v, it allocates a new address with v for its initial content and returns a reference to this address.
- Although `ref` is an ML function, it is not a function in the mathematical sense: It returns a new address every time it is called!
- The function `!`, when applied to a reference, returns its contents. This operation is called **dereferencing**.
  Clearly, `!` is not a mathememattical function, its result depends on the store.

```
val p = ref 5 and q = ref 2;
> val p = ref 5 : int ref
> val q = ref 2 : int ref
(!p, !q);
> (5, 2) : int * int
p := !p + !q;
> () : unit
(!p, !q);
> (7, 2) : int * int
```

The assignment does not change the value of reference p (an address) but its contents!

17.2.1 References in Data Structures

- Because references are ML values, they may belong to tuples, lists, etc.
- ML compilers print the value of a reference as *ref c*, where c is its contents, rather than printing the address as a number.
- References to references are allowed.

```
val refs = [p, q, p];
> val refs = [ref 7, ref 2, ref 7] : int ref list
q := 1346;
> () : unit
refs;
> [ref 7, ref 1346, ref 7] : int ref list
hd refs := 1415;
> () : unit
refs;
> [ ref 1415, ref 1346, ref 1415] : int ref list
(!p, !q);
> (1415, 1346) : int * int
val refp = ref p and refq = ref q;
> val refp = ref (ref 1415) : int ref ref
> val refq = ref (ref 1346) : int ref ref
!refq := !(! refp);
> () : unit
(!p, !q);
> (1415, 1415) : int * int
```

17.2.2 Equality of References

- The ML equality test is valid for all reference types.
- Two references of the same type are equal precisely if they denote the same address.

```
p = q;
> false : bool
hd refs = p;
> true : bool
hd refs = q;
> false : bool
```

- In Pascal (and C), two pointer variables are equal if they contain the same address. An assignement makes two pointers equal.
- In imperative languages, where all variables can be updated, a pointer variable really involves two levels of references. The usual notion of pointer equality is like comparing references to references (as *refp* and *refq* above).

```
!refp = !refq;
> false : bool
refq := p;
> () : unit
!refp = !refq;
> true : bool
```

- **Aliasing**: When two references are equal, assigning to one affects the contents of the other. This can cause confusion!
- In procedural languages, aliasing can lead to problems: in a procedure call, a global variable and a formal parameter (using call by reference) may denote the same address.

17.2.3 Cyclic Data Structures

- Circular chains of references arise in many situations.
- Updating references to create a cycle is sometimes called ‘tying the knot’.
- Many function language interpreters implement recursive functions by creating a cycle in the execution environment.
- Example: *cFact* refers to itself via *cp*

```
val cp = ref (fn k => k+1);
> val cp = ref fn : (int -> int) ref
fun cFact n = if n=0 then 1 else n * !cp(n-1);
> val cFact = fn : int -> int
cFact 8;
> 64 : int
cp := cFact;
> () : unit
cFact 8;
> 40320 : int
```

17.2.4 Imperative Calculation of Factorial

```
fun impFact n =
  let val resultp = ref 1
      and ip = ref 0
  in  while !ip < n do (ip := !ip + 1;
                        resultp := !resultp * !ip);
      !resultp
  end;

fun pFact (n, resultp) =
  let val ip = ref 0
  in  resultp := 1;
      while !ip < n do (ip := !ip + 1;
                        resultp := !resultp * !ip)
  end;
```

- The second version is a ‘procedure’ with two formal parameters, the second parameter is a reference parameter!

17.2.5 Library Functions

- `ignore`: Ignores its argument and returns `()`.

```
if !skip then ignore (TextIO.inputLine file)
         else skip := true
```

  Return type is unit. As side-effect, one line of the input file is skipped.

- `before`: returns its first argument; e. g. switching values

```
y := (!x before x := !y);
```

  as alternative to `y := #1 (!x, x := !y)`.

- `app`: list functional which applies a command to every element of a list.

```
fun initialize rs x = app (fn r => r := x) rs;
> val initialize = fn : 'a ref list -> 'a -> unit
initialize refs 1815;
> () : unit
refs;
> [ref 1815, ref 1815, ref 1815] : int ref list
```

  `app f l` is similar to `ignore (map f l)` but avoids building a list of results.

17.2.6 Polymorphic References

- In some (old) programming languages, no type information was kept about the contents of a reference. Thus, e. g., a character code could be interpreted as a real number.
- With types in ML (as with pointer in Pascal and other languages) type-safety it is ensured.
- But: What does the type τ *ref* mean if τ is polymorphic?

An illegal session:

```
fun I x = x;
> val I = fn : 'a -> 'a
val fp = ref I;
> val fp = ref fn : ('a -> 'a) ref
(!fp true, !fp 5);
> (true, 5) : bool * int
fp := not;
!fp 5; <<<< run time error !!!
```

- With `not` a function of type `bool -> bool` is assigned to `fp`.
- ML is supposed to dected all type errors at compile time. But – in this imaginary session – it cannot detect that `!fp` applied to 5 results in a type error.
- Don't worry, type checking in ML is safe! Expressions in polymorphic *val* declarations must be 'syntactic values' (explained below).

An example with real polymorphism:

```
fun irev l =
  let val resultp = ref []
      and lp = ref l
  in  while not (null (!lp)) do
          (resultp := hd(!lp) :: !resultp;
           lp := tl(!lp));
      !resultp
  end;
```

- To understand the problem of polymorphic references, let's first look at polymorphism and subsitution.
- `val Id` $=$ `E` makes Id a synonym for E

```
let val I = fn x => x in (I, true, I 5) end;
> (true, 5) : bool * int
(( fn x => x) true, (fn x => x) 5);
> (true, 5) : bool * int
let val nill = [[]] in (["Exeter"]::nill, [1415]::nill) end;
> ([["Exeter"], []], [[1415], []])
> : string list list * int list list
([["Exeter"], []], [[1415]::[]])
> ([["Exeter"], []], [[1415], []])
> : string list list * int list list
```

- Substituting declarations by expressions does not affect the value nor the typing.
- Let us consider the erroneous assignment of a polymorphic reference above in this context.

```
let val fp = ref I
in ((!fp true, !fp 5), fp := not, !fp 5) end;
> Error: Type conflict: expected int, found bool
```

- If we substitute the declarations we now (because of the use of references) have a different expression:

```
((!(ref I) true, !(ref I) 5), (ref I) := not, !(ref I) 5);
> ((true, 5), (), 5) : (bool * int) * unit * int
```

- The problem is *sharing* (the same store address) which is not respected by substitution. Therefore, the *creation* of polymorphic references (not assignements to them) must be regulated!

Polymorphic value declarations

- If E is a **syntactic value** then the polymorphic declaration `val Id = E` is equivalent to a substitution.
- In ML, polymorphic declarations are only allowed for syntactic values.
- Syntactic values: constants; identifiers; tupels, records, and constructors (except *ref*) of syntactic values, functions in *fn* notation (even if the body contains *ref*).

Illegal expressions:

```
let val fp = ref I in fp := not; !fp 5 end;
let val fp = ref I in (!fp true, !fp 5) end;
val fp = ref I;
```

Legal expressions: with monomorphic type constraints

```
let val fp = ref I in fp := not; !fp true end;
> false : bool
val fp = ref (I: bool -> bool);
> val fp = ref fn : (bool -> bool) ref
```

Remark: Polymorphic references are an interesting topic of current research.

17.3 References in Data Structures

- Recursive datat structures are typically realized using explicit links (references).
- In ML we presented recursive data types such as lists and trees without using explicit links.
- Now we will give an example of implementing a linked data structure in ML: a doubly-linked circular list (ring-buffer).

```
signature RINGBUF =
  sig
  eqtype 'a t
  exception Empty
  val empty: unit -> 'a t
  val null: 'a t -> bool
  val label: 'a t -> 'a
  val moveLeft: 'a t -> unit
  val moveRight: 'a t -> unit
  val insert: 'a t * 'a -> unit
  val delete: 'a t -> 'a
  end;

(*Note use of :> for abstraction*)
structure RingBuf :> RINGBUF =
  struct
  datatype 'a buf = Nil
                  | Node of 'a buf ref * 'a * 'a buf ref;
  datatype 'a t = Ptr of 'a buf ref;

  exception Empty;

  fun left (Node(lp,_,_)) = lp
    | left Nil = raise Empty;

  fun right (Node(_,_,rp)) = rp
    | right Nil = raise Empty;
```

```
(*Must be a function, as each ring buffer needs a separate reference.
   Also the type checker would reject the polymorphic reference.*)
fun empty() = Ptr(ref Nil);
fun null (Ptr p) = case !p of Nil => true
                            | Node(_,x,_) => false;

fun label (Ptr p) = case !p of Nil => raise Empty
                             | Node(_,x,_) => x;

fun moveLeft (Ptr p) = (p := !(left(!p)));
fun moveRight (Ptr p) = (p := !(right(!p)));

(*Insert to left of the window, which is unchanged unless empty. *)
fun insert (Ptr p, x) =
    case !p of
        Nil =>
            let val lp = ref Nil
                and rp = ref Nil
                val new = Node(lp,x,rp)
            in  lp := new;  rp := new;  p := new  end
      | Node(lp,_,_) =>
            let val new = Node(ref(!lp), x, ref(!p))
            in  right(!lp) := new;  lp := new  end;

(*Delete the current node, raising Empty if there is none.
  Observe the comparison of left refs to see whether they
  are identical. *)
fun delete (Ptr p) =
    case !p of
        Nil => raise Empty
      | Node(lp,x,rp) =>
           (if left(!lp) = lp then p := Nil
            else (right(!lp) := !rp;  left (!rp) := !lp;
                  p := !rp);
            x)
end;
```

17.4 Input and Output

- ML provides functions for processing strings and substrings: Conversions to/from strings; splitting strings; scanning from character sources.

```
(*** String conversions ***)

val months = ["JAN", "FEB", "MAR", "APR", "MAY", "JUN",
              "JUL", "AUG", "SEP", "OCT", "NOV", "DEC"];

(*An example of using exception Bind -- forward reference from Chapter 4?*)
fun dateFromString s =
  let val sday::smon::syear::_ = String.tokens (fn c => c = #"-") s
      val SOME day = Int.fromString sday
      val mon      = String.substring (smon, 0, 3)
      val SOME year = Int.fromString syear
  in  if List.exists (fn m => m=mon) months
      then SOME (day, mon, year)
      else NONE
  end
  handle Subscript => NONE
       | Bind      => NONE;
```

- Text input/output is handled over streams.

```
(*** Stream Input/Output ***)

(** Initial letters of words in each line **)

fun firstChar s = String.sub(s,0);

val initials = implode o (map firstChar) o (String.tokens Char.isSpace);

initials "My ransom is this frail and worthless trunk";

fun batchInitials (is, os) =
  while not (TextIO.endOfStream is)
  do TextIO.output(os, initials (TextIO.inputLine is) ^ "\n");

fun promptInitials (is, os) =
  while (TextIO.output(os, "Input line? ");  TextIO.flushOut os;
         not (TextIO.endOfStream is))
  do TextIO.output(os, "Initials:   " ^ initials(TextIO.inputLine is) ^ "\n");


(*** Conversion to HTML ***)

fun firstLine s =
    let val (name,rest) =
                Substring.splitl (fn c => c <> #".") (Substring.all s)
    in  "\n<P><EM>" ^ Substring.string name ^
        "</EM>"     ^ Substring.string rest
    end;

fun htmlCvt fileName =
    let val is = TextIO.openIn fileName
        and os = TextIO.openOut (fileName ^ ".html")
        fun cvt _ "" = ()
          | cvt _ "\n" = cvt true (TextIO.inputLine is)
          | cvt first s =
                (TextIO.output (os,
                                if first then firstLine s
                                else "<BR>" ^ s);
                 cvt false (TextIO.inputLine is));
    in  cvt true "\n";  TextIO.closeIn is;  TextIO.closeOut os  end;
```

```
(**** Pretty printing ****)

signature PRETTY =
  sig
   type t
   val blo : int * t list -> t
   val str : string -> t
   val brk : int -> t
   val pr  : TextIO.outstream * t * int -> unit
   end;

structure Pretty : PRETTY =
  struct
  (*Printing items: compound phrases, strings, and breaks*)
  datatype t =
      Block of t list * int * int       (*indentation, length*)
    | String of string
    | Break of int;                     (*length*)

  (*Add the lengths of the expressions until the next Break; if no Break then
    include "after", to account for text following this block. *)
  fun breakdist (Block(_,_,len)::es, after) = len + breakdist(es, after)
    | breakdist (String s :: es, after) = size s + breakdist (es, after)
    | breakdist (Break _ :: es, after) = 0
    | breakdist ([], after) = after;

  fun pr (os, e, margin) =
   let val space = ref margin

       fun blanks n = (TextIO.output(os, StringCvt.padLeft #" " n "");
                       space := !space - n)

       fun newline () = (TextIO.output(os,"\n");  space := margin)

       fun printing ([], _, _) = ()
         | printing (e::es, blockspace, after) =
          (case e of
               Block(bes,indent,len) =>
                  printing(bes, !space-indent, breakdist(es,after))
             | String s => (TextIO.output(os,s);   space := !space - size s)
             | Break len =>
                 if len + breakdist(es,after) <= !space
```

```
                then blanks len
                else (newline();  blanks(margin-blockspace));
           printing (es, blockspace, after))
   in  printing([e], margin, 0);  newline()  end;

  fun length (Block(_,_,len)) = len
    | length (String s) = size s
    | length (Break len) = len;

  val str = String  and  brk = Break;

  fun blo (indent,es) =
    let fun sum([], k) = k
          | sum(e::es, k) = sum(es, length e + k)
    in  Block(es,indent, sum(es,0))  end;
  end;



local open Pretty
  in

  fun prettyshow (Atom a) = str a
    | prettyshow (Neg p) =
        blo(1, [str"(~", prettyshow p, str")"])
    | prettyshow (Conj(p,q)) =
        blo(1, [str"(", prettyshow p, str" &",
                brk 1, prettyshow q, str")"])
    | prettyshow (Disj(p,q)) =
        blo(1, [str"(", prettyshow p, str" |",
                brk 1, prettyshow q, str")"]);

  end;
```

18 Advanced Topics and Special Aspects

- Lambda Calculus.
- Automatic Theorem Proving (Isabelle).
- Verification of functional programs.
- Writing parser-generators/interpreters with ML.
- Comparing object-oriented and functional programming.
- Lisp vs. ML, Haskell vs. ML.
- Erlang: Functional programming for network management.
- Cognitive and educational aspects of functional vs. imperative programming.