

Introduction to Functional Programming

Prof Saroj Kaushik, CSE Department, IITD

Functional Languages

Focus on data values described by expressions built from function applications. Functional languages emphasize the evaluation of expressions, rather than execution of commands. Using functions to combine the basic values forms the expressions in these languages. A functional program can be considered as a mapping of inputs to outputs.

A mathematical function also maps domain elements (inputs) to range elements (outputs). For example, define a function $f : Domain \rightarrow Range$, where $Domain = \{a, b, c\}$ and $Range = \{p, q, r\}$. Let $f(a) = r$; $f(b) = p$; $f(c) = q$. The domain and range might be infinite. So one can define general rule of mapping. Consider a function *square* from Integer to Integer as follows:

square: $Integer \rightarrow Integer$

$square(x) = x * x; \quad x \in Integer.$

Here a variable x is called a parameter of the definition. It stands for any member of domain set. At the time application, a particular member of the domain is specified such as $Square(3) = 9$, $Square(10) = 100$.

Combining other functions one can create new functions. The most common form of combining functions in mathematics is composition i.e., $f \equiv g \circ h$. Applying f on arguments is defined to be equivalent to applying h on arguments and then applying g to the result i.e., $f(x) \equiv g(h(x))$. Functional programming is characterized by the programming with values, functions and functional forms. The compositional operator is an example of a *functional form*. Functional programming is based on the mathematical concept of a function. It includes a set of primitive functions and a set of functional forms. Pure functional languages perform all their computations via function application. The variables in functional languages are values rather than memory locations. So a variable in functional language means the same thing no matter where it appears in the body of a function. This makes it easier to understand the program. A good point of functional programs is that they are small and concise. Functional program encourages thinking at higher levels of abstraction by providing higher-order functions.

A higher-order function has inputs or outputs that could also be function abstractions. It also has an ability to define functions in the form of equations by using of pattern matching. The list manipulations are simple because of its simple syntax. True functional languages treat functions as first-class values. Many theorem provers and other systems for mathematics by computer make use of functional languages.

A **functional program** consists of an expression E (representing both the algorithm and the input). This expression E is subject to some reduction rules. **Reduction** consists of replacing some part P of E by another expression P' according to the given reduction rules. This process of reduction will be repeated until the resulting expression has no more parts that can be reduced. The expression E'' thus obtained is called the **normal form** of E and constitutes the output of the functional program. LISP, Scheme, ML, Miranda and Haskell are just some of the languages to implement this elegant computational paradigm. The basic concepts of functional programming originated from

lambda calculus. It is widely agreed that languages such as Haskell and Miranda are *purely functional*, while SML and Scheme are not. However, there are some small differences of opinion about the precise technical motivation for this distinction. Scheme and Standard ML are predominantly functional but also allow 'side effects' (computational effects caused by expression evaluation that persist after the evaluation is completed). Sometimes, the term "purely functional" is also used in a broader sense to mean languages that might incorporate computational effects, but without altering the notion of 'function' (as evidenced by the fact that the essential properties of functions are preserved.). Typically, the evaluation of an expression can yield a 'task', which is then executed separately to cause computational effects. The evaluation and execution phases are separated in such a way that the evaluation phase does not compromise the standard properties of expressions and functions. Since all functional languages rely on implicit memory management, garbage collection is an important component in any implementation. Functional languages manage storage automatically. The user does not decide when to deallocate storage. The system scans the memory at intervals marking every thing that is accessible and reclaiming remaining. This operation is called garbage collection. Well-structured software in functional programming is easy to write, easy to debug, and provides a collection of modules that can be re-used to reduce future programming costs. Conventional languages place conceptual limits on the way problems can be modularized. Functional languages push those limits back. Use of higher-order functions and lazy evaluation technique can contribute greatly to modularity. Since modularity is the key to successful programming, functional languages are very important to the real world. Functional languages facilitate the expression of concepts and structures at a high level of abstraction. A key property of functional languages is the referential transparency. The phrase 'referentially transparency' is used to describe notations where only the value of immediate component expressions is significant in determining the value of a compound expression. Equal sub expressions can be interchanged in the context of a larger expression to give equal results. Hence the value of an expression depends only on the values of its constituent expressions (if any) and these sub expressions may be replaced freely by others possessing the same value. [Reade, Bird]. The meaning of an expression is its value and there are no other effects, hidden or otherwise, in any procedure for actually obtaining it. Functional programming languages are important for real-world applications in Artificial intelligence. Functional programming is useful for developing **executable specifications** and **prototype implementations**.

Mathematical function verses Imperative Programming Function:

The most important difference is based on the notion of a modifiable variable. Mathematical function parameters simply represent some value that is fixed at function application time. A function written in a truly functional language is a mathematical function, which evaluates an expression and returns its value. A function written in an imperative language, such as Fortran, Pascal, or C, which evaluates the same expression, may also change a value in a memory location having nothing to do with the expression being evaluated. While it may appear harmless, this side effect prevents the substitution of the expression's value for the invocation of the function that evaluates it. Another difference is the way functions are defined. Programming functions are defined

procedurally whereas mathematical functions are defined in terms of expression or application of other functions.

Logic and functional programming

The interpretation of logic program $cube(X,Y) \leftarrow Y = X * X * X$ is that $\forall X, \forall Y, (Y \text{ is a cube of } X \text{ if } Y = X * X * X)$. It asserts a condition that must hold between the corresponding domain and range elements of the function. Here *cube* is a predicate name. We can see that $cube(2, 8)$ is true and $cube(2,5)$ is false.

In functional definition $cube(X) = X * X * X$ introduces a functional object to which functional operations such as functional composition may be applied. Here *cube* is a function which when applied to domain value, gives a value in the range such as $cube(2)$ gives 8 as a result.

Problems with Functional languages:

In functional languages, complexity (in terms of space and time) analysis is an area that has been more or less neglected until recently. One of the reasons for this is that it is rather different from ordinary complexity analysis. Some work has been done by Bror Bjerner's and Sören Holmström. Functional languages (particularly lazy ones - those whose arguments are evaluated when needed) are difficult to implement efficiently.

Because of the constraints such as memory management, garbage collection etc. as compared to imperative languages. Input/output and non-determinism are weak areas for functional languages. Arrays are a important data structure for some algorithms. Unfortunately the way they are traditionally used, i.e., with updating, does not fit well into a pure functional world. A different approach to handling arrays where updates can be implemented as in-place-updating.

Lambda Calculus

Lambda Calculus (λ -calculus) is a functional notation introduced by Alonzo Church and Stephen Kleene in the early 1930s to formalize the notion of computability. Church introduced pure lambda calculus to study the computation with functions. The lambda calculus is a formalization of the notion and a theory of functions. It is mainly concerned with functional applications and the evaluation of λ -expressions by techniques of substitution. The lambda-calculus is a simple language with few constructs and a simple semantics. But, it is expressive; it is sufficiently powerful to express all computable functions. It is an abstract model of computation equivalent to the Turing machine, recursive functions, and Markov chains. Unlike the Turing machine which is sequential in nature, they retain an implicit parallelism that is present in mathematical expressions. The pure lambda-calculus does not have any built-in functions or constants but these are included in applied lambda calculus. Different languages are obtained for different choices of functions and constants. Functional programming languages are based on applied λ -calculus. The lambda-calculus is rather minimal in form but as powerful as any other programming language for describing computations. Calculation in the lambda-calculus is by rewriting (reducing) a lambda-expression to a normal form. For the pure lambda-calculus, lambda-expressions are reduced by substitution. That is, occurrences of the parameter in the body are replaced with (copies of) the argument. In our extended lambda-calculus we also apply the usual reduction rules.

Historically, it precedes the development of all programming languages. It provides a very concise notation for functions, especially for list processing and has been used as the basis of LISP, SCHEME, POP-2, SML etc. LISP (LISt Processing) was designed by John McCarthy in 1958. It was developed keeping the interest of symbolic computation that is primarily used in the areas such as mechanizing theorem proving, modeling human intelligence, and natural language processing etc. In each of these areas, list processing was seen as a fundamental requirement. LISP was developed as a system for list processing based on recursive functions. Scheme is a dialect of LISP. It is a relatively small language with static rather than dynamic scope rules. SML is acronym of Standard Meta Language. It is a functional programming language and is the newest member of the family of functional languages. It was initially developed at Edinburgh by Mike Gordon, Robin Milner and Chris Wadsworth around 1975.

Pure Lambda Calculus

Pure lambda calculus has mainly three constructs: *variables*, *function application* and *function abstraction*. We use the following notational conventions. Lowercase roman letter denotes a variable e.g., x, y, z, f, g etc. and capital roman letter denotes any arbitrary λ -term e.g., M, N, P, Q etc.

Definition: λ -term is defined as follows:

- All variables are λ -terms and are called *atoms*.

- If M and N are arbitrary λ -terms, then (MN) is a λ -term, called *function application*. More usual notation for function application is M(N) but historically (MN) has become standard in λ -calculus.
- If M is any λ -term and x is any variable, then $(\lambda x . M)$ is a λ -term. This is called a *function abstraction*.

Formally, the grammar of λ -term in pure lambda calculus is defined as:

$$\langle \lambda\text{-term} \rangle ::= x \mid (MN) \mid (\lambda x . M)$$

Examples: Following are λ -terms.

1. $(\lambda x . (xy))$
2. $((\lambda y . y) (\lambda x . (xy)))$
3. $(x(\lambda x . (\lambda x . x)))$
4. $(\lambda x . (yz))$

Informal Interpretation of λ -term

A term $(\lambda x . M)$ represents a function or an operator whose value at an argument N is calculated by substituting N for all free occurrence of x in M. For example, $(\lambda x . x(xy))$ represents an operation of applying a function twice to an object denoted by y e.g., $(\lambda x . x(xy)) N = N(Ny)$.

If M has been interpreted as a function or operator, then (MN) is interpreted as the result of applying M to an argument N provided the result is meaningful. For example, if $M = (\lambda x . xy)$, then $MN = (\lambda x . xy)N = Ny$.

Definition: The λ -expression of the form $(\lambda x . y)$ is called *constant function* which when applied to any argument N gives y as a result e.g., $(\lambda x . y) N = y$ or $(\lambda x . y) M = y$.

Definition: The λ -expression of the form $(\lambda x . x)$ is called *identity function* which when applied to any argument N gives itself e.g., $(\lambda x . x) N = N$.

Pure lambda calculus is untyped and functions can be applied freely. The λ -term (xx) is valid, where variable x is applied to itself. In notion of computation, it may not be meaningful. For the sake of simplicity, the following syntactic conventions are used to minimize brackets.

- $MNPQ$ is same as $((MN)P)Q$ - (left associative).
- $\lambda x . MN$ is same as $(\lambda x . (MN))$.

Currying of function (λ - function with more arguments) :

The lambda-calculus *curries* its functions of more than one variables into a function of single argument i.e. λ -function $(\lambda x_1 x_2 \dots x_n . M)$ with more arguments can be expressed in terms of λ -function with one argument. It is right associative and written as:

$$\lambda x_1 x_2 \dots x_n . M = \lambda x_1 . (\lambda x_2 . (\dots (\lambda x_n . M) \dots))$$

When λ -function with n arguments is applied on one argument, then it reduces to a function of (n-1) arguments. For example, if $(\lambda xy . x y)$ is applied to a single argument say, z then the result is $\lambda y . z y$ which is another function. A function of more than one argument is regarded as a *function* of one variable whose value is a function of the remaining variables.

Definition: A term P is defined to be a sub term of Q (Q contains P) if one of the following

rules hold:

- P occurs in P.
- If P occurs in M or N, then P occurs in (MN).
- If (P occurs in M) or (P = x), then P occurs in ($\lambda x . M$).

Examples:

1. (xy) and x are sub terms of ((xy) ($\lambda x . (xy)$)) as there are two occurrences of (xy) and three occurrences of x.
2. (ux), (yz), u, x, y, z are all subterms of ux(yz) whereas x(yz) is not. The term ux(yz) is represented as (ux)(yz).

Definition: For a particular occurrence of ($\lambda x . M$) in a term P, the occurrence of M is called the *scope* of this particular λ .

Example: Consider a formula ($\lambda_1 y . yx (\lambda_2 x . y (\lambda_3 y . z)x$)vw, the scopes of various λ 's are given below:

- Scope of λ_1 is $yx (\lambda_2 x . y (\lambda_3 y . z)x$)
- Scope of λ_2 is $y(\lambda_3 y . z)x$
- Scope of λ_3 is z
- vw does not fall under the scope of any λ .

Definition: An occurrence of a variable x in a term P is bound if and only if, it is in a part of P with the form ($\lambda x . M$) otherwise x is said to be free variable.

If x has at least one free occurrence in a term P then it is called a free variable of P. The set of all such variables is denoted by FV(P).

Definition: A term is closed if it does not have free variables.

Examples: Consider the following λ -terms. Find out the free variables in each case.

1. $N = xv(\lambda y . y v)w$, then, $FV(N) = \{x, v, w\}$
2. $M = (\lambda y . y x (\lambda x . y (\lambda y . z) x))v w$, then, $FV(M) = \{x, z, v, w\}$

Substitutions

The notation $[x | N]M$ means to obtain the result after substituting N for each free occurrence of x in a term M. The substitution is said to be valid if no free variable in N becomes bound after substitution. There are few substitution rules as given below:

1. $[x | N]x = N$
2. $[x | N]y = y$, $\forall y \neq x$
3. $[x | N](P Q) = ([x | N]P [x | N]Q)$
4. $[x | N](\lambda x . M) = \lambda x . M$ (since x is bound in M and can not be replaced by N)
5. $[x | N](\lambda y . M) = \lambda y . [x | N] M$, if $y \neq x$ and $\{y \notin FV(N) \text{ or } x \notin FV(M)\}$.
6. $[x | N](\lambda y . M) = \lambda z . [x | N] [z | y] M$, if $y \neq x$ and $y \in FV(N)$ and $x \in FV(M)$.

Examples: Consider x, y and z to be distinct variables. Evaluate the following λ -expressions by using substitution rules.

1. $[x | z] (\lambda y . x)$

We get

$$\begin{aligned} \lambda y . [x | z] x & \quad \{ \text{using rule (5)} \} \\ \lambda y . z & \quad \{ \text{using rule (1)} \} \end{aligned}$$

Hence, $[x | z] (\lambda y . x) = \lambda y . z$

2. $[x | y] (\lambda y . x)$

We get,

$$\begin{aligned} [x | y] (\lambda y . x) & = \lambda z . [x | y] [y | z] x \quad \{ \text{using rule (6)} \}, \\ & \quad \text{since rule (5) can not be applied as } y \in \text{FV}(N) = \{y\}. \\ & = \lambda z . [x | y] x \quad \{ \text{using rule (2)} \} \\ & = \lambda z . y \quad \{ \text{using rule (1)} \} \end{aligned}$$

Hence, $[x | y] (\lambda y . x) = \lambda z . y$

In both 1 and 2, $(\lambda y . x)$ represents a constant function.

3. $[x | (\lambda y . xy)] (\lambda y . x(\lambda x . x))$

Let $N = (\lambda y . xy)$, $\text{FV}(N) = \{x\}$

and $M = x(\lambda x . x)$, $\text{FV}(M) = \{x\}$

$$\begin{aligned} [x | (\lambda y . xy)] (\lambda y . x(\lambda x . x)) & = \lambda y . [x | (\lambda y . xy)] x (\lambda x . x) \\ & \quad \{ \text{using rule (5)} \} \\ & \quad \text{since } y \text{ does not belong to } \text{FV}(N) \\ & = \lambda y . (\lambda y . xy) [x | (\lambda y . xy)] (\lambda x . x) \quad \{ \text{using rule (3)} \} \\ & = \lambda y . (\lambda y . xy) (\lambda x . x) \quad \{ \text{using rule (4)} \} \end{aligned}$$

Hence, $[x | (\lambda y . xy)] (\lambda y . x(\lambda x . x)) = \lambda y . (\lambda y . xy) (\lambda x . x)$

4. $[x | (\lambda y . vy)] (y (\lambda v . xv))$

$$\begin{aligned} [x | (\lambda y . vy)] (y (\lambda v . xv)) & \\ & = [x | (\lambda y . vy)] y [x | (\lambda y . vy)] (\lambda v . xv) \quad \{ \text{using rule (3)} \} \\ & = y [x | (\lambda y . vy)] (\lambda v . xv) \\ & = y (\lambda z . [x | (\lambda y . vy)] [v | z] xv) \quad \{ \text{using rule (6)} \} \\ & = y (\lambda z . [x | (\lambda y . vy)] xz) \\ & = y (\lambda z . (\lambda y . vy) z) \end{aligned}$$

Hence, $[x | (\lambda y . vy)] (y (\lambda v . xv)) = y (\lambda z . (\lambda y . vy) z)$

Some obvious results:

1. $[x | x]M = M$
2. $[x | N]M = M$, if $x \notin \text{FV}(M)$
3. $\text{FV}([x | N]M) = \text{FV}(N) \cup (\text{FV}(M) - \{x\})$, if $x \in \text{FV}(M)$

Lemma: Let x, y, v be distinct variables and bound variables in M are not free in v, P, Q . Then the following results hold true.

1. $[v | P] [x | v]M = [x | P]M$ if $v \notin \text{FV}(M)$
2. $[v | x] [x | v]M = M$ if $v \notin \text{FV}(M)$
3. $[x | P] [y | Q]M = [y | ([x | P]Q)] [x | P]M$ if $y \notin \text{FV}(P)$
4. $[x | P] [y | Q]M = [y | Q] [x | P]M$ if $y \notin \text{FV}(P), x \notin \text{FV}(Q)$
5. $[x | P] [x | Q]M = [x | ([x | P]Q)]M$

λ -Conversion Rules

The λ -expression (λ -term) is simplified by using conversion or reduction rules. There are mainly two types of λ - conversions rules which make use of substitution explained above.

- α -conversion
- β -conversion

Definition: Let a term P contains an occurrence of $(\lambda x . M)$ and let $y \notin FV(M)$. The act of replacing $(\lambda x . M)$ by $(\lambda y . [x | y] M)$ is called a change of bound variable in P.

Definition: P α -converts to Q if and only if Q has been obtained from P by finite series of changes of bound variables. The terms P and Q have identical interpretations and play identical roles in any application of λ -calculus.

Definition: Two terms M and N are congruent if M α -converts to N. It is denoted by $M \equiv_{\alpha} N$.

α -conversion rule :

Any abstraction of the form $\lambda x . M$ can be converted to $\lambda y . [x | y]M$ provided this substitution is valid. This is called α -conversion or α -reduction rule.

Examples:

1. $\lambda x . xy \equiv_{\alpha} \lambda z . zy$, whereas $\lambda x . (xy) \not\equiv_{\alpha} \lambda y . (yy)$
2. $\lambda xy . x(xy) = \lambda x . (\lambda y . x(xy))$ (currying of function)
 $\equiv_{\alpha} \lambda x . (\lambda z . x(xz))$
 $\equiv_{\alpha} \lambda u . (\lambda z . u(uz))$
 $\equiv_{\alpha} \lambda uz . u(uz)$

β -conversion rule:

A λ -expression $(\lambda x . M)N$ represents a function $\lambda x . M$ applied to an argument N. Any λ -expression of the form $(\lambda x . M)N$ is reduced to $[x | N]M$ provided the substitution of N for x in M is valid. This type of reduction is called β -conversion or β -reduction rule. It is the most important conversion rule .

The informal interpretation of β -conversion rule is that the value of $\lambda x . M$ at N is calculated by substituting N for all free occurrence of x in M so $(\lambda x . M) N$ can be simplified to $[x | N]M$. Rules for substitution are applied which are basically formed using λ -conversion rules.

Definition: Any term of the form $(\lambda x . M) N$ is called a β -redex (redex stands for reducible expression) and the corresponding term $[x | N]M$ is called its contractum.

Definition: If a term P contains an occurrence of $(\lambda x . M) N$ and if we replace that occurrence by $[x | N]M$ to obtain a result Q , then we say that we have contracted the redex occurrence in P or P β -contracts to Q (denoted as $P \Rightarrow_{\beta} Q$).

We say that P β -reduces to Q ($P \Rightarrow_{\beta} Q$) if and only if Q is obtained from P by finite (perhaps empty) series of β -contractions.

Examples:

1. $(\lambda x . x) y \Rightarrow [x | y] x \Rightarrow y$
Hence, $(\lambda x . x) y \Rightarrow_{\beta} y$
2. $(\lambda x . x(xy)) N \Rightarrow [x | N] x(xy) \Rightarrow N(Ny)$
Hence $(\lambda x . x(xy)) N \Rightarrow_{\beta} N(Ny)$
3. $(\lambda x . y) N \Rightarrow [x | N] y \Rightarrow y$
Hence $(\lambda x . y) N \Rightarrow_{\beta} y$
4. $(\lambda x . (\lambda y . yx) z) v \Rightarrow [x | v] ((\lambda y . yx) z)$
 $\Rightarrow (\lambda y . yv) z$
 $\Rightarrow [y | z] (yv)$
 $\Rightarrow zv$
Hence $(\lambda x . (\lambda y . yx) z) v \Rightarrow_{\beta} zv$

Equality of λ -expressions

Two λ -expressions M and N are equal if they can be transformed into each other by finite sequence of λ -conversions (forward or backward). The meaning of λ -expressions are preserved after applying any conversion rule. For example, $\lambda x . x$ is same as $\lambda z . z$. We represent M equal to N by $M = N$. Equality satisfies the following properties (equivalence relation in set theory):

- *Idempotence:* A term M is equal to itself e.g., $M = M$.
- *Commutativity:* If M is equal to N, then N is also equal to M
i.e., if $M = N$, then $N = M$.
- *Transitivity:* If M is equal to N and N is equal to P then M is equal to P
i.e., if $M = N$ and $N = P$ then $M = P$.

Definition: Two λ -expressions M and N are identical, if they consists of same sequence of characters. For example, $\lambda x . xy$ and $\lambda z . zy$ are not identical but equal. So, identical expressions are equal but converse may not be true.

Computation with Pure lambda Term

Computation in the lambda-calculus is done by rewriting (reducing) a lambda-expression into a simple form as far as possible. The result of computation is independent of the order in which reduction is applied. A reduction is any sequence of λ -conversions. If a term can not be further reduced, then it is said to be in a normal form. The normal form is formally defined as follows:

Definition: A lambda-expression is said to be in *normal form* if no beta-redex (a sub expression of the form $(\lambda x . M)N$) occurs in it. For example, the normal form of the λ -expression $(\lambda x . (\lambda y . yx) z) v$ is zv .

It is not necessary that all λ -expressions have the normal forms because λ -expression may not be terminating. It is easy to see that λ -expression $(\lambda x . xxy) (\lambda x . xxy)$ does not have a normal form because it is non terminating.

- $(\lambda x . xxy) (\lambda x . xxy) \Rightarrow [(\lambda x . xxy) | x] (xxy)$
 $\Rightarrow (\lambda x . xxy) (\lambda x . xxy) y$
 $\Rightarrow (\lambda x . xxy) (\lambda x . xxy) yy$
 \vdots
 \vdots

This reduction is nonterminating

Reduction Order

For a given λ -expression, the substitution and λ -conversion rules provide the mechanism to reduce a λ -expression to normal form but these do not tell us what order to apply the reductions when more than one redex is available. Mathematician Curry proved that if an expression has a normal form, then it can be found by leftmost reduction.

Theorem: (Normalization theorem)

If M has a normal form N then there is a leftmost reduction of M to N e.g., by repeatedly reducing leftmost redex of M , the reduction will terminate with an expression N which in the normal form which can not be further reduced.

The leftmost reduction is called *lazy reduction* because it does not first evaluate the arguments but substitutes the arguments directly into the expression. *Eager reduction* is one where the arguments are reduced before substitution. These are explained in later chapter.

Theorem: (Church Rosser)

If M reduces to N and to P then there exist R such that N and P both reduce to R i.e.,

$\exists R$ such that	$M \Rightarrow_{\beta} N,$	$M \Rightarrow_{\beta} P$
	$N \Rightarrow_{\beta} R,$	$P \Rightarrow_{\beta} R$

This theorem says that λ -expressions can be evaluated in any order.

Corollaries to Church Rosser theorem:

- If N and P are normal forms of M , then N is congruent to P i.e., N and P are same except that one is obtained from other by renaming of bound variables (using α -rule).
- If the reduction terminates in normal form, then they are unique.
- If M has a normal form and $M = N$, then N has a normal form.

Applied Lambda Calculus

Constants and predefined functions are added to pure lambda calculus in order to justify the claim that functional programming languages have been originated from lambda calculus. Because of adding constants, lambda calculus becomes applied lambda calculus. All the theorems, definitions, λ -conversion rules, substitution rules of pure lambda calculus are valid for applied calculus. The λ -term in applied lambda calculus is redefined as

$$\langle \lambda\text{-term} \rangle ::= c / x / (MN) / (\lambda x . M),$$

where c corresponds to any constant whose value does not change.

Examples:

1. $(\lambda x . x)c \Rightarrow [x | c] x \Rightarrow c$
Therefore, $(\lambda x . x)c \Rightarrow_{\beta} c$
2. $(\lambda x . c)x \Rightarrow [x | c] c \Rightarrow c$
Therefore, $(\lambda x . c)x \Rightarrow_{\beta} c$

3. $(\lambda x. c) (xy) \Rightarrow [x | (xy)] c \Rightarrow c$
Therefore, $(\lambda x. c) (xy) \Rightarrow_{\beta} c$
4. $(\lambda x. xc) (\lambda y. y) \Rightarrow [x | (\lambda y. y)] (xc) \Rightarrow (\lambda y. y)c \Rightarrow c$
Therefore, $(\lambda x. xc) (\lambda y. y) \Rightarrow_{\beta} c$
5. $(\lambda y.c)((\lambda x.x x x)(\lambda x.x x x)) \Rightarrow_{\beta} c$

Representation of Constants as λ -expressions

We have extended the syntax of pure lambda to allow λ -term to be a constant and additional computation rules for replacing constants in expressions. Some of the constants are:

*true, false, if, or, and, not, numerals {0, 1, 2, ...}, arithmetic operators {+, -, *} and relational operators {<, >, =, <>} etc*

We must think of appropriate functions to represent all the constants we want. Let us begin with a well known representation for non negative integers. Consider a term of the form:

$$f(f(\dots(f x) \dots))$$

with n ($n \geq 0$) occurrence of f . We abbreviate such term as $f^n x$. This means that apply f to x exactly n times. Abstract such term so that we get, $\lambda f. \lambda x. f^n x$ which expresses the idea of n -fold application. Such a term will be chosen as a representation for integer n .

- $n = \lambda f. \lambda x. f^n x$

We can easily show that $(f^n x)$ can be written as $(n f x)$

In particular, we get,

- $0 = \lambda f. \lambda x. f^0 x$
 $= \lambda f. \lambda x. x$
- $1 = \lambda f. \lambda x. f(x)$
- $2 = \lambda f. \lambda x. f(f(x))$

Now let us see the suitable representation for the successor function.

$$\begin{aligned} \text{succ } n &= n + 1 \\ &= \lambda f. \lambda x. f^{n+1} x \\ &= \lambda f. \lambda x. f(f^n x) \\ &= \lambda f. \lambda x. f(n f x) \end{aligned}$$

Therefore,

- $\text{succ} = \lambda n. (\lambda f. \lambda x. f(n f x))$

Similar reasoning leads to the definitions:

- $\text{add} = \lambda n. \lambda m. (\lambda f. \lambda x. f^n (f^m x))$
 $= \lambda n. \lambda m. (\lambda f. \lambda x. n f (m f x))$
- $\text{mult} = \lambda n. \lambda m. (\lambda f. \lambda x. (f^m)^n x)$
 $= \lambda n. \lambda m. (\lambda f. \lambda x. n (f^m x))$
 $= \lambda n. \lambda m. (\lambda f. \lambda x. n (m f x))$

Lambda expressions corresponding to following constants.

- $\text{true} = \lambda x. \lambda y. x$
- $\text{false} = \lambda x. \lambda y. y$
- $\text{if} = \lambda x. \lambda y. \lambda z. x y z$
- $\text{or} = \lambda x. \lambda y. \text{if } x \text{ true } y$
- $\text{and} = \lambda x. \lambda y. \text{if } x \text{ } y \text{ false}$

- not = $\lambda x . \text{if } x \text{ false true}$

1. **Reduction rule:** $\text{if true } M N \Rightarrow_{\beta} M$
Proof: $\text{if true } M N = (\lambda x . \lambda y . \lambda z . x y z) \text{ true } M N$
 $\Rightarrow (\lambda y . \lambda z . \text{true } y z) M N$
 $\Rightarrow (\lambda z . \text{true } M z) N$
 $\Rightarrow \text{true } M N = (\lambda xy . x) M N$
 $\Rightarrow (\lambda y . M) N \Rightarrow M$
Therefore, $\text{if true } M N \Rightarrow_{\beta} M$

2. **Reduction rule:** $\text{if false } M N \Rightarrow_{\beta} N$
Proof: $\text{if false } M N = (\lambda xyz . x y z) \text{ false } M N$
 $\Rightarrow (\lambda yz . \text{false } y z) M N$
 $\Rightarrow (\lambda z . \text{false } M z) N$
 $\Rightarrow \text{false } M N = (\lambda xy . y) M N$
 $\Rightarrow (\lambda y . y) N \Rightarrow N$
Therefore, $\text{if false } M N \Rightarrow_{\beta} N$

Constant, *if* is a curried conditional. Above rules say that *if* with first argument as true reduces to second argument and with first argument as false reduces to third argument.

3. **Reduction rule:** $\text{or true } y \Rightarrow_{\beta} \text{true}$
Proof: $\text{or true } y = (\lambda xy . \text{if } x \text{ true } y) \text{ true } y$
 $\Rightarrow (\lambda y . \text{if true true } y) y$
 $\Rightarrow \text{if true true } y$
 $\Rightarrow \text{true}$

4. **Reduction rule:** $\text{or false } y \Rightarrow_{\beta} y$
Proof: $\text{or false } y = (\lambda xy . \text{if } x \text{ true } y) \text{ false } y$
 $\Rightarrow (\lambda y . \text{if false true } y) y$
 $\Rightarrow \text{if false true } y$
 $\Rightarrow y$

5. **Reduction rule:** $\text{and true } y \Rightarrow_{\beta} y$
Proof: $\text{and true } y = (\lambda xy . \text{if } x \text{ y false}) \text{ true } y$
 $\Rightarrow (\lambda y . \text{if true } y \text{ false}) y$
 $\Rightarrow \text{if true } y \text{ false}$
 $\Rightarrow y$

6. **Reduction rule:** $\text{and false } y \Rightarrow_{\beta} \text{false}$
Proof: $\text{and false } y = (\lambda xy . \text{if } x \text{ y false}) \text{ false } y$
 $\Rightarrow (\lambda y . \text{if false } y \text{ false}) y$
 $\Rightarrow \text{if false } y \text{ false}$
 $\Rightarrow \text{false}$

7. **Reduction rule:** $\text{not true} \Rightarrow_{\beta} \text{false}$
Proof: $\text{not true} = (\lambda x . \text{if } x \text{ false true}) \text{ true}$
 $\Rightarrow \text{if true false true}$
 $\Rightarrow \text{false}$

8. **Reduction rule:** $\text{not false} \Rightarrow_{\beta} \text{true}$
Proof: $\text{not false} = (\lambda x . \text{if } x \text{ false true}) \text{ false}$

\Rightarrow if false false true
 \Rightarrow true

Let us assume arithmetic and relational operators with usual meanings such as $3 + 4 = 7$, $(5 > 3) = \text{true}$ etc.

Arithmetic expression:

Arithmetic expression $3 + 7$ is written in prefix notation as $+ 3 7$. It is treated as an expression and written as:

$$(\lambda xy . + x y) 3 7 \quad (\lambda y . + 3 y) 7 \quad \Rightarrow \quad + 3 7 = 10$$

Now onwards, we will use the following conventions:

- infix notation for arithmetic expression such as $3 + 7$ instead of $+ 3 7$ for the sake of convenience.
- Conditional expression in programming context as "if x then y else z" instead "if x y z"

Examples:

- $(\lambda x . x + 5) 2 \Rightarrow [x | 2] (x + 5) \Rightarrow 7$
 Thus, $(\lambda x . x + 5) 2 \Rightarrow_{\beta} 7$
- $(\lambda x . x * 4) 7 \Rightarrow [x | 7] (x * 4) \Rightarrow 28$
 Thus, $(\lambda x . x * 4) 7 \Rightarrow_{\beta} 28$
- $(\lambda y . y * y) 6 \Rightarrow [y | 6] (y * y) \Rightarrow 6 * 6 = 36$
 Thus, $(\lambda y . y * y) 6 \Rightarrow_{\beta} 36$
- $(\lambda xy . x + y - 4) 5 \Rightarrow [x | 5] (\lambda y . x + y - 4) \Rightarrow (\lambda y . y + 1)$
 Thus, $(\lambda xy . x + y - 4) 5 \Rightarrow_{\beta} (\lambda y . y + 1)$
- $(\lambda x . ((\lambda y . x + y) 5)) \Rightarrow (\lambda x . ([y | 5] (x + y))) \Rightarrow \lambda x . x + 5$
 Thus, $(\lambda x . ((\lambda y . x + y) 5)) \Rightarrow_{\beta} \lambda x . x + 5$
- $(\lambda f . f 2) (\lambda y . y * 4) \Rightarrow [f | (\lambda y . y * 4)] (f 2)$
 $\Rightarrow (\lambda y . y * 4) 2 \Rightarrow [y | 2] (y * 4) \Rightarrow 8$
 $(\lambda f . f 2) (\lambda y . y * 4) \Rightarrow_{\beta} 8$
- $(\lambda xy . y * x - (y + 2)) 2 3$
 $\Rightarrow ([x | 2] (\lambda y . y * x - (y + 2))) 3$
 $\Rightarrow (\lambda y . y * 2 - (y + 2)) 3$
 $\Rightarrow [y | 3] (y * 2 - (y + 2))$
 $\Rightarrow 3 * 2 - (3 + 2)$
 $= 6 - 5 = 1$
 $(\lambda xy . y * x - (y + 2)) 2 3 \Rightarrow_{\beta} 1$
- $(\lambda x . (\lambda y . x + y) 5) ((\lambda y . y * y) 6)$
 $\Rightarrow [x | ((\lambda y . y * y) 6)] (\lambda y . x + y) 5$
 $\Rightarrow [x | 36] (\lambda y . x + y) 5$
 $\Rightarrow (\lambda y . 36 + y) 5$
 $\Rightarrow [y | 5] (36 + y)$
 $\Rightarrow 36 + 5 = 41$
 $(\lambda x . (\lambda y . x + y) 5) ((\lambda y . y * y) 6) \Rightarrow_{\beta} 41$
- $(\lambda x . \text{if } x > 4 \text{ then } x + 2 \text{ else } 1) 7 \Rightarrow [x | 7] (\text{if } x > 4 \text{ then } x + 2 \text{ else } 1)$

$$\begin{aligned}
& \Rightarrow 9 \\
& (\lambda x . \text{if } x > 4 \text{ then } x + 2 \text{ else } 1) 7 \Rightarrow_{\beta} 9 \\
10. (\lambda x . x * 4) ((\lambda x . \text{if } x > 4 \text{ then } x + 2 \text{ else } 1) 7) & \\
& \Rightarrow [x \mid ((\lambda x . \text{if } x > 4 \text{ then } x + 2 \text{ else } 1) 7)] (x * 4) \\
& \Rightarrow [x \mid 9] (x * 4) \\
& \Rightarrow 9 * 4 = 36 \\
& (\lambda x . x * 4) ((\lambda x . \text{if } x > 4 \text{ then } x + 2 \text{ else } 1) 7) \Rightarrow_{\beta} 36 \\
11. (\lambda y . y * 2 - y / 5 + (y + 4)) (x * 3) & \\
& \Rightarrow [y \mid (x * 3)] (y * 2 - y / 5 + (y + 4)) \\
& \Rightarrow (x * 3) * 2 - (x * 3) / 5 + ((x * 3) + 4)
\end{aligned}$$

Therefore, we can write an expression $(x * 3) * 2 - (x * 3) / 5 + ((x * 3) + 4)$ using λ -notation as $(\lambda y . y * 2 - y / 5 + (y + 4)) (x * 3)$

Function definition using λ -notation

λ -function can be named or unnamed. Unnamed function has to be applied directly to argument values as shown earlier whereas named function are applied by using function name along with the arguments. The λ -function has the following valid form.

$$\text{fun_name} \equiv \lambda \text{ list of arguments . expression}$$

Expression can be simple or conditional. In conventional mathematical usage, the application of n -argument function f to arguments x_1, \dots, x_n is written as $f(x_1, \dots, x_n)$. In λ -calculus, there are two ways of representing such application.

- $(f \ x_1 \dots x_n)$ - it is a curried representation where f expects its argument one at a time. Important advantage of curried functions is that they can be partially applied.
- Application of f to n -tuple (x_1, \dots, x_n) such as $f(x_1, \dots, x_n)$, where all the arguments should be available to function at the first application of it.

Examples:

$$\begin{aligned}
1. f & \equiv \lambda x . 2 * x + 3 \\
f 3 & = (\lambda x . 2 * x + 3) 3 \Rightarrow_{\beta} [x \mid 3] (2 * x + 3) \\
& \Rightarrow_{\beta} 9
\end{aligned}$$

$$\text{Similarly, } f 1 = 5.$$

$$\begin{aligned}
2. g & \equiv \lambda x . \text{if } x > 4 \text{ then } f \ x \ \text{else } -x \\
3. g 10 & = (\lambda x . \text{if } x > 4 \text{ then } f \ x \ \text{else } -x) 10 \\
& \Rightarrow_{\beta} [x \mid 10] (\text{if } x > 4 \text{ then } f \ x \ \text{else } -x) \\
& \Rightarrow_{\beta} f 10 \Rightarrow_{\beta} [x \mid 10] (2 * x + 3) \\
& = 23.
\end{aligned}$$

$$\text{Similarly, } g 3 = -3.$$

$$\begin{aligned}
4. \text{ plus} & \equiv \lambda xy . x + y \\
5. \text{ plus } 5 7 & = (\text{plus } 5) 7 = ((\lambda xy . x + y) 5) 7 \\
& = ((\lambda x . \lambda y . x + y) 5) 7 \\
& \Rightarrow_{\beta} [x \mid 5] (\lambda y . x + y) 7 \\
& \Rightarrow_{\beta} (\lambda y . 5 + y) 7 \\
& \Rightarrow_{\beta} [y \mid 7] (5 + y) = 12 \\
6. \text{ times} & \equiv \lambda xy . x * y \\
\text{times } 4 6 & = (\text{times } 4) 6 = ((\lambda xy . x * y) 4) 6
\end{aligned}$$

$$\begin{aligned}
&= ((\lambda x. \lambda y. x * y) 4) 6 \\
&\Rightarrow_{\beta} [x | 4] (\lambda y. x * y) 6 \\
&\Rightarrow_{\beta} (\lambda y. 4 * y) 6 \\
&\Rightarrow_{\beta} [y | 6] (4 * y) = 24 \\
7. \text{ f1} &\equiv (\lambda x. \text{times } x \ x) \\
\text{f1 (plus 3 2)} &= (\lambda x. \text{times } x \ x) (\text{plus 3 2}) \\
&\Rightarrow_{\beta} [x | (\text{plus 3 2})] (\text{times } x \ x) = [x | 5] (\text{times } x \ x) \\
&\Rightarrow_{\beta} \text{times } 5 \ 5 = 25 \\
8. \text{ g1} &\equiv (\lambda x. \text{plus } x \ 1) \\
\text{g1 } ((\lambda y. \text{times } y \ y) 3) &= (\lambda x. \text{plus } x \ 1) ((\lambda y. \text{times } y \ y) 3) \\
&\Rightarrow_{\beta} [x | (\lambda y. \text{times } y \ y) 3] (\text{plus } x \ 1) \\
&\Rightarrow_{\beta} [x | (\text{times } 3 \ 3)] (\text{plus } x \ 1) \\
&\Rightarrow_{\beta} [x | 9] (\text{plus } x \ 1) \\
&\Rightarrow_{\beta} \text{plus } 9 \ 1 = 10
\end{aligned}$$

Recursive Definitions in λ - Notation

We have seen that how we could use λ -expression to write an applicative expression which computes the same result as a sequence of assignments and conditional statements. However we can't perform the equivalent of iteration or recursion which are important constructs of programming languages. To do this we must allow recursive definitions for λ -expression. We do not require new syntax except that we give a name to λ -expression so that we can apply it to itself. Also in order to make the recursion terminate, we shall use conditional expressions. Recursion is achieved in two ways viz., Downgoing and Upgoing Recursion.

Downgoing Recursion

Downgoing style of recursion keeps breaking the problem down recursively into simpler version until a terminal case is reached. Then it starts building the result upward by passing intermediate results back to calling functions. While writing function using downgoing recursion, we should follow the following tips:

- Definition for terminal case (0 – for numbers, [] – for list).
- For non terminal case with argument 'n', assume we have definition for (n-1), use this to construct the next case up.
- Combine above mentioned cases in a conditional expression.

The following examples use downgoing recursion. Now onwards, we will use equality sign (=) instead of \Rightarrow_{β} for the sake of convenience.

Examples:

1. Write λ -function for computing factorial of n (positive integer)

$$\begin{aligned}
\text{fact} &\equiv \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact } (n-1) \\
\text{fact } 3 &= (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact } (n-1)) (3) \\
&= 3 * \text{fact } 2 \\
&= 3 * 2 * \text{fact } 1 \\
&= 3 * 2 * 1 * \text{fact } 0 \\
&= 3 * 2 * 1 * 1 \\
&= 6
\end{aligned}$$

2. Write λ -function for computing combinatorial function (nC_r) (n and r are positive integers)

$$comb \equiv \lambda nr . fact (n) \text{ div } (fact r * fact (n-r))$$

OR

$$comb \equiv \lambda nr . \text{if } r = 1 \text{ then } n \text{ else } (n * comb (n-1) (r-1)) \text{ div } r$$

$$\begin{aligned} comb\ 4\ 2 &= 4 * comb\ 3\ 1 \text{ div } 2 \\ &= 4 * 3 \text{ div } 2 = 6 \end{aligned}$$

3. Compare two whole numbers for equality

$$equal \equiv \lambda xy . \text{if } x = 0 \text{ then if } y = 0 \text{ then true else false else equal } (x-1) (y-1)$$

$$\begin{aligned} equal\ 2\ 3 &= equal\ 1\ 2 \\ &= equal\ 0\ 1 \\ &= false \end{aligned}$$

Upgoing Recursion

In upgoing recursion, the intermediate results are computed at each stage of recursion, thus building up the answer and passing it in a workspace parameter until the terminal case is reached. At this stage the final result is already computed. This style of recursion is similar to iteration having same complexity in terms of space and computing time. We use the following tips.

- Construct a new function with an additional workspace parameter to build up the result.
- Set workspace parameter equal to the value of the function for terminal case.
- For non terminal case, call the function with new parameter expressed in terms of old.

Let us write function for computing factorial using this approach.

$$fact \equiv \lambda n . \text{factorial } n\ 1$$

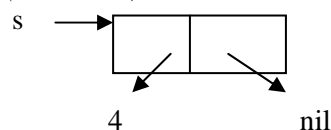
$$\text{factorial} \equiv \lambda nw . \text{if } n = 0 \text{ then } w \text{ else factorial } (n-1) (n * w)$$

Here 'w' is a workspace parameter used to build up the results. Since original function 'fact' has only one parameter 'n', we have to define an auxiliary function 'factorial' with extra parameter 'w'. The function 'fact' calls 'factorial' with workspace parameter initialized to 1. Let us see the working of this function.

$$\begin{aligned} fact\ 3 &= \text{factorial } 3\ 1 \\ &= \text{factorial } 2\ (3*1) \\ &= \text{factorial } 1\ (2*3) \\ &= \text{factorial } 0\ (1*6) \\ &= 6 \end{aligned}$$

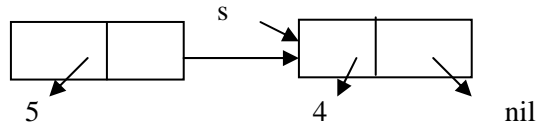
Recursive List Processing

Consider list processing applications which require a list constructor, often called *cons*, and written as an infix operator '::' in many functional languages. The constant *nil* denoted by [] represents the empty list. The functions *car* (for getting first element of the list) and *cdr* (for getting tail of the list) dismantle a list and help us to separate head from tail of the list. The function *null* tests a list whether it is empty or not. A list is constructed using function *cons* whose *car* field contains first argument and *cdr* field points to second argument of *cons* function. For example, a list $s = (cons\ 4\ nil)$ has the following graphical representation.

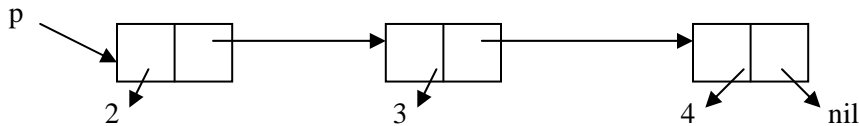


Example:

1. Graphical representation of a list $(cons\ 5\ s)$, where s is a list shown above.



2. Graphical representation of a list $p = \text{cons } 2 (\text{cons } 3 (\text{cons } 4 \text{ nil}))$



Now,

$\text{car } s = 2$
 $\text{cdr } s = \text{cons } 3 (\text{cons } 4 \text{ nil})$

For the sake of convenience, we write a list $\text{cons } 2 (\text{cons } 3 (\text{cons } 4 \text{ nil}))$ as $[2,3,4]$, a conventional notation in functional programming languages.

Important list functions using λ - Notation

1. Membership function:

$\text{mem} \equiv \lambda x s. \text{if } \text{null } s \text{ then } \text{false} \text{ else if } x = \text{car } s \text{ then } \text{true} \text{ else } \text{mem } x (\text{cdr } s)$
 $\text{mem } 2 [3,4,2] = \text{mem } 2 [4,2]$
 $= \text{mem } 2 [2]$
 $= \text{true}$

2. Write a function that selects nth element of the list.

$\text{select} \equiv \lambda n s. \text{if } \text{null } s \text{ then } \text{nil} \text{ else if } n = 1 \text{ then } \text{car } s \text{ else } \text{select } (n-1) (\text{cdr } s)$
 $\text{select } 2 [4,6,7,8] = \text{select } 1 [6,7,8]$
 $= \text{car } [6,7,8]$
 $= 6$
 $\text{select } 3 [4,6] = \text{select } 2 [6]$
 $= \text{select } 1 []$
 $= \text{nil}$

3. Concatenating two lists.

$\text{append} \equiv \lambda m n. \text{if } \text{null } m \text{ then } n \text{ else } \text{cons } (\text{car } m) (\text{append } (\text{cdr } m) n)$
 $\text{append } [2,3] [4,5,6,7] = \text{cons } 2 (\text{append } [3] [4,5,6,7])$
 $= \text{cons } 2 (\text{cons } 3 (\text{append } [] [4,5,6,7]))$
 $= \text{cons } 2 (\text{cons } 3 [4,5,6,7])$
 $= \text{cons } 2 [3,4,5,6,7]$
 $= [2,3,4,5,6,7]$

Here elements of the first list are copied and last node points to the second list. Therefore, the second list is no longer available. One can write another version of append where entirely new list is created and original lists are also available. The brackets are used for clarity otherwise they can be removed.

4. Copy a list using both types of recursions

Downgoing version:

$\text{copy} \equiv \lambda s. \text{if } \text{null } s \text{ then } \text{nil} \text{ else } \text{cons } (\text{car } s) (\text{copy } (\text{cdr } s))$

Upgoing version:

```

copy    ≡    copy1 s []
copy1  ≡    λsw. if null s then w else
            copy1 (cdr s) (append w (list (car s)))

```

Here a function *list* is used to convert an element (head in this case) into list. Let us see the working of both the functions.

Downgoing version:

```

copy [2,3,4]    =    cons 2 (copy [3,4])
                =    cons 2 (cons 3 (copy [4]))
                =    cons 2 (cons 3 (cons 4 (copy [])))
                =    cons 2 (cons 3 (cons 4 nil))
                =    cons 2 (cons 3 [4])
                =    cons 2 [3,4]
                =    [2,3,4]

```

Upgoing version:

```

copy [2,3,4]    =    copy1 [2,3,4] []
                =    copy1 [3,4] (append [] [2])
                =    copy1 [4] (append [2] [3])
                =    copy1 [] (append [2,3] [4])
                =    [2,3,4]

```

5. Reverse a given list:

Downgoing version:

```

rev    ≡    λs. if null s then [] else append rev (cdr s) (list car s)

```

Upgoing version:

```

rev    ≡    rev1 s []
rev1   ≡    λsw. if null s then w else rev1 (cdr s) (cons (car s) w)

```

6. Find out common elements of two lists

```

common ≡    λmn. if null m then [] else if mem (car m) n then cons (car m)
            (common (cdr m) n) else common (cdr m) n)

```

Tree recursion

Tree is represented using generalized list whose elements may be lists themselves.

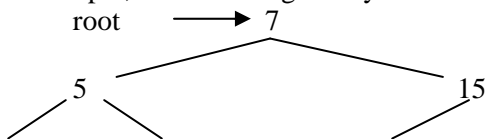
The representation of general tree is given as follows:

[root, br₁, br₂, ..., br_n], where br_i is ith branch of a root and is tree in itself.

The representation of binary tree (tree with node having at most two branches) is special case of general representation.

[root, left_br, right_br], where left_br and right_br are left and right binary sub trees.

For example, the following binary tree is represented using list representation as follows:



2

6

10
 \ 12

[7, [5, [2, nil, nil], [6, nil, nil]] , [15, [10, nil, [12, nil, nil]], nil]]

Examples:

1. Count number of nodes in a binary tree.

$$tcount \equiv \lambda t. \text{if null } t \text{ then } 0 \text{ else } 1 + tcount \text{ (car (cdr t))} + tcount \text{ (car (cdr (cdr t)))}$$

We need an extra function called *atom* which when applied on a binary tree becomes true if element of tree is an atom otherwise false for if it is list. Let us see the following functions for creating mirror copy of a binary tree

2. Copying a binary tree

$$tcopy \equiv \lambda t. \text{if atom } t \text{ then } t \text{ else if null } t \text{ then nil else cons (tcopy (car t)) (tcopy (cdr t))}$$

Binary Search Tree

Definition: *Binary search tree* is a binary tree if all the keys on the left of any node (say, N) are numerically (alphabetically) less than the key in a node N and all the keys on the right of N are numerically (alphabetically) greater than the key of a node N.

3. Search for an element in binary search tree.

$$bsearch \equiv \lambda t x. \text{if null } t \text{ then false else if } x = (\text{car } t) \text{ then true else if } x < (\text{car } t) \text{ then bsearch (car (cdr t)) } x \text{ else bsearch (car (cdr (cdr t))) } x$$

4. Insert an element

$$binsert \equiv \lambda t x. \text{if null } t \text{ then (cons } x \text{ t) else if } x < (\text{car } t) \text{ then binsert (car (cdr t)) } x \text{ else binsert (car (cdr (cdr t))) } x$$

Introduction to SML

SML is acronym of Standard Meta Language. It is a functional programming language and is the newest member of the family of functional languages. It was initially developed at Edinburgh by Mike Gordon, Robin Milner and Chris Wadsworth around 1975. Since then numerous variations and implementations have arisen. SML has basic data objects as expressions, functions and list etc. Function is the first class data object that may be passed as an argument, returned as a result and stored in a variable. SML is interactive in nature where each data object is entered, analyzed, compiled and executed. The value of the object is reported along with its type. SML is strongly typed language meaning by that each data object has a type determined automatically from its constituents by the interpreter if not specified. SML has a polymorphic typing mechanism in which the type of data object is determined using the context. It is statically scoped language where the scope of a variable is determined at compile time that helps more efficient and modular programs development. It also has exception handling facilities using which one can handle unusual situations at run time. SML also supports abstract data types that are useful mechanism for program modularization.

Interaction with SML

Basic form of interaction is *read, evaluate* and *display*. An expression is entered and terminated by semi colon (;). An expression is analysed, compiled and executed by SML interpreter and the result is printed on the terminal along with its type. In SML, the *type* is a collection of values. The basic types are *int* (integers), *real* (real), *char* (character), *bool* (boolean) and *string* (sequence of character). In addition to the basic types, SML also allows user defined types (discussed later). An *expression* in SML denotes a *value* and the type of an expression is uniquely determined from its constituents. If it does not succeed in identifying the type, then some error message is issued.

Conventions:

We have used the following conventions in order to distinguish between user input and SML system's response.

- The SML system prompts with “ - “ for an expression to be entered by an user.
- It displays the output after the symbol “ > “. The output is shown in *italic*.

Examples:

```
-      2 + 5;           ← user's input
>      val it = 7 : int ← system's response
-      3 + 2.5;
>      Error: operator and operand don't agree
```

The result starts with the reserved word *val* that indicates that the value has been computed and is assigned to system defined identifier *it* along with its type implicitly derived by the system from expression's constituents.

Example:

```
-      2 > 5 ;
```

```
> val it = false : bool
```

Each time a new expression is entered, the value of *it* gets changed. An expression is of boolean type that gets evaluated to *false* value. This value is assigned to *it* followed by its type *bool*. Now the previous value of *it* is not available. Few examples are given below:

Examples:

```
- not false;
> val it = true : bool
- ~3.45 ;           {~ is unary minus }
> val it = ~3.45 : real
- (25 + 5) mod 2 ;   {mod gives remainder}
> val it = 0 : int
- floor (3.5 + 2.7) ; {floor gives integral part of real value}
> val it = 6 : int
- abs (~ 6);        {abs gives absolute value}
> val it = 6 : int
```

Value Declaration

Value can be given a name called *variable* or *identifier*. The value of a variable can not be updated and the life time of a variable is until it is redefined. The keyword *val* is used to define the value of a variable.

For example, the execution of a value declaration *val var = exp* causes an variable *var* to be bound to the value of an expression *exp*. The name of a variable is formed by using alphanumeric characters [a – z, A – Z, numerals, underscore (*_*) and primes (*'*)] and it must start with letter. If value is named by a variable, then it is bound to that variable otherwise it is bound to system defined identifier *it*.

Examples:

```
- val x = 3 + 5 * 2 ;   ← user's input
> val x = 13 : int     ← system's response
- val y = x + 3;
> val y = 16 : int
- y + x ;             ← without value declaration
> val it = 29 : int
- val x = ~1.23E-8 ;   ← x is redefined
> val x = ~1.23E8 : real {~1.23E8 denotes -1.23*108}
- val t = y + x ;
> Error: operator and operand don't agree
```

Bindings and Environments

We have seen that a variable can be bound to the value of an expression on the right hand side. The collection of bindings at any particular state is called an *environment* of that state. Execution of any declaration causes extension or change in the environment. The notation used for environment is not of SML program but our own notation to explain the meaning of SML programs. For example, the execution of the value declaration *val x = 3 + 5 * 2* causes the environment *env = [x |⇒ 13 : int]*. Each execution creates updated environment.

Examples:

```
- val x = 3 + 5 * 2;
```

```

> val x = 13 : int
      env1 = [x |⇒ 13 : int ]
-   val y = x + 3;
>   val y = 16 : int
      env2 = [x |⇒ 13 : int, y |⇒ 16 : int]
-   y + x ;
>   val it = 29 : int
      env3 = [x |⇒ 13 : int , y |⇒ 16 : int, it |⇒ 29 : int]
-   val x = ~1.23E~8 ;
>   val x = ~1.23E8 : real
      env4 = [y |⇒ 16 : int, it |⇒ 29 : int, x |⇒ -1.23*108 : real ]
-   y;
>   val it = 16 : int

```

Multiple Bindings

Multiple variables can be bound simultaneously using key word *and* as a separator. It is also called *simultaneous declaration*. A *val* declaration for simultaneous declaration is of the form

$val\ v_1 = e_1\ and\ v_2 = e_2\ and\ \dots\ and\ v_n = e_n$

SML interpreter evaluates all the expressions e_1, e_2, \dots, e_n and then bounds the variables v_1, v_2, \dots, v_n to have the corresponding values. Since the evaluations of expressions are done independently, the order is immaterial. Consider some more examples and continue with the previous environment $env5 = [y |⇒ 16 : int, x |⇒ -1.23*10^8 : real, it |⇒ 16 : int]$

Examples:

```

-   val y = 3.5 and x = y ;
>   val y = 3.5 : real
>   val x = 16 : int
      env6 = [ it |⇒ 16 : int, y |⇒ 3.5 : real, x |⇒ 16 : int]

```

In multiple value bindings, the values on right hand sides are evaluated first and then bound to the corresponding variable in the left hand sides. Therefore, in the above example x does not get the current value of y which is 3.5 but bound to the value 16 available from the previous environment $env5$.

Examples:

```

-   val y = y + 3.0 and x = y ;
>   val y = 6.5 : real
>   val x = 3.5 : real
      env7 = [ it |⇒ 16 : int, y |⇒ 6.5 : real, x |⇒ 3.5 : real]

```

Compound Declarations

Two or more declarations can be combined and separated by semicolon. The general form of *compound declaration* is $D_1; D_2; \dots; D_n$. SML first evaluates the first declaration D_1 , produces an environment, then evaluates the second declaration D_2 , updates the previous environment and proceeds further in the sequence. It must be noted that the subsequent declarations in sequential composition may override the identifiers declared in the left hand side declarations. Consider few examples and consider the previous environment as

$env7 = [it |⇒ 16 : int, y |⇒ 6.5 : real, x |⇒ 3.5 : real]$

Examples:

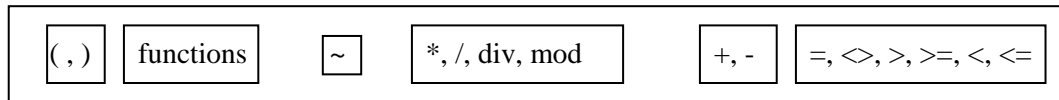
```

-      val x = 34; val x = true  and  z = x ; val z = x;
>      val x =  34 : int
          env8 = [ it |⇒ 16 : int, y |⇒ 6.5 : real, x |⇒ 34 : int]
>      val x =  true : bool
>      val z =  34 : int
          env9 = [it |⇒ 16 : int, y|⇒ 6.5: real, x|⇒ true:bool, z|⇒ 34 : int]
>      val z =  true : bool
          env10 = [ it|⇒ 16:int, y|⇒6.5:real, x|⇒ true:bool, z|⇒ true:bool]

```

Expressions and Precedence

Expressions in SML are evaluated according to operator precedence. The higher precedence means earlier evaluation. Equal precedence operators are evaluated from left to right. Operators are of two kinds viz., infix operator and unary operator. *Infix operator* is placed between two operands and is also called *dyadic operator*. An *unary operator* is always written in front of an operand and has higher precedence than any infix operator. It is also called *monadic operator*. In SML, infix minus is represented by - whereas unary minus represented by ~. Function application also has higher precedence than any infix operator. Precedence of operators is shown below in decreasing order without mentioning the actual priority value. Operators enclosed in inner boxes indicate operators with equal priority values.



For example, fully parenthesized expression according to the precedence of operators for an expression $\sim 5 + 56 \text{ mod } 4 * 10 > 34 - 45 \text{ div } 3 + \text{abs } \sim 3$ is $((\sim 5) + ((56 \text{ mod } 4) * 10)) > ((34 - (45 \text{ div } 3)) + (\text{abs } (\sim 3)))$

Conditional Expressions

The general form of *conditional expression* is *if E then E₁ else E₂*. It is evaluated by evaluating first the *boolean expression* E and then depending on the value of E, either than part or else part (but not both) is evaluated i.e., if E is true then E₁ is the value of the conditional expression otherwise E₂ is the value. The type of E₁ and E₂ should be the same whereas the type of E is *bool*. Conditional expression depends on boolean expression (called condition in short). The *condition* is formed using *arithmetic, relational, boolean* and *string* operators. It is evaluated using precedence of operators and gives true or false value. Priority of operators is: arithmetic operators, relational operators followed by boolean / logical operators.

Arithmetic Operators: (Precedence has been shown earlier)

Integers : +, -, *, div, mod, abs, ~ (unary minus)

Real : +, -, *, /, sqrt, floor, sin, cos etc.

Arithmetic operators +, -, and * are defined for both integers and reals and thus are overloaded. The operators are overloaded if defined for more than one type of datatypes. SML can deduce the type in most of the expressions, functions from the type of the constituents used.

Relational Operators: (All operators have equal precedence)

Integers & reals < (less than),

<= (less or equal to),

	>	(greater than),
	>=	(greater or equal to)
For all except reals	=	(equal to),
	<>	(not equal to)

Boolean Operators: (Precedence is in decreasing order of *not*, *andalso* and *orelse*)

<i>not</i>	(Logical negation),
<i>andalso</i>	(Logical AND),
and <i>orelse</i>	(Logical OR)

The boolean operators *andalso* and *orelse* are evaluated using lazy evaluation strategy which means that evaluate whenever it is required (explained in detail later). For example, a boolean expression or condition $2 + 3 > x - 2$ *andalso* *true* *orelse* $x = y$ when evaluated gives true or false value.

Examples:

(for conditions):

```
-      val n = ~ 6 ;
>      val n = ~ 6 : int
           env = [ n |⇒ ~ 6 : int]
-      val x = true andalso false;
>      val x = false : bool
           env = [ n |⇒ ~ 6 : int, x |⇒ false : bool ]
-      val y = x orelse true;
>      val y = true : bool
           env = [ n |⇒ ~ 6 : int, x |⇒ false : bool, y |⇒ true : bool ]
-      val z = x orelse not(y);
>      val z = false : bool
           env = [n|⇒ ~ 6:int, x|⇒ false:bool, y|⇒ true
```

(for conditional expressions):

```
-      val t = if n +3 > 0 orelse y then 10 else 16 ;
>      val t = 10 : int
env = [ n |⇒ ~ 6 : int, x |⇒ false : bool, y |⇒ true : bool, z |⇒ true : bool, t |⇒ 10: int]
-      val t2 = if x andalso y orelse z then n + 10 else 30;
>      val t = 30 : int
env = [ n |⇒ ~ 6 : int, x |⇒ false : bool, y |⇒ true : bool, z |⇒ true : bool, t |⇒ 30: int]
-      val n = if x andalso y orelse not z then 10 else 23;
>      val n = 10 : int
```

for conditional expressions):

```
-      val t = if n +3 > 0 orelse y then 10 else 16 ;
>      val t = 10 : int
env = [ n |⇒ ~ 6 : int, x |⇒ false : bool, y |⇒ true : bool, z |⇒ true : bool, t |⇒ 10: int]
-      val t2 = if x andalso y orelse z then n + 10 else 30;
>      val t = 30 : int
env = [ n |⇒ ~ 6 : int, x |⇒ false : bool, y |⇒ true : bool, z |⇒ true : bool, t |⇒ 30: int]
-      val n = if x andalso y orelse not z then 10 else 23;
>      val n = 10 : int
```

Characters and strings

Characters are encoded using ASCII (American Standard Code for Information Interchange) coding scheme. In SML, a character is enclosed within double quotes and is preceded by a

symbol #. A *string* is a sequence of characters except double quote and is also enclosed within double quotes. There are various built-in functions to manipulate strings. Some of the functions are *size*, *ord*, *chr*, *^* (symbol for concatenation of strings) etc.

- Function *size* returns the length of a string.
- An ordinal function denoted by *ord* yields ASCII code corresponding to a character as an integer between 0 – 127.
- Function *chr* is an inverse function of *ord*, gives character corresponding to a code. For instance, 97 and 51 are ASCII codes for characters "a" and "3" respectively.
- The concatenation operator (*^*) joins two strings.

Few examples are given to illustrate the use of these built-in functions. Consider the previous environment and continue.

Examples:

```

-      val x = #"t";
>      val x = #"t" : char
env = [y |⇒ true : bool, z |⇒ true : bool, t |⇒ 30: int, n |⇒ 10: int, x |⇒ # "t" : char]
-      val y = "prog" ^ "ram"
>      val y = "program" : string
env = [z |⇒ true : bool, t |⇒ 30: int, n |⇒ 10: int, x |⇒ # "t" : char, y |⇒ "program" : string]
-      val t = size "testing";
>      val t = 7 : int
env = [z |⇒ true : bool, n |⇒ 10: int, x |⇒ # "t" : char, y |⇒ "program" : string , t |⇒ 10: int]
-      val n = size ("to" ^ "gether");
>      val n = 8 : int
env = [z |⇒ true : bool, x |⇒ # "t" : char, y |⇒ "program" : string , t |⇒ 10: int, n |⇒ 8: int]
-      val x = ord #"a";
>      val x = 97 : int
env = [z |⇒ true : bool, y |⇒ "program" : string , t |⇒ 10: int, n |⇒ 8: int, x |⇒ 97 : int]
-      val z = chr 51;
>      val z = #"3" : char
env = [y |⇒ "program" : string , t |⇒ 10: int, n |⇒ 8: int, x |⇒ 97 : int, z |⇒ #"3" : char]

```

Function Declaration

Functions are also values in SML and are defined in much the same way as in mathematics. A function declaration is a form of value declaration and so SML prints the value and its type. The general form of function definition is:

fun fun_name (argument_list) = expression

The keyword *fun* indicates that function is defined. fun_name is user defined variable and argument_list consists of arguments separated by comma. Let us write a function for calculating circumference of a circle with radius *r*.

Examples:

```

-      val pi = 3.1414;
>      pi = 3.1414 : real
env = [ pi |⇒ 3.1414 : int ]
-      fun circum ( r ) = 2.0 * pi * r;
>      val circum = fn : real → real
env = [ pi |⇒ 3.1414 : int, circum |⇒ fn r ⇒ 2.0 * pi * r : real → real]
-      circum (3.0);
>      val it = 18.8484 : real

```

```
env = [ pi |⇒ 3.1414 : int, circum |⇒ fn r ⇒ 2 * 3.1414 * r : real → real, it |⇒ 18.8484 : real]
```

Here *circum* is a function name that takes an argument of real type and returns real value. SML can infer the type of an argument from an expression ($2.0 * \pi * r$). Since multiplication can be done between two real values as SML is strongly typed, so *r* is also real. If there is one argument of a function, then circular brackets can be removed. The same function can be written as:

Examples:

```
- fun circum r = 2.0 * pi * r;  
> val circum = fn : real → real  
env = [ pi |⇒ 3.1414 : int, it |⇒ 18.8484 : real, circum |⇒ fn r ⇒ 2 * 3.1414 * r : real → real]
```

```
- circum 1.5;  
> val it = 9.4242 : real  
env = [ pi |⇒ 3.1414 : int, circum |⇒ fn r ⇒ 2 * 3.1414 * r : real → real, , it |⇒ 9.4242 : real]
```

```
env = [circum |⇒ fn r ⇒ 2 * 3.1414 * r : real → real, pi |⇒ 1.0 : real, it |⇒ 9.4242 : real]
```

If the function body contains identifiers which are not in the list of arguments, then they are called *free* variables. Variables appearing in the argument list are said to be *bound*. In the function *circum*, *pi* is a free identifier whereas *r* is bound.

Examples:

```
- val pi = 1.0;  
> val pi = 1.0 : real  
env = [circum |⇒ fn r ⇒ 2 * 3.1414 * r : real → real, , it |⇒ 9.4242 : real, pi |⇒ 1.0 : real]  
- circum 1.5;  
> val it = 9.4242 : real
```

We notice that the result of *circum 1.5* is still 9.4242 even though *pi* is bound to new value to 1.0. The reason is that SML uses the environment valid at the time of declaration of function rather than the one available at the time of function application. This is called the *static binding* of free variables in the function.

Examples:

```
- pi;  
> val it = 1.0 : real  
env = [circum |⇒ fn r ⇒ 2 * 3.1414 * r : real → real, pi |⇒ 1.0 : real, it |⇒ 1.0 : real]  
- val circum = 2.5;  
> val circum = 2.5 : real  
env = [pi |⇒ 1.0 : real, it |⇒ 1.0 : real, circum |⇒ 2.5 : real]  
- val x = pi + circum;  
> val x = 3.5 : real  
env = [pi |⇒ 1.0 : real, it |⇒ 1.0 : real, circum |⇒ 2.5 : real, x |⇒ 3.5 : real]
```

Since the variable *circum* gets new binding, the environment gets updated with the value of *circum* as 1.0 and function *circum* is removed from the modified environment.

Unnamed Function

Unnamed Function is defined using a functional expression. Its general form is:

```
fn ( argument_list ) => expression ;
```

If *argument_list* consists of one argument, then brackets are optional.

Examples:

```

-      fn r => 3.1414 * r * r ;
>      val it = fn : real → real
env = [ it |⇒ fn r => 3.1414 * r * r : real → real ]
-      val x = it 1.0;
>      val x = 3.1414 : real
env = [ it |⇒ fn r => 3.1414 * r * r : real → real, x |⇒ 3.1414 : real ]
-      val f = it;
>      val f = fn : real → real
env = [ it |⇒ fn r => 3.1414 * r * r : real → real, x |⇒ 3.1414 : real, f |⇒ fn r => 3.1414 * r * r : real → real ]
-      (fn r => 3.14 * r * r) (2.0) ;
>      val it = 12.56636 : real
env = [ x |⇒ 3.1414 : real, f |⇒ fn r => 3.1414 * r * r : real → real , it |⇒ 12.56636 : real ]
-      val f = fn (p, q) => 2 + p * q ;
>      val f = fn : int * int -> int
env = [ x |⇒ 3.1414 : real , it |⇒ 12.56636 : real , f |⇒ fn (p, q) => 2 + p * q : int * int ]
-      f (3, 5);
>      val it = 17 : int
env = [ x |⇒ 3.1414 : real , f |⇒ fn (p, q) => 2 + p * q : int * int ]

```

Function declaration is only a derived form of a particular kind of value declaration. The following value declaration of function has the same effect as of normal function declaration shown earlier. It can be easily seen from the environment created in both types of declarations.

Examples:

```

-      val pi = 3.1414;
>      pi = 3.1414 : real
-      val circum = fn r => 2 * pi * r ;
>      val circum = fn : real → real
env = [ pi |⇒ 3.1414, circum |⇒ fn r => 2 * 3.1414 * r ]

```

Now onwards we will skip the environment for the sake of simplicity. There are different ways of defining functions given as follows:

Examples:

```

-      fun cube (x : real) = x * x * x ;
-      fun cube (x) : real = x * x * x ;
-      fun cube x : real = x * x * x ;
-      fun cube x = x * x * x : real ;

```

The diagram shows two labels on the right: "type of an argument" and "type of result". Arrows point from "type of an argument" to the parameter `x` in the first two definitions. Arrows point from "type of result" to the colon-separated type `: real` in the first two definitions, and to the trailing type `: real` in the last two definitions.

The system's output in all above cases is:

```

>      val cube = fn : real → real

```

Some SML systems assume basic arithmetic operators (+,-,*) as of *int* type by default but some may flash error if unable to resolve overloading. From the body of the function given below, it can not be derived whether * is used for integers or reals but as mentioned above it takes int type by default.

```

-      fun cube x = x * x * x;
>      val cube = fn : int → int

```

In the definitions of average given below, the types have been deduced automatically by seeing the types of the constituents in the bodies.

Examples:

```
- fun average1 (x, y) = (x + y) / 2.0 ;
> val average1 = fn : real * real → real
- fun average2 (x, y) = (x + y) div 2 ;
> val average2 = fn : int * int → int
```

Curried Function

A partially applicable function is called a *curried function* after the name of the logician H. B. Curry. A *partially applicable function* is a function that returns as a result a more specialized function when applied on the first argument. Hence any function of n ($n > 1$) arguments can be expressed as a curried function whose result is another function of $(n-1)$ arguments. Partially applicable functions are convenient to use and supports the search for powerful functions useful in variety of applications. The effect of such function is same when applied with the values of all arguments.

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

An arrow (\rightarrow) associates to the right so the type of average $real \rightarrow real \rightarrow real$ is interpreted as $real \rightarrow (real \rightarrow real)$. The function *average* takes a real argument and returns a function of the type $real \rightarrow real$ as a result.

Examples:

```
- val p = average 2.0;
> val p = fn : real → real
- p 6.4;
> val it = 4.2 : real
```

It is not necessary to introduce new name for the generated function. We can directly apply it as follows:

```
- (average 2.0) 6.4 ;
> val it = 4.2 : real
```

Since function application associates to the left, the parenthesis can be avoided and we can write *average 2.0 6.4* instead of *(average 2.0) 6.4*. Therefore, we can apply function with all the argument values without using brackets.

```
- average 2.0 6.4;
> val it = 4.2 : real
```

Few more examples are given below:

Examples:

```
- fun small x y = if x < y then x else y;
> val small = fn : int → int → int
- fun min x y z = small x (small y z);
> val min = fn : int → int → int → int
- val x = min 34 23 67;
> val x = 23 : int
- val x = min (34,23,67); ← wrong
```

Tuples

The type $\alpha * \beta$, where α and β are of any type, is the type of *ordered pair* whose first component is of type α and second component has type β . An ordered pair is written as (e_1, e_2) where e_1 and

e_2 are expressions of any type. Similarly we can define n-tuple (e_1, \dots, e_n) , where each expression e_i , $(1 \leq i \leq n)$ is of type α_i , $(1 \leq i \leq n)$ and each expression is separated by comma. The type of n-tuple is of the form $\alpha_1 * \dots * \alpha_n$.

Examples:

```
- val x = (2.3, true) ;
> val x = (2.3, true) : real * bool
- val y = (25 div 3, true, 2.34 + 1.2) ;      {div is integer division}
> val y = (8, true, 3.54) : int * bool * real
- val z = (25 div 3, (true, 2.34 + 1.2)) ;
> val z = (8, (true, 3.54)) : int * (bool * real)
- val p = (x, y, z);
> val p = ((2.3, true), (8, true, 3.54), (8, (true, 3.54)))
      : (real * bool) * (int * bool * real) * (int * (bool * real))
```

Polymorphic Function Declarations

Sometimes we see that type of the function is not deducible seeing the arguments or the body at the time of defining function. The arguments can be of any type say α and β . The actual type would be decided at the time of applying function. Such a type is called *polytype* and function using polytype is called *polymorphic function*. Such functions can be applied to arguments of any type.

Examples:

```
- fun tuple_self x = (x, x, x) ;
> val tuple_self = fn :  $\alpha \rightarrow \alpha * \alpha * \alpha$ 
- val p = tuple_self 25;
> val p = (25, 25, 25) : int * int * int
- val p2 = tuple_self true;
> val p2 = (true, true, true) : bool * bool * bool
- val p1 = tuple_self ("hi", 2);
> val p1 = (("hi", 2), ("hi", 2), ("hi", 2))
      : (string * int) * (string * int) * (string * int)
- fun pair (x, y) = (x, y);
> val pair = fn :  $\alpha * \beta \rightarrow \alpha * \beta$ 
- pair ((12, "test", 23.4), true);
> val it = ((12, "test", 23.4), true)
      : (int * string * real) * bool
- fun first_of_pair (x, y) = x ;
> val first_of_pair = fn :  $\alpha * \beta \rightarrow \alpha$ 
- fun second_of_pair (x, y) = y ;
> val second_of_pair = fn :  $\alpha * \beta \rightarrow \beta$ 
- val f = first_of_pair p1;
> val f = ("hi", 2) : string * int
- val s = second_of_pair f ;
> val s = 2 : int
- val f1 = second_of_pair (first_of_pair p1);
> val f1 = 2 : int
- val f1 = second_of_pair (first_of_pair p2);
> Error: invalid
- fun fstfst x = first_of_pair (first_of_pair x) ;
> val fstfst = fn :  $(\alpha * \beta) * \tau \rightarrow \alpha$ 
- fstfst p1;
> val it = "hi" : string
```

Here in the definition of a function *fstfst*, the inner function *first_of_pair* has a type $(\alpha * \beta) * \tau \rightarrow \alpha * \beta$ whereas outer function *first_of_pair* has a type $\alpha * \beta \rightarrow \alpha$. A polymorphic function can have different types within the same expression. In SML, the polymorphic type begins with a single quote followed by a character such as 'a', 'b' etc.

Equality test in Polymorphic Functions

Test for *equality* (=) and *inequality* (<>) are restricted form of polymorphism. Since equality type is forbidden on real numbers, function type and abstract type (explained later), equality operator can't be used on real numbers and we can not test if two functions are equal. The type variables with two primes 'a', 'b' etc. are used to denote all those types which allow test for equality and inequality. A function's type contains equality type variables if it performs polymorphic equality test directly or indirectly. Consider few examples to illustrate the use of equality test.

Examples:

```
- fun equal (x, y) = if x = y then true else false ;
> val equal = fn : 'a * 'a -> bool
- equal (23, 33);
> val it = false : bool
- equal ((1,2), (1,2));
> val it = true : bool
- equal (2.3, 2.3);
> Error: operator and operand don't agree
- fun notequal (x, y) = x <> y orelse false;
> val notequal = fn : 'a * 'a -> bool
```

Patterns

A *pattern* is an expression consisting of variables, constructors and wildcards. The *constructors* comprises of *constants* (integer, character, bool and string), *tuples*, record formation, datatype constructors (explained later) etc. The simplest form of pattern matching is *pattern = exp*, where *exp* is an expression. When *pattern = exp* is evaluated, it gives true or false value depending upon whether pattern matches with an expression or not.

Examples:

```
- true = ( 2 < 3);
> val it = true : bool
- 23 = 10 + 14;
> val it = false : bool
- "abcdefg" = "abc" ^ "defg";
> val it = true : bool
- #'a" = # "c";
> val it = false : bool
- (23, true) = (10+13, 2<3);
> val it = true : bool
- val v = 3;
> val v = 3 : int
- v = 2 + 1;
> val it = true : bool
```

The wildcard pattern can match to any data object. It is represented by underscore (`_`) and has no name thus returns an empty environment. Use of wildcard (don't care entry) will be explained later. In SML, the pattern matching occurs in several contexts.

- Pattern in value declaration. It is of the form `val pat = exp`. If pattern is a simple variable, then it is same as value declaration.
- If patterns are Pairs, tuples, record structure, then they may be decomposed into their constituent parts using pattern matching. The result of matching changes the environment. We go through a process of reduction to atomic value bindings, where an atomic bindings is one whose pattern is a variable pattern. The binding `val (pat1, ..., patn) = (val1, ..., valn)` reduces to


```

      >   val pat1 = val1
           ⋮
      >   val patn = valn
      
```
- This decomposition is repeated until all bindings are atomic.

Examples:

```

-   val ((p1, p2), (p3, p4 , p5)) = ((1,2), (3.4, "testing", true));
>   val p1 = 1 : int
>   val p2 = 2 : int
>   val p3 = 3.4 : real
>   val p4 = "testing" : string
>   val p5 = true : bool
-   val (p1, p2, _, p4) = (12, 3.4, true, 67);
                               ↙ wildcard
>   val p1 = 12 : int
>   val p2 = 3.4 : real
>   val p4 = 67 : int
  
```

Alternative Pattern

Functions can be defined using alternative patterns as follows:

$$\text{fun } pat_1 = exp_1 \mid pat_2 = exp_2 \mid \dots \mid pat_n = exp_n;$$

Each pattern pat_k consists of same function name followed by arguments. The patterns are matched from top to bottom until the match is found. The corresponding expression is evaluated and the value is returned.

Examples:

```

-   fun fact 1 = 1
      | fact n = n * fact (n-1);
>   val fact = fn : int -> int
-   fact 3;
>   val it = 6 : int
-   fun negation true = false
      | negation false = true;
>   val negation = fn : bool -> bool
-   negation (2 > 3);
>   val it = true : bool
  
```

Case Expression

Case expression is a mechanism for pattern matching. Conditional expression takes care of only two cases whereas if we want to express more than two cases, then the nested *if-then-else* expression is used. Alternatively we can handle such situations using case expression. The general form of case expression is:

```

case exp of  pat1 => exp1
            pat2 => exp2
            ⋮
            patn => expn

```

The value of *exp* is matched successively against the patterns *pat₁*, *pat₂*, ..., *pat_n*. If *pat_j* is the first pattern matched, then the corresponding *exp_j* is the value of the entire case expression. For example the nested nested if-then-else expression

if x = 0 then "zero" else if x = 1 then "one" else if x = 2 then "two" else "none"

is equivalent to the following case expression

```

case x of 0 => "zero"
        | 1 => "one"
        | 2 => "two"
        | _ => "none"
        ↙ wild card

```

Consider another example. Given a month number from (1 - 12), corresponding string stating month name is returned as a result of the case expression.

Example:

```

- fun convert month = case month of
    1 => "jan" | 2 => "feb" | 3 => "mar"
    | 4 => "apr" | 5 => "may" | 6 => "jun"
    | 7 => "july" | 8 => "aug" | 9 => "sept"
    | 10 => "oct" | 11 => "nov" | 12 => "dec"
    | _ => "none";
> val convert = fn : int -> string
- convert(12);
> val it = "dec" : string
- convert(13);
> val it = "none" : string

```

It should be noted that an expression *exp* is of enumerated type. Enumerated types are: int, bool, char and user defined datatype (explained later).

Infix Operators

An *infix operator* is a function that is written between two arguments. Most of the functional languages allow programmer to define their own infix operators. Let us define a boolean function implication *imply* as follows:

Example:

```

- fun imply(x, y) = not x orelse y;
> val imply = fn : bool * bool -> bool
- val q = imply ( true, false);
> val q = false : bool
- val p = imply (true, true);

```



```
> val p = true : bool
```

It is more convenient if we write infix operator between two arguments. SML provides directive called *infix* that helps user to define infix operator. Operator might be simple name or symbol.

- Let us define boolean implication function denoted by symbol \rightarrow as an infix operator

Examples:

```
- infix  $\rightarrow$ ;  
> infix  $\rightarrow$   
- fun x  $\rightarrow$  y = not x orelse y;  
> val  $\rightarrow$  = fn : bool * bool -> bool  
- true  $\rightarrow$  true;  
> val it = true : bool  
- true  $\rightarrow$  false;  
> val it = false : bool  
- false  $\rightarrow$  true;  
> val it = true : bool  
- false  $\rightarrow$  false;  
> val it = true : bool
```

- Boolean AND operator denoted by

Example:

```
- infix &;  
> infix &  
- fun true & b = b  
  | false & b = false;  
> val & = fn : bool * bool -> bool  
- val x = true & 3 < 5;  
> val x = true : bool
```

- Boolean OR operator denoted by V

Example:

```
- infix V;  
> infix V  
- fun true V b = true  
  | false V b = b;  
> val V = fn : bool * bool -> bool  
- 3 > 5 V 4 > 3;  
> val it = true : bool
```

Precedence of infix operator may be specified between 0 to 9. Default precedence is 0. The directive *infix* causes left association { $a + b + c$ means $(a + b) + c$ } whereas *infixr* causes right association (operator *exponentiation* is right associative).

Records

A *record* of n components is a n -tuple whose components are identified by *labels*. It is different from n -tuple as each component of n -tuple, is identified by its position from 1 to n . The components of a record are called **fields**. The general forms are given below:

Record type

```
{label1 = type1, ..., labeln = typen}
```

e.g., $\{name : string, age : int, adult : bool\}$
Record value
 $\{label_1 = exp_1, label_2 = exp_2, \dots, label_n = exp_n\}$
e.g., $\{name = "john", age = 35, adult = true\};$
Record pattern
 $\{label_1 = pat_1, \dots, label_n = pat_n\} : \{label_1 = type_1, \dots, label_n = type_n\},$
where, each pat_k has a type $type_k$ for $1 \leq k \leq n$.
e.g., $\{name="john", age=35, adult=true\} : \{name : string, age : int, adult : bool\}$

SML system arranges labels of a record in alphabetical order.

Examples:

```
-      val x = {name = "john", age = 35, adult = true};
>      val x = {adult=true, age=35, name = "john"}
           : {adult : bool, age : int, name : string}

-      val p = 3;
>      val p = 3 : int

-      val q = 4.5;
>      val q = 4.5 : real

-      val t = { c = "record", a = p + 5, b = q };
>      val t = {a = 8, b = 4.5, c = "record"} : {a : int, b : real, c : string}
```

Accessing values of labels

The general form of accessing a particular *label value* from a record is:

$\# label_name \ rec_name$

The field $label = identifier$ gives each *identifier* the value of the corresponding *label*.

Examples:

```
-      val r = {a = 3, p = 2} ;
>      val r = {a = 3, p = 2} : {a : int, p : int}

-      val x1 = #a r and x2 = #p r ;
>      val x1 = 3 : int
>      val x2 = 2 : int

-      val { a = a1, p = p1 } = r;
>      val a1 = 3 : int
>      val p1 = 2 : int

-      val {a = a1, p = p1} = { p = "trial", a =45.3};
>      val a1 = 45.3 : real
>      val p1 = "trial" : string
```

If we need the values of only few fields, then put three dots after listing the required fields.

Examples:

```
-      val s = {x = 3, a = 23.5, z = true};
>      val s = {a = 23.5, x = 3, z = true} : {a : real, x : int, z : bool}

-      val {a = a1, z = z1, ...} = s;
>      val a1 = 23.5 : real
>      val z1 = true : bool
```

We can also use label names directly instead of identifiers.

Examples:

```
-      val {a, x, ...} = s;
```

```
> val a = 23.5 : real
> val x = 3 : int
```

Expressing tuple as a record

An n-tuple (x_1, x_2, \dots, x_n) is just an abbreviation for a record with numbered fields as $\{ 1 = x_1, 2 = x_2, \dots, n = x_n \}$.

Examples:

```
- val r = { 1 = 23, 2 = 3.4 3 = false };
> val r = (23, 3.4, false) : int * real * bool
- # 2 r;
> val it = 3.4 : real
```

Records can be compared for equality or inequality.

Examples:

```
- { name = "shiva", age = 13 } = { name = "jim", age = 45 };
> val it = false : bool
- { name = "shiva", age = 13 } <> { name = "jim", age = 45 };
```

Local declarations

In some programs, we require auxiliary declarations, which are valid locally. Hiding declarations while writing big programs is useful facility. Locally declared variables or functions can have the same names which are globally used without any problem as local declarations are not visible outside its scope. In SML there are two types of local declarations. The general forms of local declarations are given as follows:

Let expression : (declarations local to an expression)

let $D_1; D_2; \dots; D_n$ in E end

Let expression is evaluated as follows:

- Declaration D_1 is visible in D_2 , D_1 & D_2 are visible in D_3 and subsequently D_1, \dots, D_{n-1} are visible in D_n . All the declarations are visible in an expression E. E is called the *scope* of all declarations.
- Environment *env* before executing *let expression* gets updated by each let declaration as $env_1; env_2; \dots; env_n$, each environment env_j contains previous environments env_1, \dots, env_{j-1} . n expression E is evaluated in the last environment env_n and the value of expression is added to original environment *env* by undoing the environments obtained because of *let* declarations.

Examples:

```
- let val x = 3; val y = 4 in x + y end;
> val it = 7 : int
- val x = 5;
> val x = 5 : int
- val p = let val x = 3 in x * 2 end; { x is local }
> val p = 6 : int
```

```

-      p + x;                                {x is global & its value is 5}
>      val it = 11 : int
-      val q = let val x = 3 and y = 6; val z = 12 in x + y - z end;
>      val q = ~3 : int
-      fun area_triangle (x, y, z) = let val s = (x + y + z) / 2.0 in
                                         sqrt(s * (s - x) * (s - y) * (s - z)) end;
>      val area_triangle = fn : real * real * real -> real
-      val p = s;
>      Error: s is not defined

```

Local declarations: (declarations local to other declarations)

local $D_1; D_2; \dots; D_n$ in $D'_1; D'_2; \dots; D'_m$ end

Local declaration is executed as follows:

- Declaration D_1 is visible in D_2 , D_1, D_2 in D_3 and subsequently D_1, \dots, D_{n-1} in D_n and all are visible in the declarations D_1' . Further, $D_1, \dots, D_{n-1}, D_1'$ in D_2' and so on in D_m' .
- New environments are created in the sequence $env_1, env_2, \dots, env_n, env_1', env_2', \dots$ and env_m' each one including the previous environments.
- After executing local declaration, the resulting environment env' is obtained after removing the bindings obtained from the declarations D_1, D_2, \dots and D_n .

Examples:

```

-      local val x = 3; val y = 4 in val p = x * y end;
>      val p = 12 : int
-      local val x = 3; val y = x - 2 in val z = x * y; val w = z - y end;
>      val z = 3 : int
>      val w = 2 : int
-      local fun divide(x, y) = x mod y = 0 in
                                         fun leap (year) = divide(year, 4) ;
                                         fun millenium (year) = divide(year, 1000) end;
>      val leap = fn : int -> bool
>      val millenium = fn : int -> bool
-      val p = leap (1900);
>      val p = true : bool
-      leap(1998);
>      val it = false : bool
-      millenium(1000);
>      val it = true : bool
-      val q = millenium(1900);
>      val q = false : bool

```

Distinction between let expression and local declaration

Let is used most frequently as compared to local declaration but problem can be solved using both. Few examples are given below:

Examples:

```

-      val p = 4;
>      val p = 4 : int
-      val q = 3;

```

```

> val q = 3 : int
- val y = let val x = p + q in x * x end;
> val y = 16 : int
- local val x = p + q in val y = p * p end;
> val y = 16 : int

```

← same effect

Example: Write a function to solve quadratic equations $ax^2 + bx + c = 0$.

Using let:

```

- fun q_solve(a, b, c) =
    let val t = b * b - 4.0 * a * c in
      if t > 0.0 andalso a > 0.0 then
        let val t1 = Math.sqrt (t);
            val t2 = 2.0 * a;
            val t3 = ~ b
        in ((t3 + t1) / t2, (t3 - t1) / t2) end
      else raise Invalid
    end;
> val q_solve = fn : real * real * real -> real * real
- val root = q_solve(1.0, ~8.0, 15.0);
> val root = (5.0,3.0) : real * real
- val root = q_solve(1.0, 4.0, ~21.0);
> val root = (3.0,~7.0) : real * real
- q_solve(0.0, 2.3, 3.4);
> exception Invalid raised
- q_solve(12.0, 2.3, 3.4);
> exception Invalid raised

```

← exception name explained later

Using Local:

```

- local
    fun discriminant (a, b, c) = b * b - 4.0 * a * c ;
    fun valid (a, b, c) =
        discriminant (a, b, c) > 0.0 andalso a > 0.0;
    fun compute(a, b, c) = Math.sqrt (discriminant (a, b, c));
    fun double_a (a) = 2.0 * a
  in fun q_solve(a, b, c) = if valid (a, b, c) then
    ((~ b + compute(a, b, c)/double_a (a)), (~ b - compute(a, b, c)/double_a (a)))
    else raise Invalid
  end;
> q_solve = fn : real * real * real -> real * real
- val root = q_solve(0.0, 1.2, 3.2);
> exception Invalid raised
- val root = q_solve(23.5, 1.5, 3.4);
> exception Invalid raised
- val root = q_solve(1.0, 23.6, ~2.5);
> val root = (0.105460931858,~23.7054609319) : real * real
- val root = q_solve(1.0, ~8.0, 15.0);
> val root = (5.0,3.0) : real * real
- val root = q_solve(1.0, 4.0, ~21.0);
> val root = (3.0,~7.0) : real * real

```

We see from the above example that local declaration is not convenient way of writing function as compared to function written using let expression.

- It is more straight forward and easy to understand using let expression.
- The function is more efficient computationally using let expression.
- Repetitive computations can be easily avoided using let expression

Therefore, let expression is frequently used as compared to local declaration.

SML provides library of mathematical functions in a structure called *Math*. Declarations are grouped to form a *structure* by enclosing them in the keywords **struct** and **end**. The structures are discussed in the next chapter in detail. An access to a functions definition defined in a structure is

`structure_name . function_name (actual arguments)`

The function *sqrt* defined in **Math** structure is accessed as **Math.sqrt**.

Type Declaration

In SML, the *types* are built from predefined types viz., int, real, char, string, bool and unit. We are familiar with all types except unit. The type *unit* consists of a single value, denoted by (). This type is used whenever an expression has no interesting value or function does not have arguments. The *type declaration* is defined in the same way as the data object is named in variable declaration. The general form of type declaration is:

`type var1 = type1 and var2 = type2 and ... and varn = typen`

where each *var_k* is a type variable and *type_k* is a type expression.

The type declaration can also be used to define alternative type for existing types. New as well as old types are available for use. Let us see the declarations made in the following examples:

Examples:

```
-      type integer = int;
>      type integer = int
-      type float = real and boolean = bool and character = char;
>      type float = real
>      type boolean = bool
>      type character = char
-      type pair = integer * boolean;
>      type pair = integer * boolean
-      (23, true) : pair;
>      val it = (23,true) : pair
-      type person_rec = {name:string, age:float, address : string, salary:real};
>      type person_rec = {address:string,age:float,name:string,salary: real}
-      val p={name="john",age=67,address="New Delhi", salary=2000.00};
>      val p={address="NewDelhi",age=67,name="john", salary=2000.0}
           : {address : string, age : real, name : string, salary:real}
```

Example: Function for getting the name of a person with age > 60.

```
-      fun get_name(x : person_rec) =
```

```

        if # age x > 60.0 then #name x else "sorry";
>     val get_name = fn : person_rec -> string
-     get_name(p);
>     val it = "john" : string

```

Datatype Declaration

Datatype declaration provides a means to introduce a completely new user defined *type constructors* in the program along with one or more *value constructors* for the type constructor.

- The *type constructor* may take zero or more arguments.
- Each *value constructor* may also take zero or more arguments.
- The type and value constructors are new and distinct from all other previously or system defined types.

Here we discuss *nullary* type constructor which means a type constructor with zero argument, *Nullary* value constructor is a value constructor with zero argument. Such types are also called *enumerated types* where value constructors are enumerated. Advanced type constructors will be discussed in the next chapter. The general form of datatype definition is as follows:

```

datatype type_constructor_name
    = value_constructor1 | value_constructor2 | ... | value_constructorn

```

We use the following conventions. These are purely user defined conventions.

- Name of *type constructor* is formed using upper case letters.
- Name of *value constructor* starts with upper case letter followed by lowercase letters.

Enumerated types (Nullary Type Constructor with Nullary Value Constructors):

Consider datatype for type constructor DAY that defines value constructors Mon, Tue, Wed, Thr, Fri, Sat and Sun with zero argument. Such value constructors are called nullary value constructors.

Examples:

```

-     datatype DAY = Mon | Tue | Wed | Thr | Fri | Sat | Sun;
>     datatype DAY = Fri | Mon | Sat | Sun | Thr | Tue | Wed
-     fun working_day (day :DAY) = day <> Sat andalso day <> Sun;
>     val working_day = fn : DAY -> bool
-     val d = working_day(Tue);
>     val d = true : bool
-     datatype MONTH =      Jan | Feb | Mar | Apr | May | Jun | July | Aug
                           | Sept | Oct | Nov | Dec ;
>     datatype MONTH= Apr | Aug | Dec | Feb | Jan | July | Jun
                           | Mar | May | Nov | Oct | Sept
-     fun month_days (month, year) = let
                                     fun leap (year) = year mod 4 = 0;
                                     val x = if leap(year) then 1 else 0
                                     in
                                     case month of
                                     Jan => 31 | Feb =>28 + x
                                     | Mar => 31 | Apr => 30
                                     | May =>31 | Jun =>30
                                     | July =>31 | Aug => 31

```

```

| Sept => 30 | Oct =>31
| Nov =>30 | Dec =>31
end ;
> val month_days = fn : MONTH * int -> int
- month_days(Mar, 1998);
> val it = 31 : int
- month_days(Feb, 1999);
> val it = 28 : int
- month_days(Feb, 2000);
> val it = 29 : int

```

Let us consider another example of user defined datatype named as RELATION with nullary value constructors as Less, Equal and Greater for handling real comparisons. The function grade defined below makes use of the newly defined value constructors in case expression.

Examples:

```

- datatype RELATION = Less | Equal | Greater;
> datatype RELATION = Equal | Greater | Less
- fun compare(x, y :real) : RELATION = if x < y then Less
                                     else if x > y then Greater else Equal;
> val compare = fn : real * real -> RELATION
- fun grade (marks) = case compare(marks, 80.0) of
    Greater => "A"
  | Equal => "A"
  | Less => case compare(marks, 60.0) of
    Greater => "B"
  | Equal => "B"
  | Less => case compare(marks,50.0) of
    Greater => "C"
  | Equal => "C"
  | Less => case compare(marks, 40.0) of
    Greater => "D"
  | Equal => "D"
  | Less => "F" ;
> val grade = fn : real -> string
- grade(98.5);
> val it = "A" : string
- grade(34.5);
> val it = "F" : string
- grade(70.0);
> val it = "B" : string

```

Nullary type constructor with value constructors with more than zero arguments:

Consider datatype for type constructor SHAPE that defines value constructor with more than zero arguments. These value constructors are all functions giving the values of type SHAPE when applied to the argument(s).

Examples:

```

- datatype SHAPE = Circle of real | Rectangle of real * real
                | Square of real ;
> datatype SHAPE
    Circle = fn: real → SHAPE
    Rectangle = fn : real * real → SHAPE
    Square =fn : real → SHAPE

```



```

-   Circle 3.5;
>   val it = Circle 3.5 : SHAPE
-   val s = Square 1.2;
>   val s = Square 1.2 : SHAPE

```

Let us write function for calculating area of a geometrical figures mentioned above.

Examples:

```

-   fun area (s: SHAPE) = case s of
                               Circle r => 3.14 * r * r
                              | Rectangle (x, y) => x * y
                              | Square side => side * side
                              | _ => 0.0;
>   val area = fn : SHAPE -> real

```

Last pattern in the case expression is underscore to take care of values for undefined shapes. At this point exception can be raised. We will discuss exception in the next chapter. Let us call these functions.

Examples:

```

-   area (Circle 1.2);
>   val it = 4.5216 : real
-   area (Rectangle (2.3, 4.5));
>   val it = 10.35 : real
-   area (Square 4.5);
>   val it = 20.25 : real

```

List and Advanced Features in SML

List is an ordered sequence of data objects, all of which are of the same type. In SML, the list consists of finite sequence of values of type α and the entire list is of type α list. The elements of list are enclosed in square brackets and are separated by comma. The empty list denoted by [] that contains no elements. The order of elements is significant. List can contain varying number of elements of the same type whereas in tuples and records the number of elements are fixed and are of any type. The first element of the list is at 0 position. Typical lists are:

Examples:

```

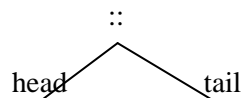
- [2, 3, 4];
> val it = [2,3,4] : int list
- ["john", "mary", "jack"];
> val it = ["john","mary","jack"] : string list
- [(2, true), (4, false)] etc.
> val it = [(2,true),(4,false)] : (int * bool) list
- val x = [];
> val x = [] : 'a list
- val r = [2.3, 4.5 / 1.2, 8.9 + 2.3];
> val r = [2.3,3.75,11.2] : real list
- val y = [[1,2], [3,4,5,6], [ ]];
> val y = [[1,2],[3,4,5,6],[ ]] : int list list
- val p = [floor, round, trunc];
> val p = [fn,fn,fn] : (real -> int) list

```

Constructing a List

A list is constructed by two primitives: one a constant *nil* (empty list denoted by []) and other an infix operator *cons* represented by ::. A list is represented as *head :: tail*, where *head* is a first element of the list and *tail* is the remaining list. The operator *cons* builds a tree for a list from its head to tail. For example, a list [2] can be represented as 2 :: nil.

The tree representation for a list *head :: tail* is as follows:



A list can be constructed by adding an element in the beginning using *cons* operator. Few examples for constructing a list are given below:

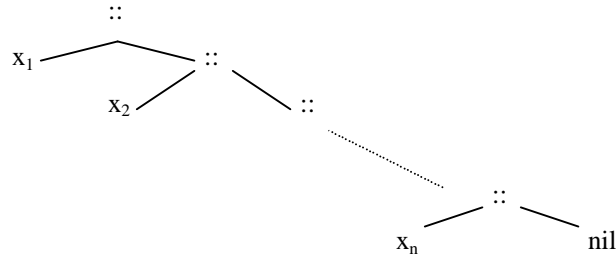
Examples:

```

- 4::nil;
> val it = [4] : int list
- val p = 3 :: [4];
> val p = [3,4] : int list
- val q = 2 :: p;
> val q = [2,3,4] : int list

```

A list $[x_1, x_2, \dots, x_n]$ can also be written as $x_1 :: \dots :: x_n :: \text{nil}$. It's tree representation is



Two lists can be compared for equality or inequality. They are equal if are of same size, type and all corresponding elements are equal.

Examples:

```
- [1,2,3] = 1 :: 2 :: 3 :: nil;
> val it = true : bool
- val p = [2, 3, 4] <> [1+1, 4-1, 2+2];
> val p = false : bool
```

Standard Functions for List handling in SML

There are only few standard functions for handling lists in SML

- *List constructor operator: (denoted by ::)*

```
- 2 :: [3,4,5];
> val it = [2,3,4,5] : int list
- true :: [2>3, false];
> val it = [true, false, false] : bool list
- (1,2) :: [(3,4), (5,6)];
> val it = [(1,2), (3,4),(5,6)] : (int * int) list
```

- *Append operator :(denoted by @)*

```
- [1,2,3] @ [4,5];
> val it = [1,2,3,4,5] : int list
- [[1,2], [3,4], [5]] @ [[7,8],[9]];
> val it = [[1,2],[3,4],[5],[7,8],[9]] : int list list
- [(1,"ab"), (2, "bc")] @ [(5, "we")];
> val it = [(1,"ab"),(2,"bc"),(5,"we")] : (int * string) list
```

- *Function for reversing the elements of a list: (rev)*

```
- rev [1,2,3];
> val it = [3,2,1] : int list
- rev [[1,2], [3,4], [5]];
> val it = [[5],[3,4],[1,2]] : int list list
- rev [(10,"abc"), (20, "bcd")];
> val it = [(20,"bcd"),(10,"abc")] : (int * string) list
```

- *Functions for finding Head and Tail of a list :(hd - for head and tl - for tail)*

```

-      hd [(1,"ab"), (2, "bc")];
>      val it = (1,"ab") : int * string
-      hd [[1,2], [3,4], [5]];
>      val it = [1,2] : int list
-      tl [(1,"ab"), (2, "bc")];
>      val it = [(2,"bc")] : (int * string) list
-      tl [[1,2], [3,4], [5]];
>      val it = [[3,4],[5]] : int list list

```

- *Finding the length of a list: (length)*

```

-      length [1,2,3,6];
>      val it = 4 : int
-      length [(1,"ab"), (2, "bc")];
>      val it = 2 : int
-      length [[1,2], [3,4], [5]];
>      val it = 3 : int

```

- *Conversion between strings and list of characters : (explode and implode)*

```

-      val x = explode "abcdefg";
>      val x = [#"a",#"b",#"c",#"d",#"e",#"f",#"g"] : char list
-      implode x;
>      val it = "abcdefg" : string

```

Recursive functions for List

Recursive function can be easily written for lists using induction principle of natural numbers. Empty list corresponds to zero and constructor operator `::` corresponds to successor function on natural numbers. Recursion on list is defined as follows:

- Define a function for base case.
- Assumption that function computes correct result for a list 'xs'. Find how to compute the result for a list `x :: xs`.

The skeleton of recursive function for a list is as follows:

```

-      fun    recursive_fun []                =      ?
              | recursive_fun (x :: xs)      =      ?

```

We will see variations of the basic scheme but the idea remains same. If we have to write function with several arguments, then the following definition can be observed.

```

-      fun    multiple_arg_fun ([], ys)      =      ?
              | multiple_arg_fun (x :: xs, ys) =      ?

```

Some commonly used recursive functions for handling lists are defined below:

List Generations

- *Create a list of elements from n to m, where n and m are integers and $m \geq n$.*

```

-      fun create_list (n, m) = if n>m then [] else n::create_list (n+1, m);
>      val create_list = fn : int * int -> int list
-      create_list (2, 7);
>      val it = [2,3,4,5,6,7] : int list
-      create_list (~2, 5);
>      val it = [-2,~1,0,1,2,3,4,5] : int list

```

- Create a list of elements from 1 to n, $n \geq 1$

```

- fun oneto 1 = [1]
  | oneto n = oneto (n-1) @ [n];
> val oneto = fn : int -> int list
- oneto 6;
> val it = [1,2,3,4,5,6] : int list

```

Alternatively,

```

- fun oneto (n) = create_list(1,n); {call another function create_list }

```

- Create a list of even numbers starting from 2 for a given size.

```

- fun gen 1 = [2]
  | gen n = gen (n-1) @ [2*n];
> val gen = fn : int -> int list
- gen 4;
> val it = [2,4,6,8] : int list
- gen 6;
> val it = [2,4,6,8,10,12] : int list

```

Various other important list functions

- Adding all the elements of a list

```

- fun add [] = 0
  | add (x::xs) = x + add xs;
> val add = fn : int list -> int
- add [2,3,4,7,8];
> val it = 24 : int

```

- Multiplying all the elements of a list

```

- fun mult [] = 0
  | mult [x] = x
  | mult (x::xs) = x * mult xs ;
> val multl = fn : int list -> int
- mult [2,3,4];
> val it = 24 : int

```

- Merging two integer lists in increasing order assuming original lists are in increasing order.

```

- fun merge ([], ys:int list) = ys
  | merge (xs, []) = xs
  | merge ((x::xs), (y::ys)) = if x < y then
                                (x::(merge (xs, (y::ys))))
                                else (y::(merge ((x::xs), ys)));
> val merge = fn : int list * int list -> int list
- merge ([2,4,6], [1,2,5,7,9]);
> val it = [1,2,2,4,5,6,7,9] : int list

```

- Selecting a particular position value from a list

```

- fun select (n, []) = 0
  | select (n, (x::xs)) = if n = 1 then x else select ((n-1), xs);
> val select = fn : int * int list -> int
- select (2, [3, 4, 5]);
> val it = 4 : int

```

- *Finding the maximum value of the elements in a list*

```

- fun max [] = 0
  | max [x] = x
  | max (x::y::xs) = if x > y then max (x::xs)
                    else max (y::xs);
> val max = fn : int list -> int
- max [3,9,1,3,56,7];
> val it = 56 : int

```

Functions with Polymorphic Equality Type

- *Check whether an element is a member of a list or not*

```

- fun mem (x, []) = false
  | mem (x, (y::ys)) = if x = y then true else mem (x, ys);
> val mem = fn : 'a * 'a list -> bool
- mem (3, [1,2,3,4]);
> val it = true : bool
- mem ((2, true), [(1, false), (2, false)]);
> val it = false : bool
- mem ({name = "asd", age = 23},
      [{name = "asd", age = 23}, {name = "VVBB", age = 45}]);
> val it = true : bool

```

- *Delete an element from a given list*

```

- fun delete (x, []) = []
  | delete (x, (y::ys)) = if x = y then delete (x, ys)
                        else (y::delete (x, ys));
> val delete = fn : 'a * 'a list -> 'a list
- delete (3, [1,2,3,4,3,3]);
> val it = [1,2,4] : int list
- delete (3, [5,6,7,8]);
> val it = [5,6,7,8] : int list

```

- *Whether two lists are disjoint or not*

```

- fun disjoint ([], x) = true
  | disjoint ((x::xs), ys) = if (mem (x, ys)) then false
                            else disjoint (xs, ys);
> val disjoint = fn : 'a list * 'a list -> bool
- disjoint ([2,3], [5,6]);
> val it = true : bool
- disjoint ([{name = "hi", age = 5}], [{name = "hi", age = 5}]);
> val it = false : bool

```

- *Find the common elements of two lists*

```

- fun common ( [], ys) = []
  | common ( xs, [] ) = []
  | common ((x::xs), ys) = if mem (x, ys) then
                           (x:: common (xs, ys))
                           else common (xs, ys);
> val common = fn : 'a list * 'a list -> 'a list
- common ([2,3], [1,2,3]);
> val it = [2,3] : int list
- common ([(1,2), (3,4)], [(2,3), (3,4), (5,6)]);
> val it = [(3,4)] : (int * int) list

```

- Compare two lists for equality: (*equal*)

```

- infix equal;
> infix equal
- fun ( [] equal [] ) = true
  | ((a::x) equal (b::y)) = (a = b) andalso (x equal y)
  | ((a::x) equal []) = false
  | ([] equal (b::y)) = false;
> val equal = fn : 'a list * 'a list -> bool
- [(1,2), (3,4)] equal [(1,2), (4,3)];
> val it = false : bool

```

List equality (denoted by =) is available in SML.

Functions with polymorphic type

- Pairing corresponding elements of two lists of same size.

```

- fun pair ([],[]) = []
  | pair (x::n, y::m) = [(x,y)] @ pair (n, m);
> val pair = fn : 'a list * 'b list -> ('a * 'b) list
- pair ([2,3,4], ["ad", "rw", "j"]);
> val it = [(2,"ad"),(3,"rw"),(4,"j")] : (int * string) list
- pair ([(12,2.3), (34, 12.3)], [(23, 2.3), (21, 6.7)]);
> val it = [((12,2.3),(23,2.3)),((34,12.3),(21,6.7))]
           : ((int * real) * (int * real)) list

```

- Make identical copy of a given list

```

- fun copy [] = []
  | copy (x::xs) = x::copy xs;
> val copy = fn : 'a list -> 'a list
- copy [10.2,20.3,30.6,40.7];
> val it = [10.2,20.3,30.6,40.7] : real list

```

- Reverse a given list

```

- fun reverse [] = []
  | reverse (x::xs) = (reverse xs) @ [x];
> val reverse = fn : 'a list -> 'a list
- reverse [3,4,6,7];
> val it = [7,6,4,3] : int list
- reverse ["cat", "rat", "dog"];
> val it = ["dog","rat","cat"] : string list
- reverse [(2.3,14), (4.5,12)];
> val it = [(4.5,12),(2.3,14)] : (real * int) list

```

- *Appending two lists*

```

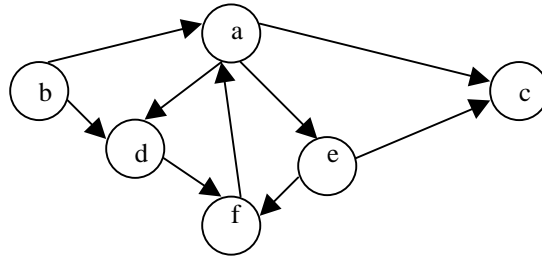
- fun    append ( [], ys)          =    ys
      | append ( x::xs, ys)       =    x:: append (xs, ys);
> val append = fn : 'a list * 'a list -> 'a list
- append ( [2,3,4] , [1,8,9,5,6] );
> val it = [2,3,4,1,8,9,5,6] : int list
- append ( [(1,2),(3,4)] , [(9,8),(7,6)] );
> val it = [(1,2),(3,4),(9,8),(7,6)] : (int * int) list

```

Graphs as an Application of a List

A *Graph* is defined to be collection of nodes and edges. A pair (x, y) can be used to represent edge from a node x to node y of a directed graph. A list of pairs is used to represent a directed graph.

Representation of Graphs in SML



```

- val graph_list = [ ("a", "d"), ("a", "e"), ("a", "c"), ("b", "a"), ("b", "d"),
                    ("d", "f"), ("e", "c"), ("e", "f"), ("f", "a") ];
> val graph_list = [ ("a","d"), ("a","e"), ("a","c"), ("b","a"), ("b","d"), ("d","f"),
                    ("e","c"), ("e","f"), ("f","a") ] : (string * string) list

```

- *Find the successor of a given node: (Nodes of graph have equality type)*

```

- fun    succ (x, [])              =    []
      | succ (x, (p, q) :: xs)    =    if x = p then q :: succ(x, xs)
                                      else succ (x, xs);
> val succ = fn : 'a * ('a * 'b) list -> 'b list
- succ ("a", graph_list); ← graph_list is a graph defined above
> val it = ["d", "e", "c"] : string list
- succ ("b", graph_list) @ succ ("e", graph_list);
> val it = ["a", "d", "c", "f"] : string list

```

Traversal of graph using depth first search (DFS) algorithm

In *depth first search* (dfs), the nodes of a graph not yet traversed reachable from the start node are fully explored and then backtrack to immediate predecessor node and search starts in dfs order until all nodes have been visited.

The function *df_search* defined below takes a list of all the nodes, graph and the list of nodes visited (initially empty) and returns a list of nodes reachable from the first node in the list.


```

-      fun      df_search ([], graph, visted)          = rev (visted)
          | df_search ( (x :: xs), graph, visted)      = if mem (x, visted)
                                                    then df_search (xs, graph, visted)
                                                    else
                                                    df_search (succ(x, graph) @ xs, graph, (x::visted));
>      val df_search = fn : "a list * ("a * "a) list * "a list -> "a list
-      df_search ([ "a", "b", "c", "d", "e", "f"], graph_list, []);
>      val it = [ "a", "d", "f", "e", "c", "b" ] : string list
-      df_search ( [ "c", "d", "e", "f", "b", "a"], g, []);
>      val it = [ "c", "d", "f", "a", "e", "b" ] : string list

```

Exercise: Find the path between two nodes of a graph.

Recursive Datatype Declarations

We have seen that the recursive functions are defined in terms of themselves. Similarly datatypes can also be defined recursively for specially those data structures which are recursive in nature. The natural numbers, lists, trees are few examples of recursive data structures. SML allows two kinds of recursive definition for datatype declarations.

- Ordinary recursive definition where datatype constructor is defined in terms of itself.
- Parameterized type constructor which may have polymorphic types and contains type variables specified in front of the type constructor in the datatype declaration.

Consider a datatype definition for natural numbers. Natural numbers are constructed by using two constructors *Zero* (nullary value constructor) and *Succ* of the type NATURAL -> NATURAL. This means that *Succ* is a function that takes argument of the type NATURAL and produces a value again of type NATURAL. This definition is recursive in nature and is non parameterized.

```

-      datatype NATURAL = Zero | Succ of NATURAL;
>      datatype NATURAL = Succ of NATURAL | Zero
-      Zero;
>      val it = Zero : NATURAL
-      Succ (Zero);
>      val it = Succ Zero : NATURAL
-      Succ (Succ (Zero));
>      val it = Succ (Succ Zero) : NATURAL

```

Polymorphic recursive datatype (Parameterized type constructor)

We have already seen earlier that a list is constructed using two constructors namely, :: (constructor) and [] (empty list). Let us define our own list that is of polymorphic type with two value constructors *Nil* of type 'a LIST and *Cons* of the type 'a * 'a LIST -> 'a LIST . Recursive datatype definition of a list is as follows:

```

-      datatype 'a LIST = Nil | Cons of 'a * 'a LIST;
>      datatype 'a LIST = Cons of 'a * 'a LIST | Nil
-      Cons;
>      val it = fn : 'a * 'a LIST -> 'a LIST
-      Nil;
>      val it = Nil : 'a LIST
-      Cons (2, Nil);
>      val it = Cons (2,Nil) : int LIST
-      Cons (3, it);
>      val it = Cons (3,Cons (2,Nil)) : int LIST
-      Cons (("pair", 3), Nil);
>      val it = Cons (("pair",3),Nil) : (string * int) LIST

```

Trees in SML

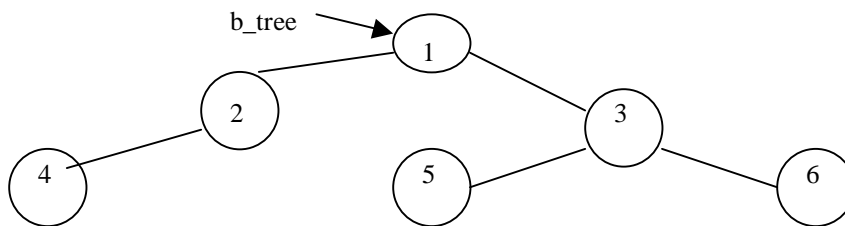
Tree is a nonlinear data structure consisting of nodes and branches to sub trees. The binary tree is a specialized tree with at most two branches. Each node of a binary tree consists of a data, left link pointing to left sub tree and right link pointing to sub tree. We define *recursive datatype* for a binary tree as follows:

```

-      datatype 'a BT = Null | Node of 'a BT * 'a * 'a BT;
>      datatype 'a BT = Null | Node of 'a BT * 'a * 'a BT

```

Consider the following binary tree.



According to the above definition of binary tree, it is represented as:

```

-      val b_tree = Node(Node(Node(Null, 4, Null), 2, Null), 1,
                          Node(Node(Null,5, Null), 3, Node(Null, 6, Null)));
>      val b_tree = Node(Node(Node #, 2, Null), 1, Node(Node #,3,Node #)) : int BT

```

Here # represents the entire corresponding sub tree. Complete tree is not displayed by the system. The node values of binary tree of height 2 are shown and remaining values are represented by #.

Binary Tree Traversal

Nodes of a binary tree can be visited in three ways: preorder (root, left sub tree in preorder, right sub tree in preorder), inorder (left sub tree in inorder, root, right sub tree in inorder), postorder (left sub tree in postorder, right sub tree in postorder, root). Let us define all these functions and apply them on a tree t defined above.

```

-      fun preorder      Null      = []
          | preorder (Node (left, data, right))= [data] @ preorder left
          @ preorder right;
>      val preorder = fn : 'a BT -> 'a list
-      preorder b_tree;

```

```

> val it = [1,2,4,3,5,6] : int list
- fun inorder Null = []
  | inorder (Node(left, data, right)) = inorder left @ [data] @
  inorder right;

> val inorder = fn : 'a BT -> 'a list
- inorder b_tree;
> val it = [4,2,1,5,3,6] : int list
- fun postorder Null = []
  | postorder (Node(left, data, right)) = postorder left @
  postorder right @ [data];

> val postorder = fn : 'a BT -> 'a list
- postorder b_tree;
> val it = [4,2,5,6,3,1] : int list

```

Useful functions for manipulating binary trees

- *find the depth (height) of a binary tree*

```

- fun depth Null = 0
  | depth (Node(left, data, right)) = 1 + let
    val l_depth = depth left;
    val r_depth = depth right;
  in if l_depth < r_depth then r_depth else l_depth end;
> val depth = fn : 'a BT -> int
- depth b_tree;
> val it = 3 : int

```

- *Count the number of nodes in a binary tree*

```

- fun count Null = 0
  | count (Node(left, data, right)) = 1 + (count left)
  + (count right);
> val count = fn : 'a BT -> int
- val num = count b_tree;
> val num = 6 : int

```

- *Create mirror image (swap) of a binary tree*

```

- fun swap Null = Null
  | swap (Node(left, data, right)) =
  Node (swap right, data, swap left);
> val swap = fn : 'a BT -> 'a BT
- val t1 = swap t;
> val b_tree1 = Node (Node (Node #, 3, Node #), 1,
  Node (Null, 2, Node #)) : int BT
- swap b_tree1 = b_tree; ← comparing two trees for equality
> val it = true : bool

```

- *Generate a full binary tree of depth n labeling the nodes from 1 to 2ⁿ - 1*

```

- fun full (k, 0) = Null
  | full (k, n) = Node (full (2*k, n-1), k,

```

```

full (2 * k + 1, n-1);
> val full = fn : int * int -> int BT
- fun full_tree (n) = full(1,n);
> val full_tree = fn : int -> int BT
- val t = full_tree (2);
> val t = Node (Node (Null,2,Null), 1, Node (Null,3,Null)) :int BT
- inorder(t);
> val it = [2,1,3] : int list
- val t1= full_tree(3);
> val t1= Node(Node (Node#,2,Node #),1,Node(Node #,3,Node #)) : int BT
- inorder(t1);
> val it = [4,2,5,1,6,3,7] : int list
- val t2 = full_tree(4);
> val t2 = Node(Node(Node #,2,Node#),1,Node (Node #,3,Node #)): int BT
- inorder(t2);
> val it = [8,4,9,2,10,5,11,1,12,6,13,3,14,7,15] : int list

```

Height Balanced Binary Tree

Definition: A height balanced binary tree is one in which each node satisfies the property that $|\text{count left_subtree} - \text{count right_subtree}| \leq 1$.

- Construct a height balanced binary tree from a list containing its preorder sequence.
We make use of functions *take* and *drop* which are of the type 'a list * 'a -> 'a list. These are defined in predefined structure called *List*.
 - List.take (list, n) - returns first n elements of the list
 - List.drop (list, n) - returns a list after dropping first n elements of a list.

```

- fun bal_tree [] = Null
  | bal_tree (x :: xs)= let val n = length xs div 2 in
  Node (bal_tree (List.take(xs, n)), x , bal_tree(List.drop (xs, n)))
  end;
> val bal_tree = fn : 'a list -> 'a BT
- bal_tree([1,2,3]);
> val it = Node (Node (Null,2,Null),1,Node (Null,3,Null)) : int BT
- val bt =bal_tree ([1,2,3,4,5,6,7,8]);
> val bt = Node(Node(Node #,2,Node#),1,Node(Node #,5,Node #)): int BT
- preorder bt;
> val it = [1,2,3,4,5,6,7,8] : int list
- inorder bt;
> val it = [3,2,4,1,6,5,7,8] : int list
- postorder bt;
> val it = [3,4,2,6,8,7,5,1] : int list
- val bt1 = swap bt;
> val bt1 = Node(Node(Node#,5,Node #),1,Node(Node#,2,Node #)): int BT
- preorder bt1;
> val it = [1,5,7,8,6,2,4,3] : int list
- inorder bt1;
> val it = [8,7,5,6,1,4,2,3] : int list
- postorder bt;
> val it = [3,4,2,6,8,7,5,1] : int list

```

Binary Search Tree

Definition: *Binary search tree* is a binary tree if all the keys on the left of any node (say, N) are numerically (alphabetically) less than the key in a node N and all the keys on the right of N are numerically (alphabetically) greater than the key of a node N.

If binary search tree is traversed in inorder, then the elements in the form of (key, value) pair are listed in increasing order of key.

- *Insert a pair of key and value as (key, value) in a binary search tree (BST).*

```

-      fun      insert((key, value), Null) = Node (Null, (key, value), Null)
          | insert ((key, value), Node (left, (k, v), right)) = if key < k
                        then Node( insert ((key, value), left), (k, v), right)
                        else Node(left, (k, v), insert ((key, value), right));
>      val insert = fn : (int * 'a) * (int * 'a) BT -> (int * 'a) BT
-      insert ((8,12.3),Null) ;
>      val it = Node (Null,(8,12.3),Null) : (int * real) BT
-      insert((2, 23.4), it);
>      val it = Node (Node (Null, (#, #),Null) , (8,12.3), Null): (int * real) BT
-      insert((7, 11.2), it);
>      val it = Node (Node (Null,(#, #),Node #),(8,12.3),Null) : (int * real) BT
-      insert((9,16.6), it);
>      val it = Node(Node (Null, (#,#), Node #),(8,12.3), Node
                        (Null,(#, #),Null)):(int * real) BT
-      val st = insert ((5, 22.3), it);
>      val st = Node (Node (Null,(#, #),Node #), (8,12.3), Node
                        (Null,(#, #),Null)) : (int * real) BT
-      preorder st;
>      val it = [(8,12.3),(2,23.4),(7,11.2),(5,22.3),(9,16.6)] : (int * real) list
-      inorder st;
>      val it = [(2,23.4),(5,22.3),(7,11.2),(8,12.3),(9,16.6)] : (int * real) list
-      postorder st;
>      val it = [(5,22.3),(7,11.2),(2,23.4),(9,16.6),(8,12.3)] : (int * real) list

```

- *Searching for a key in a binary search tree*

If key is found, then return the key and value pair otherwise return ~1.0 instead of value. Exceptional cases can also be handled by exceptions to be discussed later.

```

-      fun      search (key, Null)          =      (key, ~1.0)
          | search (key, Node (left, (k, v), right)) =      if key = k
                                                                then (k, v) else if key > k then
                                                                search (key, right)
                                                                else search (key, left);
>      val search = fn : int * (int * real) BT -> int * real
-      search (3, st);
>      val it = (3,~1.0) : int * real
-      search (2, st);
>      val it = (2,23.4) : int * real
-      search (8, st);

```

```
> val it = (8,12.3) : int * real
```

- Delete a node from a binary search tree

Deletion requires a function that deletes the smallest element from BST. Note that the smallest element in a BST is the leftmost leaf-node (if it exists, otherwise it is the root) and returns the (key, value) pair of the smallest element to enable tree reordering for deletion.

```
- fun delmin (Null) = (_, Null)
  | delmin (Node (Null, x, right)) = (x, right)
  | delmin (Node (left, x, right)) = let
    val (z, L) = delmin (left)
  in (z, Node (L, x, right)) end ;

> val delmin = fn : 'a BT -> 'a * 'a BT

- fun delete (k, Null) = Null
  | delete (k, Node (left, (k1,v), right)) =
    if (k = k1)
    then if left = Null then right
         else if right = Null then left else
         let val (z, R) = delmin (right) in Node (left, z, R) end
    else if k < k1 then Node (delete (k, left), (k1,v), right)
         else Node (left, (k1,v), delete (k, right));

> val delete = fn : int * (int * 'a) BT -> (int * 'a) BT
```

Functional Array

An array of type 'a array is a data structure having contiguous memory locations each holding the values of the type 'a. These locations are numbered by indices 0, 1, ...n-1, where n is the length of the array. The possible operations on array are update and retrieve value at kth index. The size of array is always fixed and access to array element is random in contrast to list where the size is variable and access is sequential. The conventional programming languages support array as a predefined data structure.

SML does not support array directly but we can implement it using binary tree. We call it as a functional array that provides a mapping from index to value with an update operation that creates a new array B as follows:

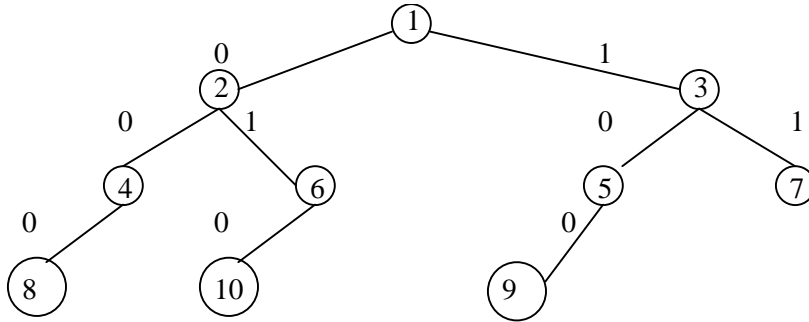
- B = update (A, k, value), such that B(k) = value and B(j) = A(j) , $\forall j \neq k$
- Array A continues to exist and additional array is created from it.

Implementation of Functional array in SML using height balanced binary tree

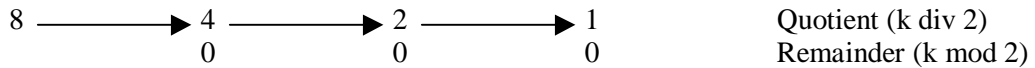
We can implement functional array using binary tree. If we can ensure that binary tree is height balanced then the retrieval time will be of $O(\log_2 n)$. The conventional array takes constant amount of time for retrieval but in SML we can not implement conventional array. The best could be that if we can store elements of an array in such a way that retrieval time is much lesser than that of retrieving an element from a list. Assuming the previous datatype definition of a binary tree, the elements of a functional array are stored in binary tree as follows:

- First index value of an array is the root of a binary tree.
- For any index k, the position in a binary tree is determined by starting from the root and repeatedly dividing k by 2 until it is reduced to 1.
- Each time when remainder is 0, move to left sub tree else if it is 1, then move to right sub tree

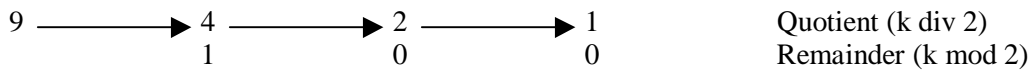
The binary tree corresponding to an array of 10 elements is shown below.



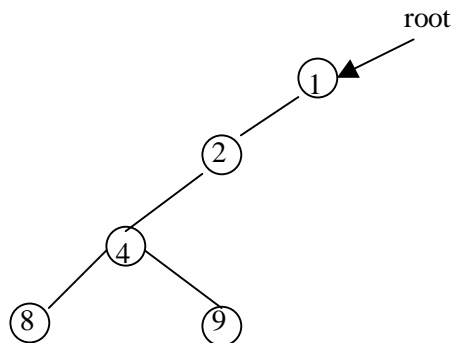
For example, an element at index 8 is stored as follows:



Node with index 8 will be stored / retrieved from root as left, left, left and node with index $k = 9$ is stored as follows from root is: right, left, left :



Therefore, the path traversed in a binary tree to store or retrieve an element at index 8 and 9 is as follows:



This method is used to store, update or retrieve the value corresponding to an index in a binary tree. It is noted that the tree comes out to be a height balanced binary tree. Few functions to manipulate functional array are given below:

- Create an array of size n with each index containing fixed value 'v'

- fun create (0, _) = Null
 | create(n, v) = if n=1 then update(Null,1,v)

```

else update(create(n-1, v),n,v);
> val create = fn : int * 'a -> 'a BT
- val p = create(5,0);
> val p = Node (Node (Node #,0,Null),0,Node (Node #,0,Null)) : int BT

```

- *Function for updating the value of a given index in an array*

```

- fun update (Null, index, value) = if (index = 1) then
Node(Null,value,Null) else Null
| update (Node(left,data,right), index, value) = if (index = 1)
then Node(left, value,right) else
if index mod 2 = 0 then
Node(update(left, index div 2,value), data, right)
else Node(left, d, update(right, index div 2, value)) ;
> val update = fn : 'a BT * int * 'a -> 'a BT
- update(Null,1,23);
> val it = Node (Null,23,Null) : int BT
- update(it,2,45);
> val it = Node (Node (Null,45,Null),23,Null) : int BT
- update(it,3,22);
> val it = Node (Node (Node (Null,45,Null),23,Node (Null,22,Null)) : int BT
- update(it,4,55);
> val it = Node (Node (Node (Node #,45,Null),23, Node (Null,22,Null)) : int BT
- update(it,5,11);
> val it = Node (Node (Node (Node #,45,Null),23,Node (Node #,22,Null)) : int BT
- update(it, 6,78);
> val it =Node(Node(Node#,45,Node #),23,Node (Node #,22,Null)): int BT
- update(it,7,33);
> val it = Node (Node (Node #, 45, Node # ), 23 ,Node (Node #, 22,
Node #)) : int BT
- val arr = update(it,8,99);
> val arr = Node (Node (Node #,45,Node #),23,Node (Node #,22,
Node #)) : int BT
- preorder arr;
> val it = [23,45,55,99,78,22,11,33] : int list
- inorder arr;
> val it = [99,55,45,78,23,11,22,33] : int list
- postorder arr;
> val it = [99,55,78,45,11,33,22,23] : int list
- val arr1 = update(arr, 4,40); ← changing the value at 4th index
> val arr1 = Node (Node (Node #,45,Node #),23, Node
(Node #,22,Node #)): int BT
- arr = arr1; ← array comparison for equality
> val it = false : bool

```

- *Retrieving a corresponding value from a given index of an array*

```

- fun retrieve (Null, _) = ~10000 ← exceptional case
| retrieve (index, Node(left, data, right)) = if index = 1 then
data else if index mod 2 = 0 then
retrieve ( index div 2, left)
else retrieve (index div 2, right);
> val retrieve = fn : int * int BT -> int
- retrieve(2, arr);

```



```

> val it = 45 : int
- retrieve(4, arr);
> val it = 55 : int
- retrieve(4, arr1);
> val it = 40 : int
- retrieve(9, arr);
> val it = ~10000 : int

```

- Delete a subtree index n and replace it by null.

```

- fun delete (n, Null) = []
  | delete (n, Node (left, data, right)) = if n = 1
    then Null else if n mod 2 = 0 then
      Node(delete(n div 2, left), data, right)
    else Node (left, data, delete(n div 2, right));
> val delete = fn :int * 'a BT -> 'a BT

```

Exception Handling

Exceptions are datatypes of error values used to minimize explicit testing. Exceptions are raised when the failures are discovered and appropriately handled elsewhere. General form of exception declaration is: **exception** *exception_name*. Exception name is a new constructor of the built-in type *exn* similar to value constructor in datatype declaration. Exception constructor can be used in pattern or expression in the same way as value constructors are used. We use a convention that *exception_name* should start with capital letter. Exception can also be a function.

```

- exception Fail;
> exception Fail = Fail : exn
- exception Failure of string;
> exception Failure = fn : string -> exn
- exception Badvalues of int;
> exception Badvalues = fn : int -> exn

```

In SML, the exception declaration can declare one or more exception names and these can be raised by a construct called *raise* to force a computation to terminate with an error signal..

Raising of an exception

The general form of exception raising is: **raise** *exception_name*. It creates an exception packet containing a value of built-in type *exn*.

```

- exception Bad;
> exception Bad = Bad : exn
- fun divide (x, y) = if y = 0 then raise Bad else x div y;
> val divide = fn : int * int -> int
- divide (12,3);
> val it = 4 : int
- divide(34,0);
> uncaught exception Bad raised

```

Let us define our own functions head and tail for getting head and tail of a given list. These functions should take care of the situations when applied on empty list. For this purpose we define exceptions and raise them suitably.

```

-      exception Head;
>      exception Head = Head : exn
-      exception Tail;
>      exception Tail = Tail : exn
-      fun      head (x::_) = x
           | head [] = raise Head;
>      val head = fn : 'a list -> 'a
-      head [2,3,4];
>      val it = 2 : int
-      head [];
>      uncaught exception Head raised
-      hd []; ← system defined head function
>      uncaught exception Empty raised ← system defined exception name
-      fun      tail (_::xs)= xs
           | tail [] = raise Tail;
>      val head = fn : 'a list -> 'a list
-      tail [2,3,4,5];
>      val it = [3,4,5] : int list
-      tail [];
>      uncaught exception Tail raised
-      tl []; ← system defined function for tail
>      uncaught exception Empty raised ← system defined exception name

```

Exception handling

An exception handler tests whether the result of an expression is an exception packet or not. The exception handler is always placed after an expression. The general form is:

```

exp handle <pat1> => exp1
          | <pat2> => exp2
          | <patn> => expn

```

An exception handler catches a raised exception if one of the pattern matches the value of an exception and then corresponding expression is evaluated under this binding. Hence if exp returns a normal value, then the handler simply passes this value on. If exp returns an exception packet's content (because of raising exception) then its contents are matched against the pattern. If pat_k is the first pattern to match then the result is the value of an expression exp_k (1 ≤ k ≤ n). It should be noted that exp, exp₁, ..., and exp_n must be of the same type. If we define our own exception names then declare them before their use otherwise built in exceptions can also be directly raised. The handlers are provided in the functions which use other functions directly or indirectly having exception raised in them.

```

-      fun len x = 1 + len (tail x) handle Tail =>0;
>      val len = fn : 'a list -> int
-      len [2,3,4];
>      val it = 3 : int
-      len [];
>      val it = 0 : int
-      fun head_list x = head x handle Hd=>0;
>      val head_list = fn : int list -> int

```

```

- head_list [3,4,5];
> val it = 3 : int
- head_list [];
> val it = 0 : int
- fun tail_list x = tail x handle Tail =>[0];
> val tail_list = fn : int list -> int list
- tail_list [];
> val it = [0] : int list

```

- *Function for finding nth element of the list assuming first element is stored at 0th index*

```

- exception Subscript;
> exception Subscript = Subscript : exn
- fun nth (x::__, 0) = x
  | nth (x::xs, n) = if n>0 then nth(xs, n-1)else raise Subscript
  | nth _ = raise Subscript;
> val nth = fn : 'a list * int -> 'a
- nth ([2,3,4,5],0);
> val it = 2 : int
- nth ([1,3,5,6], 5);
> uncaught exception Subscript

```

- *Handling of raised exception*

```

- fun findnth (l,n)=nth (l,n) handle Subscript =>0;
> val findnth = fn : int list * int -> int
- findnth ([34,56,12,33],6);
> val it = 0 : int
- findnth ([34,56,12,33],2);
> val it = 12 : int

```

- *Function for computing the sum of a list's elements at position n, f(n), f(f(n)), The sequence of integer terminates at the first value out of range using exception.*

```

- fun f (n) = n-2;
> val f = fn : int -> int
- fun chain (x, n)=nth(x,n) + chain(x, f(n)) handle Subscript => 0;
> val chain = fn : int list * int -> int
- chain ([23,45,65,12],2);
> val it = 88 : int
- chain ([23,45,67],1);
> val it = 45 : int

```

- *Check whether a given positive integer is a square. Display false if number is negative integer*

```

- exception Neg;
> exception Neg = Neg : exn
- fun sq x:int = x*x;
> val sq = fn : int -> int
- fun issq i = if i > 0 then sq (round (Math.sqrt (real i))) = i
  else raise Neg;
> val issq = fn : int -> bool
- issq ~45; ← exception is raised in this function

```

```

>      uncaught exception Neg
-      fun is_sq i = issq i handle Neg => false;
>      val is_sq = fn : int -> bool
-      is_sq 64;
>      val it = true : bool
-      is_sq 32;
>      val it = false : bool
-      is_sq ~64; ←—— exception is handled in this function
>      val it = false : bool

```

Benefits of the exception mechanism

- We are forced to consider the exceptional cases otherwise we will get an uncaught exception at run time.
- We can separate the special cases from the normal case in the code rather than putting explicit checks.
- Another typical use of exception is to implement backtracking which requires exhaustive search of a state space.

Backtracking using Exception Mechanism

Let us consider the following example which implements backtracking. Exceptions are raised and handled in the same function. When exception is raised, it backtracks to previous solution with the help of handle exception. The function convert converts a given amount in number of coins starting from the highest to the lowest in the best possible way. The list of coins is passed as an argument along with the amount.

```

-      exception Invalid of int;
>      exception Invalid = fn : int -> exn
-      fun      convert (xs,0) =[]
              | convert ([], amount)      =      raise Invalid (1)
              | convert (x::xs, amount) =
                  if amount < 0 then raise Invalid (2) else
                  if x > amount then convert (xs, amount)
                  else x::convert(c::xs,(amount-x)))
              handle Invalid (1) => convert (xs, amount);
                                     ↖
                                     for backtracking

>      val convert = fn : int list * int -> int list
-      convert([5,3,2],56);
>      val it = [5,5,5,5,5,5,5,5,5,5,3,3] : int list
-      convert([5],~23);
>      uncaught exception Invalid raised
-      convert([],23);
>      uncaught exception Invalid raised
-      convert([5,3],4);
>      uncaught exception Invalid raised

```

This function raises exception when amount is negative. It is handled in another function which makes use of a function convert.

```

-      fun      convert1 ([], amount) = []
          | convert1 (list,amount) = convert(list,amount)
                                   handle Invalid (2) => [];
>      val convert1 = fn : int list * int -> int list
-      convert1 ([5,3,2],56);
>      val it = [5,5,5,5,5,5,5,5,5,5,3,3] : int list
-      convert1([5],~23);
>      val it = [] : int list
-      convert1([],23);
>      val it = [] : int list
-      convert1([5,3],4);
>      uncaught exception Invalid raised

```

The function `convert1` raises exception when the amount can't be expressed by any combination of coins. This situation can further be handled in yet another function which calls `convert1` and handles the exception raised in `convert1`.

```

-      fun      convert2 ([], amount) = []
          | convert2 (list, amount) = convert1(list, amount)
                                   handle Invalid (1) => [];
>      val convert2 = fn : int list * int -> int list
-      convert2 ([5,3,2],56);
>      val it = [5,5,5,5,5,5,5,5,5,5,3,3] : int list
-      convert2 ([5],~23);
>      val it = [] : int list
-      convert2([],23);
>      val it = [] : int list
-      convert2([5,3],4);
>      val it = [] : int list

```

Structure Declaration

Declarations (value, function, type, datatype, etc.) can be grouped to form a structure by enclosing them in the keywords `struct` and `end`. The general form of structure declaration is:

```
structure      struct_name      =      structure_expression
```

where,

```

structure_expression      =      struct
                                   declarations
                                   end

```

The result of *structure_expression* is bound to *struct_name*. The structure declaration binds the *struct_name* to two environments, one a type environment containing bindings for the type / datatype constructors and other a value environment containing *val*, *fun* declarations inside the structure expression. All identifiers are available for the user via composite identifiers such as *struct_name* . *val_name*.

Definition: Queue is a special type of list in which addition of an element is done at one end called rear and deletion at another end called front.

We use pre-defined data type list for implementing queue.

```
structure      Queue1 =      struct
```

```

type 'a queue = 'a list;
exception Qerror;
val emptyq = [];
fun    nullq ([]) = true
      | nullq (_::_) = false ;
fun    addq (q, x) = q @ [x];
fun    delq (x::q) = q
      | delq [] = raise Qerror ;
fun    headq [] = raise Qerror
      | headq (x::q) = x;
end;

```

We notice that a function for adding an element in the queue takes time linear to the length of the queue whereas both *delq* and *headq* take constant amount of time. In queue, ideally all the operations should be done in constant amount of time but since queue is implemented using predefined datatype list adding an element at the rear end can not be achieved in constant time. Of course one can think of implementing queues using altogether different datatype shown later.

The above structure can be stored in a file named (say) program1 using some editor and is opened at SML prompt by use primitive as follows:

```

-      use "program1";
>      [opening program1]
structure Queue1 :
sig
    type 'a queue = 'a list
    exception Qerror
    val addq : 'a list * 'a -> 'a list
    val delq : 'a list -> 'a list
    val emptyq : 'a list
    val headq : 'a list -> 'a
    val nullq : 'a list -> bool
end
val it = () : unit
-      Queue1.nullq [];
>      val it = true : bool
-      Queue1.nullq [2,3,4];
>      val it = false : bool
-      Queue1.delq [];
>      uncaught exception Qerror raised
-      Queue1.delq [2,3,4];
>      val it = [3,4] : int list
-      Queue1.addq ([3,4],9);
>      val it = [3,4,9] : int list

```

If we open a structure at SML by using open *struct_name*, then all the identifiers are used without prefixing structure name. Let us open earlier defined structure Queue1.

```

-      open Queue1;
>      opening Queue1
type 'a queue = 'a list
exception Qerror
val addq : 'a list * 'a -> 'a list
val delq : 'a list -> 'a list

```

```

val emptyq : 'a list
val headq : 'a list -> 'a
val nullq : 'a list -> bool
- delq [4,5,6,7];
> val it = [5,6,7] : int list
- addq([1,2,3],6);
> val it = [1,2,3,6] : int list

```

Let us define another structure containing a new datatype defining *queue*. Here *addq* will take a constant amount of time but both *delq* and *headq* are linear in the length of the queue.

```

structure Queue2 = struct
  datatype 'a queue = emptyq | addq of 'a queue * 'a;
  exception Qerror;
  fun nullq (emptyq) = true
    | nullq (addq (_,_)) = false ;
  fun delq (emptyq) = raise Qerror
    | delq (addq (emptyq, x)) = emptyq
    | delq ( addq (q, x)) = addq (delq (q), x) ;
  fun headq (emptyq) = raise Qerror
    | headq (addq (emptyq, x)) = x
    | headq (addq (q, x)) = headq (q) ;
end;

```

Since both the structures defined above are not efficient, we define yet another structure. Most of the time, the overall cost of adding and deleting an element is less than linear time in this representation. Here a peculiar representation of queues has been used. The function "normal" is special to this particular representation and is not visible to the user of the queue. The concept of information hiding is used.

```

structure Queue3 = struct
  datatype 'a queue = Queue of ('a list * 'a list);
  val emptyq = Queue ([], []);
  fun reverse (L) = let fun rev ([], M) = M
    | rev (x::xs, M) = rev (xs, x::M) ;
    in rev (L, []) end ;
  fun normal (Queue ([], tails)) = Queue (reverse (tails), [])
    | normal (q) = q ;
  fun addq (Queue (heads, tails), x) =
    normal (Queue (heads, x::tails));
  exception Qerror;
  fun delq (Queue (x::heads, tails)) =
    normal (Queue (heads, tails))
    | delq (Queue ([], _)) = raise Qerror ;
  fun headq (Queue (x::heads, tails)) = x
    | headq (Queue ([], _)) = raise Qerror ;
end;

```

Let us store both the structures in a file named program2. File can be opened

```

- use "program2";
> [opening program2]
structure Queue2 :
sig

```

```

datatype 'a queue = addq of 'a queue * 'a | emptyq
exception Qerror
val delq : 'a queue -> 'a queue
val headq : 'a queue -> 'a
val nullq : 'a queue -> bool
end
structure Queue3 :
sig
  datatype 'a queue = Queue of 'a list * 'a list
  exception Qerror
  val addq : 'a queue * 'a -> 'a queue
  val delq : 'a queue -> 'a queue
  val emptyq : 'a queue
  val headq : 'a queue -> 'a
  val normal : 'a queue -> 'a queue
  val reverse : 'a list -> 'a list
end
val it = () : unit
- Queue2.addq( Queue2.emptyq,3);
> val it = addq (emptyq,3) : int Queue2.queue
- Queue2.addq(it,5);
> val it = addq (addq (emptyq,3),5) : int Queue2.queue
- Queue2.addq(it,2);
> val it = addq (addq (addq #,5),2) : int Queue2.queue
- val q = it; ← earlier constructed queue is assigned to q
> val q = addq (addq (addq #,5),2) : int Queue2.queue
- Queue2.delq (q);
> val it = addq (addq (emptyq,5),2) : int Queue2.queue
- Queue2.headq q;
> val it = 3 : int
- open Queue2;
> opening Queue2
datatype 'a queue = addq of 'a queue * 'a | emptyq
exception Qerror
val delq : 'a queue -> 'a queue
val headq : 'a queue -> 'a
val nullq : 'a queue -> bool
- delq q;
> val it = addq (addq (emptyq,5),2) : int queue
- open Queue3;
> opening Queue3
datatype 'a queue = Queue of 'a list * 'a list
exception Qerror
val addq : 'a queue * 'a -> 'a queue
val delq : 'a queue -> 'a queue
val emptyq : 'a queue
val headq : 'a queue -> 'a
val normal : 'a queue -> 'a queue
val reverse : 'a list -> 'a list
- emptyq;
> val it = Queue ([],[ ]) : 'a queue
- addq(it, 3);
> val it = Queue ([3],[ ]) : int queue
- addq(it, 4);
> val it = Queue ([3],[4]) : int queue

```



```

-      addq(it,7);
>      val it = Queue ([3],[7,4]) : int queue
-      addq(it,9);
>      val it = Queue ([3],[9,7,4]) : int queue
-      val p = it;
>      val p = Queue ([3],[9,7,4]) : int queue
-      headq p;
>      val it = 3 : int
-      delq p;
>      val it = Queue ([4,7,9],[1]) : int queue
-      delq it;
>      val it = Queue ([7,9],[1]) : int queue
-      addq (it,1);
>      val it = Queue ([7,9],[11]) : int queue
-      addq(it,10);
>      val it = Queue ([7,9],[10,11]) : int queue
-      addq(it, 12);
>      val it = Queue ([7,9],[12,10,11]) : int queue
>      val p = Queue ([7,9],[12,10,11]) : int queue
-      delq p;
>      val it = Queue ([9],[12,10,11]) : int queue
-      delq it;
>      val it = Queue ([1,10,12],[1]) : int queue
-      delq it;
>      val it = Queue ([10,12],[1]) : int queue
-      delq it;
>      val it = Queue ([12],[1]) : int queue
-      delq it;
>      val it = Queue ([],[1]) : int queue
-      delq it;
>      uncaught exception Qerror raised

```

Functions & variables can have the same names in different structures as they are accessed with their structure name.

Signature

A signature is used to give user's view of a structure. In SML, the abstraction is captured in signature by specifying the type constructors and value declarations. General form of a signature declaration is:

$$\begin{array}{lcl}
 \text{signature } sig_name & = & \text{signature_expression.} \\
 \text{signature_expression} & = & \mathbf{sig} \quad \text{specifications} \quad \mathbf{end}
 \end{array}$$

A signature should contain all those specifications of declaration which user wants to know. Signature is used along with structure. A module is constructed by matching a signature with a structure. The matching checks that each *sig_name* is declared in the structure. The *struct_name* is an instance of *sig_name* written as *struct_name . sig_name*. In this way the user's view of structures in SML is determined by the signature. Thus all those functions definitions, type constructors, value definitions etc., which are not relevant to user and are local to structures are not included in the signature. The user is only allowed to use whatever is shown in the signature expression. The structure declaration must contain a declaration for each identifier specified in

the signature. . The structure and signature names can be same also. Let us consider an example of constructing signature and its use in structure for manipulating stack.

Definition: Stack is a special type of list where addition or deletion of an element is done at only one end. Other end is sealed. The various operations on stack are: push (adding an element at the top), pop (deleting a top element), top (getting top element).

```
signature Stack =
sig
  type 'a stack
  exception Error
  val null : 'a stack
  val null_stack : 'a stack -> bool
  val push: 'a stack * 'a -> 'a stack
  val pop: 'a stack -> 'a stack
  val top : 'a stack -> 'a
end;

structure Stack : Stack = struct
  type 'a stack = 'a list;
  exception Error;
  val null = [];
  fun null_stack ([]) = true
    | null_stack (_::_) = false ;
  fun push (x, s) = s @ [x];
  fun del (x::s) = s
    | del [] = raise Error ;
  fun pop (s) = del (s) handle Error => [];
  fun top [] = raise Error
    | top (x::s) = x;
end;
```

Store signature and structure in a file named "sprogram". At SML prompt, if we type use "sprogram", signature is opened.

```
> [opening sprogram]
signature Stack =
sig
  type 'a stack
  exception Error
  val null_stack : 'a stack -> bool
  val push : 'a * 'a stack -> 'a stack
  val pop : 'a stack -> 'a stack
  val top : 'a stack -> 'a
end
structure Stack : Stack
- open Stack;
> opening Stack
type 'a stack = 'a list
exception Error
val null : 'a stack
val null_stack : 'a stack -> bool
val push : 'a * 'a stack -> 'a stack
val pop : 'a stack -> 'a stack
val top : 'a stack -> 'a
```

```

-      push(12, null);
>      val it = [12] : int stack
-      push(43,it);
>      val it = [12,43] : int stack
-      push(10,it);
>      val it = [12,43,10] : int stack
-      val s = pop(it);
>      val s = [43,10] : int stack
-      pop(s);
>      val it = [10] : int stack
-      top(it);
>      val it = 10 : int
-      null_stack(s);
>      val it = false : bool

```

There are two ways to match signatures and structures.

Transparent signature matching

Matching defined in above example is of transparent signature matching type. It gives weak support for hiding of the details of the structure. From the system's response as `val s = [43,10] : int stack`, user gets information about the structure of stack (implemented as list) even though it is not specified in the signature. We can hide even such details using the following type of signature.

Opaque signature matching:

This type of signature matching is more strict. We can hide the actual definition of stack by the user defined in the structure. The following declaration is to be made:

```
structure struct_name :> sig_name = structure_expression
```

Consider the previous example of stack. The signature definition remains the same. The structure definition will include opaque signature matching as follows:

```
structure Stack :> Stack = struct (* entire body as above *) end;
```

Given below is the interactive session using stack structure.

```

-      Stack.push(12, Stack.null);
>      val it = - : int Stack.stack
-      Stack.push(13,it);
>      val it = - : int Stack.stack
-      val s = Stack.push(54,it);
>      val s = - : int Stack.stack
-      Stack.top(s);
>      val it = 12 : int
-      Stack.pop(s);
>      val it = - : int Stack.stack

```

Here we notice that actual implementation structure of stack is hidden. If structure Stack is opened as "open Stack", then identifiers available for use are not to be prefixed by structure name.

Eager and Lazy Evaluation

When a program executes, computation generally proceed via different evaluation orders, according to which one of many different evaluation strategies is being pursued. The evaluation strategies, are differentiated according to the method of passing arguments to functions. Mainly, parameter passing methods are *call by value*, *call by reference*, *call by name* and *call by need* a variance of call by name.

Call by value is a computation rule of programming language where the arguments are fully evaluated before outer function applications are evaluated. Therefore, corresponding to the arguments of functions, only values are passed. That is why this strategy is called call-by-value. It is used in imperative languages such as Java, C, Pascal and functional programming languages such as SML, Scheme etc.

In *call by reference* the address (reference / pointer) of actual arguments are passed to functions instead of the values. This computation rule is available in all imperative programming languages. Arrays are passed by reference in all imperative languages. In *call by name* the arguments are passed unevaluated and are evaluated every time when needed in the body of the function. If the argument is not needed, then it will not be evaluated at all.

Call by need is an optimization of call-by-name such that an expression corresponding to an argument is evaluated at most once, if at all required. That means once it is evaluated, then the result is remembered and the next access to the corresponding formal parameter uses this value. Historically, all imperative languages employ call by value, and call by reference parameter passing mechanism and many provide call-by-name facilities as well.

Evaluation Strategies

Let us illustrate different evaluation strategies using the following SML functions

```
- fun constant (x) = 1;  
> val constant = fn : 'a -> int  
- fun cube (x : int) = x * x * x;  
> val cube = fn : int -> int
```

Call by value: (value of argument of a function is evaluated)

Case1: Call to constant function

```
- val p = constant (cube (cube (cube (2))));  
> val p = 1
```

Evaluation of the function is done as follows:

```
p = constant (cube (cube (cube (2))))  
= constant (cube (cube (2*2*2))) = constant (cube (cube (8)))  
= constant (cube (8*8*8)) = constant (cube (512))  
= constant (512*512*512) = constant (134217728)  
= 1
```

Case2: Call to cube function.

```
- val q = cube (cube (cube (2)));  
> val q = 134217728 : int
```

Evaluation:

```

q      =      cube (cube (cube (2)))
      =      cube (cube (2*2*2))      =      cube (cube (8))
      =      cube (8*8*8)            =      cube (512)
      =      512*512*512            =      134217728

```

We notice that the amount of time required in both the cases is same, whereas in case1, we have done calculation in vain as the result is independent of what we have calculated. Let us see other mechanisms for evaluation which are not available in SML

Call by name: (outermost reduction order).

Case1: Call to constant function
 - val p = constant (cube (cube (cube (2))));
 > val p = 1 at one step

Case2: Call to cube function
 - val q = cube (cube (cube (2)));
 > val q = 134217728 : int

Evaluation:

```

q      =      cube (cube (cube (2))) ← outermost cube is evaluated
      =      cube (cube (2)) * cube (cube (2)) * cube (cube (2))
      =      cube (2) * cube (2) * cube (2) * cube (cube (2)) * cube (cube (2))
      =      (2*2*2) * cube (2) * cube (2) * cube (cube (2)) * cube (cube (2))
      =      8 * (2*2*2) * cube (2) * cube (cube (2)) * cube (cube (2))
      =      64 * (2*2*2) * cube (cube (2)) * cube (cube (2))

```

Here, the result of function in case1 is instant, whereas in case2, lots of computations are repeated and time required in this case is much higher than as compared to call by value (eager evaluation) method. Best of both the methods are combined in call by need (lazy evaluation). Let us see the way call by need handles such computations.

Call by need: (similar to call by name except that the repetitions are avoided)

Case1:
 - val p = constant (cube (cube (cube (2))));
 > val p = 1 at one step

Case2:
 - val q = cube (cube (cube (2)));
 > val q = 134217728 : int

Evaluation:

```

q = cube (cube (cube (2)))
Let, x = cube (cube (2)) and y = cube (2) = 2 * 2 * 2 = 8
Then, x = cube (y) = y * y * y = 8 * 8 * 8 = 512
Finally, q = cube (x) = x * x * x = 512 * 512 * 512 = 134217728

```

There are broadly two categories of evaluation strategies in functional languages viz., *eager evaluation* and *lazy evaluation*.

Eager Evaluation Strategy

Any evaluation strategy where evaluation of all function arguments is done before their values are required. *Call-by-value* is a typical example of such evaluation where the values of all arguments are evaluated and are passed. In Functional languages, this type of evaluation strategy is called eager evaluation. Eager evaluation does not specify exactly when argument evaluation takes place. The reduction of the innermost redex forces evaluation of arguments before applications

are performed. It is also called applicative reduction order. The term *eager evaluation* was invented by Carl Hewitt and Henry Baker. Eager evaluation is used to implement strict functions. SML, Scheme have eager evaluation method.

Definition: A function is said to be *strict* if the result is undefined when it is applied to undefined argument(s) and said to be *non-strict* otherwise.

Lazy Evaluation Strategy

Lazy evaluation is one where evaluation of an argument only starts when it is required first time. This evaluation strategy combines *normal order evaluation* with *updating*. Call by need uses lazy evaluation strategy.

Definition: *Normal order evaluation* means that an expression is evaluated only when its value is needed in order for the program to return (the next part of) its result. Call by name uses normal order evaluation method.

Definition: *Updating* means that if an expression's value is needed more than once (i.e. it is shared), the result of the first evaluation is remembered and subsequent requests for it will return the remembered value immediately without further evaluation.

Lazy evaluation is one evaluation strategy used to implement non-strict functions. Choosing a particular evaluation strategy is not trivial. The implications are wide-ranging and, in some cases, quite subtle. It affects execution efficiency, programming style, interaction, etc. Lazy evaluation is a very desirable property of a functional language. This means for instance that an expression being a function argument is not evaluated at the point of call (as is usually the case with traditional languages and non-lazy functional ones). Instead the expression is evaluated when its value is needed, if at all. Lazy evaluation makes it possible to have infinite lists, infinite trees, etc, where the evaluation is done incrementally on demand. Lazy evaluation is must for interactive input/output in a pure functional language (explained later).

Functional languages can be eager, lazy or a mix of the two. Functional language SML uses eager evaluation whereas languages like Haskell and Miranda use lazy evaluation scheme. Functional programming languages which employs lazy evaluation strategies are called *lazy languages* otherwise *non lazy languages*.

In lazy languages, function arguments may be infinite data structures (especially lists), the components of which are evaluated as needed. Lazy evaluation can lead to a less constrained programming style. It helps in delay of unnecessary computation (explained later). Full Laziness is obtained by further optimizing a lazy evaluation by transforming a program by evaluating all those sub expressions in a function body which do not depend on the function's arguments.

Generally, in non lazy languages, it is essential that conditional expressions have special form whose evaluation does not require full evaluation. Exceptions to the rule of strict (eager) evaluation for the conditional expression *if E then E₁ else E₂* are given below. Let us assume that the symbol ϖ denotes undefined expression.

<i>Conditional expression</i>	<i>Evaluated value</i>
If E = true then E ₁ else ϖ	E ₁
If E = false then ϖ else E ₂	E ₂
If E = ϖ then E ₁ else E ₂	ϖ

If non lazy languages do not provide lazy evaluation of conditional expression, then it could be simulated (explained later). In lazy functional languages, conditional expressions can be treated as ordinary functions such as

```

- fun conditional (true, x, y) = x
  | conditional (false, x, y) = y
> val conditional = fn : bool * 'a * 'a -> 'a

```

The evaluation of a function call *conditional* (E, E_1, E_2) has the same behaviour as *if E then E₁ else E₂* under lazy evaluation strategy.

It is evident that common style of programming is not possible for both lazy and eager evaluations because of subtle differences. Some commonality can be achieved by transformational approach to program construction.

Lazy Evaluation in SML

The evaluation rule in SML is eager (strict) evaluation, while most of pure functional languages adopt lazy evaluation. SML provides lazy evaluation of conditional expressions and infix boolean operators.

Conditional expression

As a special case, SML evaluates conditional expression using lazy strategy. Consider *if_then_else* conditional expression *if E then E₁ else E₂* that is interpreted as follows:

- If E is true then E₁ is evaluated and E₂ is not.
- If E is false then E₂ is evaluated and E₁ is not.

Examples:

```

- if true then 4 else (23 div 0) ;
> val it = 4 : int
- if false then (4 div 0) else 5;
> val it = 5 : int

```

← undefined

Boolean Operators

In SML, boolean infix operators *andalso*, *orelse* are not functions but are conditional expressions, so are evaluated using lazy strategy. All other SML infixes are really functions. Evaluation of $E_1 \text{ andalso } E_2$ amounts to basically evaluating *if E₁ then E₂ else false*, where E₁ and E₂ are expressions. So an expression E₂ is evaluated if required or needed. Similarly $E_1 \text{ orelse } E_2$ is equivalent to *if E₁ then true else E₂*. In SML infix operator *andalso* is defined as follows:

```

infix andalso
fun true andalso b = b
  | false andalso b = false

```

← b is not evaluated

```

- false andalso 4 > (6 div 0);
> val it = false : bool
- true orelse 4 > (6 div 0);
> val it = true : bool

```

← undefined

Delaying Evaluation of an expression

We know that function bodies are not evaluated until the functions are applied. This means that the evaluation of any expression can also be delayed by embedding the expression within the body of a function whose argument is dummy (unused value). Normally, empty tuple `() : unit` is used for this purpose.

```
- fun delay () = Exp
> val delay = fn : unit -> 'a
```

Here 'a is the type of an expression Exp. An application of delay to an empty tuple will require an evaluation of Exp.

Simulation of Normal Order Evaluation

It is possible to simulate lazy evaluation within an eager languages by making use of higher-order functions which means that the functions are passed in place of simple arguments. But the resulting programs are more difficult to write, read and maintain. Delay mechanism helps simulating normal order evaluation (call by name) in non lazy languages. The function compute defined below have arguments of the type 'a list. Here the lists are evaluated at the time of applying compute function as eager evaluation is used in SML.

```
- fun compute (x, y) = if null (x) then [] else hd(x) :: y;
> val compute = fn : 'a list * 'a list -> 'a list
- val p = compute([3,4,5],[6,7,8,9]);
> val p = [3,6,7,8,9] : int list
- val q = compute([], [1,2,3,4,5]);
> val q = [] : int list
```

If we want to delay the evaluation of lists at the time of applying function, we can modify above function as follows:

```
- fun delayf (fun1, fun2) = if null (fun1()) then [] else hd(fun1()):: fun2();
> val delayf = fn : (unit -> 'a list) * (unit -> 'a list) -> 'a list
```

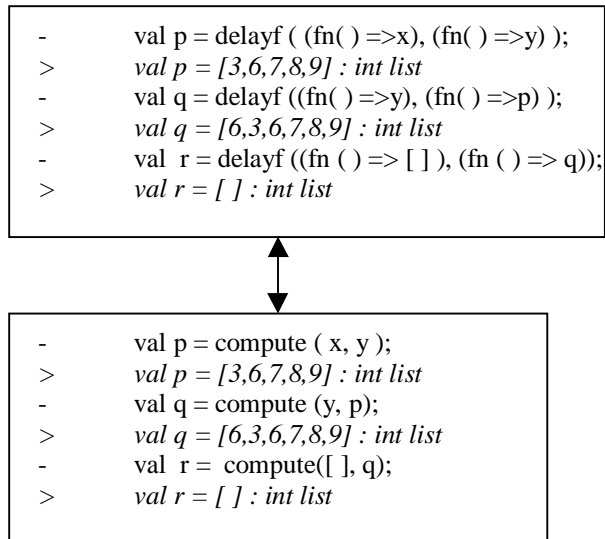
The functions *compute* and *delayf* are similar except that the arguments of function compute are of the type 'a list and that of delayf are functions of the type unit -> 'a list. Calls to both the functions are:

- `compute(a, b)`
- `delayf ((fn() => a), (fn() => b))`.

They produce the same result but the evaluation of *a* and *b* are delayed until the arguments to *delayf* are applied. In particular if the value of *a* is null then *b* is not evaluated. Few calls to both the functions are given below:

```
- val x=[3,4,5];
> val x = [3,4,5] : int list
- val y = [6,7,8,9];
> val y = [6,7,8,9] : int list
- fun fun1() = x;
> val fun1 = fn : unit -> 'a list
- fun fun2() = y;
> val fun2 = fn : unit -> 'a list
```

The following segments of SML statements are equivalent.



Therefore, the functions as arguments to other functions delay evaluation until the function argument is applied. If function argument is applied more than once then in this kind of simulation, function argument is evaluated as many times as it is applied. This is basically a simulation of call by name and not call by need (lazy evaluation). If we want to avoid repetition of evaluation of function arguments, then this can be achieved by using local definition (let in SML) to compute the value at the time of first application and then use that local variable whenever it is required. Consider the following function.

Non lazy version:

```

-   fun comp (x, y) = if null(x) then [] else y @ y @y;
>   val comp = fn : 'a list * 'b list -> 'b list

```

Simulated version:

```

-   fun delay1 (fun1,fun2) = if null(fun1()) then []
                             else fun2() @ fun2() @ fun2();
>   val delay1 = fn : (unit -> 'a list) * (unit -> 'b list) -> 'b list

```

Here if *fun1()* is not null then *fun2()* is evaluated thrice. To avoid this repetition, let us rewrite another version of the same function.

Lazy version:

```

-   fun delay_lazy (fun1,fun2) = if null(fun1()) then []
                                 else let val temp = fun2() in temp @ temp @ temp end;
>   val delay_lazy = fn : (unit -> 'a list) * (unit -> 'b list) -> 'b list

```

The function, *delay_lazy* simulates the lazy evaluation strategy.

Eagerly evaluated conditional expression in non lazy languages could also be easily simulated by using delay mechanism as follows:

```

-   fun cond (x, y, z) = if x then y() else z();
>   val cond = fn : bool * (unit -> 'a) * (unit -> 'a) -> 'a
-   val n = cond (true, (fn() => 4), (fn() => (34 div 0)));
>   val n = 4 : int

```

undefined

It is evident that the second argument (undefined) of application of function *cond* has not been evaluated. The effect is same as that of lazy evaluation and the function is behaving as non strict

function. The evaluation has been delayed by using function as an argument. Similarly, the first argument is not evaluated in the following application of `cond` in the following case.

```
- val n = cond (false, (fn() => (4 div 0)), (fn() => 34));
> val n = 34 : int
```

↑
undefined

Advantages of Lazy evaluation

There are numerous advantages as explained below. The functions are written in SML for illustration.

- ***Avoids unnecessary computation***

Lazy languages avoid unnecessary computations. If long computations are not needed, they will not be done in lazy evaluation environment. Let us consider a case of lists equality. If we compute an equality of two long lists with different first items, then their tails are not evaluated in lazy languages.

i. Comparing two lists:

```
- infix equal;
> infix equal
- fun ([] equal []) = true
  | ((a::x) equal (b::y)) = (a = b) andalso (x equal y)
  | ((a::x) equal []) = false
  | ([] equal (b::y)) = false;
> val equal = fn : 'a list * 'a list -> bool
```

In strict language such as SML, tails are also evaluated even if $a \neq b$ because `equal` is strict function and requires both the arguments to be evaluated before executing its body.

ii. Comparing two binary trees for preorder sequence equality.

Here we make use of `equality` function defined for list.

```
- datatype 'a BT = Null | Node of 'a BT * 'a * 'a BT;
> datatype 'a BT = Null | Node of 'a BT * 'a * 'a BT
- fun preorder Null = []
  | preorder (Node (left, data, right)) =
    [data] @ preorder left @ preorder right;
> val preorder = fn : 'a BT -> 'a list ▲ {list equality}
- fun tequality(t1, t2) = preorder (t1) equal preorder (t2);
> val tequality = fn : "a btree * "a btree -> bool
- val t = Node(Node (Node (Null, 3, Null), 2, Null), 1,
  Node(Node(Null,5, Null), 4, Node(Null, 6, Null)));
> val t = Node(Node (Node #, 2, Null), 1, Node (Node #,4,Node #)) : int BT
- val t1 = Node(Node (Null, 2, Null),1, Node(Node(Null,4, Null),3,
  Node(Node(Null,6,Null), 5, Null)));
> val t1 = Node(Node (Node #, 2, Null), 1, Node (Node #,3,Node #)) : int BT
- val x = tequality (t, t1);
> val x = true : bool
- tequality (Node(Null, 6, Null), Node (Node(Null, 4, Null), 2, Null))
> val it = false : bool
```

Evaluation:

```
tequality (Node(Null, 6, Null), Node (Node(Null, 4, Null), 2, Null))
= preorder (Node(Null, 6, Null)) equal preorder (Node (Node(Null, 4, Null), 2, Null))
```

```

= [6] @ preorder(Null) @ preorder(Null)
  equal
  [2] @ preorder(Node (Node(Null, 4, Null)) @ preorder(Null)
= (6:: [] @ preorder(Null) @ preorder(Null)
  equal
  (2 :: [] @ preorder(Node (Node(Null, 4, Null)) @ preorder(Null)
= (6 = 2) andalso ([] @ preorder(Null) @ preorder(Null) )
  equal
  ([] @ preorder(Node (Node(Null, 4, Null)) @ preorder(Null)
= false andalso ..... equal .....
= false

```

not evaluated

During lazy evaluation, calls to functions moves back and forth to give rise co-routine like behaviour whereby each function is driven by request for some output from another function requiring some input. Therefore, the functions call each other and returns values as and when required. For example, in tree equality, if first value of both the trees in preorder sequence are equal then second element of first tree is computed followed by second element of second tree and so on otherwise false is returned. So at no point in time both the lists are computed completely thus avoiding redundant computation.

- ***Avoids non termination***

If a function terminates via any reduction order, then the result will be the same using lazy evaluation.

i. *The functions which are non terminating in non lazy languages can terminate in lazy languages.*

For example, define a function for computing factorial as follows:

```

- fun conditional (x, y, z) = if x then y else z;
> conditional = fn : bool * 'a * 'a -> 'a
- fun fact n = conditional (n = 0, 1, n * fact(n-1));
> val fact = fn : int -> int

```

Eager evaluation:

The function *fact* is non terminating in non lazy (eager) languages. If we call *fact(0)*, it goes into non termination as the arguments of function *conditional* gets evaluated before its body is executed.

```

- val x = fact (0);
- val y = fact (2);

```

Non terminating

Examples:

```

i. fact (0) = conditional (true, 1, 0 * fact(-1) ) ← non termination
ii. y      = fact(2)
      = conditional(false, 1, 2 * fact(1))
      = conditional (false,1, 2 * conditional (false, 1, 1 * fact(0)))

```

Because of *fact (0)* in the argument of *conditional* it goes into non termination as explained above.

Lazy evaluation:

```

-      val x = fact (0);
>      val x = 1 : int
-      val x = fact (4);
>      val y = 24 : int

```

It gives result as 1 for fact(0) because 0 * fact(-1) will not be evaluated at all.

As already mentioned earlier, lazy evaluation can be simulated to some extent in eager languages. The same function *fact* is rewritten using delay mechanism for eager languages. Of course the function becomes difficult to be understood. We make use of function **cond** that simulated lazy evaluation.

```

-      fun fact n = cond ( n = 0, (fn() => 1), (fn() => n* fact (n-1)));
>      val fact = fn : int -> int
-      fact 0;
>      val it = 1 : int
-      fact 4;
>      val it = 24 : int

```

ii. Define a function to obtain first n elements of an infinite list generated.

Let us write a function named *gen* that generates an infinite list of integers in increasing order starting from some initial value as follows:

```

-      fun gen(n) = n::gen(n+1);
>      val gen = fn : int -> int list

```

Now we will use *gen* function to obtain first n elements as follows:

```

-      fun      get (1, x::xs) =      [x]
              | get (n, []) =      [0]
              | get (n, x::xs) =    x:: get(n-1, xs);
>      val get = fn : int * int list -> int list

```

Eager evaluation:

```

-      val x = get (3, gen(1) ); ← non termination

```

Evaluation:

```

x = get (3, gen(1) ) = get(3, 1 :: gen(2))
  = get(3, 1 :: 2 :: gen(3)) = get(3, 1 :: 2 :: 3 :: gen(4))
  .....

```

It is non terminating because of infinite list as an argument of function *get*. The argument is to be evaluated before executing the body of *get*.

Lazy evaluation:

```

-      val x = get (3, gen(1) );
>      val x = [1,2,3]

```

It gives a list of first 3 elements from infinite list generated by function *gen* because the elements in a list are only evaluated when required by another function.

Evaluation:

```

x = get(3, gen(1)) = get(3, 1 :: gen(2))
  = 1 :: get(2, gen(2)) = 1 :: get(2, 2::gen(3))
  = 1 :: 2 :: get(1, gen(3)) = 1 :: 2 :: get(1, 3::gen(4))

```

$$= 1 :: 2 :: [3] = [1,2,3]$$

iii. Consider another problem of generating infinite sequence of numbers in increasing order using the following formula.

$$\text{Number} = 2^n * 3^m, \quad \forall n, m \geq 0$$

The list of numbers can be defined informally as:

- 1 is valid number,
- if x is a valid number, then 2x and 3x are also valid numbers.

Thus, an infinite sequence of numbers generated by above definition is 1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, 36 In most of the languages, this is quite complex but using lazy evaluation technique it can be easily expressed.

We define a function *seq* that generates an infinite list of numbers in increasing order stated as above using another function *get_next*. The function *merge* is used in *get_next* that takes two ascending lists and merges them maintaining the ordering and removing duplicates.

```
- fun seq () = get_next [1];
> val seq = fn : unit -> int list
- fun get_next (x :: xs) = x :: get_next (merge ([2*x, 3*x], xs));
> val get_next = fn : int list -> int list
```

Eager evaluation:

```
- val t = get (5, seq()); ← non termination
```

Lazy evaluation:

```
- val t = get (5, seq());
> val t = [1,2,3,4,6] : int list
```

Evaluation:

```
t = get (5, seq ())
= get(5, get_next [1])
= get (5, 1 :: get_next (merge([2,3], [])))
= 1 :: get (4, get_next ([2,3]))
= 1 :: get (4, 2 :: get_next(merge([4,6], [3])))
= 1 :: 2 :: get(3, get_next([3,4,6]))
= 1 :: 2 :: get(3, 3 :: get_next(merge([6,9], [4,6])))
= 1 :: 2 :: 3 :: get (2, get_next ([4,6,9]) )
= 1 :: 2 :: 3 :: get (2, 4 :: get_next (merge([8,12], [6,9])))
= 1 :: 2 :: 3 :: 4 :: get (1, get_next([6, 8, 9, 12]))
= 1 :: 2 :: 3 :: 4 :: get (1, 6 :: get_next(merge ([12, 18],[8, 9, 12])))
= 1 :: 2 :: 3 :: 4 :: 6
= [1,2,3,4,6]
```

iv. Generate a list of prime numbers using Eratosthenes sieve method

The numbers using Eratosthenes sieve method are generated as follows:

- Initially, assume that 2 is a prime number.
- Find prime numbers by removing the non prime numbers from a list of integers generated from 2 onwards.
- Non prime number is one which is a multiple of some previously generated prime number.

Alternatively, we can state the method as follows:

- Get a number from a infinite list generated from 2 onwards and check if it is non prime. If not, then attach this number to the list of primes generated otherwise generate new number. Repeat this method as many time as you want.

```

- fun not_multiple x y = (y mod x <> 0);
> val not_multiple = fn : int -> int -> bool
- fun filter f [] = []
  | filter f (x::xs) = if f(x) then x :: filter f xs else filter f xs;
> val filter = fn : ('a -> bool) -> 'a list -> 'a list
- fun remove x [] = []
  | remove x xs = filter (not_multiple x) xs;
> val remove = fn : int -> int list -> int list
- fun main (x ::xs) = x :: main(remove x xs);
> val main = fn : int list -> int list
- fun primes () = main (gen 2);
> val primes = fn : unit -> int list

```

Eager evaluation:

```

- get(4, primes ()); ← non terminating

```

Lazy evaluation:

```

- get(4, primes ());
> val it = [2,3,5,7] : int list

```

• **Handling of infinite data structures**

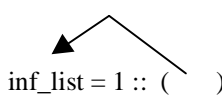
Lazy languages also include lazy constructors, hence provide one of the most important benefits of lazy evaluation i.e., the ability to handle infinite data structures. So far we have seen function and data types being defined recursively. It is very useful if we define infinite data structure like list, trees, infinite series and any infinite mathematical structure and assign them directly to the variables and use them directly in the program. Further, lazy evaluation helps in creating cyclic data structures, which would terminate. The following are recursive definition of some infinite data structures.

i. Define a variable *inf_list* which is an infinite list of 1.

```

- val rec inf_list = 1:: inf_list;
> val rec inf_list = 1:: inf_list : int LIST
- val x = get(10, inf_list);
> val x = [1,1,1,1,1,1,1,1,1,1] : int list

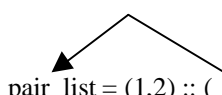
```



```

- val rec pair_list = (1,2) :: pair_list;
> val rec pair_list = (1, 2) :: pair_list : (int * int) LIST
- val y = inf_list equal pair_list;
> val y = false : bool
- val z = get(3, pair_list);
> val z = [(1,2), (1,2), (1,2)] : int list

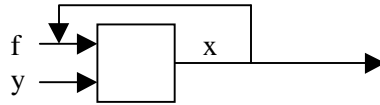
```



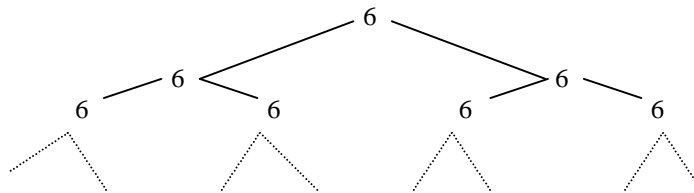
Infinite list can be constructed and easily used in lazy languages as arguments of other functions. Functions *get* and *equal* are defined earlier.

- **Cyclic structures**

- `val rec x = f(x, y);`

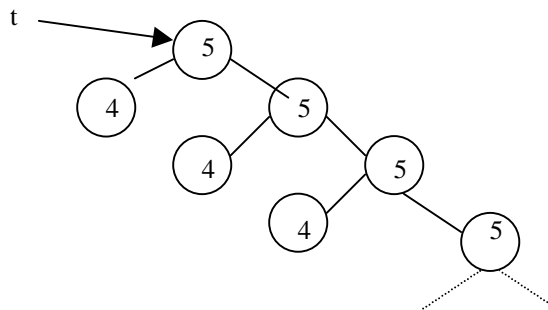


Define a variable `inf_tree` for infinite binary tree with all nodes having value 6.



```
- val rec inf_tree = Node (inf_tree, 6, inf_tree);
> val rec inf_tree = Node (inf_tree, 6, inf_tree) : int BT
- val x = get (5, preorder (inf_tree));
> val x = [6,6,6,6,6] : int list
```

- **Recursive definition of the following infinite binary tree**



```
- val rec t = Node (Node (Null, 4, Null), 5, t);
> val rec t = Node (Node (Null, 4, Null), 5, t) : int BT
```

i. Recursive definition for an infinite sequence of numbers in increasing order using the following formula.

$$\text{Number} = 2^n * 3^m, \quad \forall n, m \geq 0$$

This problem has also been discussed earlier and was solved by using recursive function. Alternatively the problem is stated as:

- i. 1 is valid number,
- ii. if x is a valid number, then $2x$ and $3x$ are also valid numbers.

```
- val rec seq_num =
  1:: merge ((map(times 2) seq_num), (map(times 3) seq_num))
> val rec seq_num = 1::
  merge ((map(times 2) seq_num), (map(times 3) seq_num)):int list
```

The function `map` multiplies 2 or 3 whichever the case might be to the head of recursive list `seq_num` whenever there is a need. It is defined as follows:

```

-      fun    map f []      =      []
          | map f (x :: xs) =      f x :: map f xs ;
>      val map = fn : ('a -> 'b) -> 'a list -> 'b list

```

List is constructed as follows:

```

seq_num      = 1:: merge ((map(times 2) seq_num), (map(times 3) seq_num))
              = 1 :: merge (2 :: (map(times 2) seq_num), 3 :: (map(times 3) seq_num))
              = 1 :: 2 :: merge ((map(times 2) seq_num), 3 :: (map(times 3) seq_num))
              = 1::2::merge (4 :: (map(times 2) seq_num), 3 :: (map(times 3) seq_num))
              = 1::2::3 :: merge (4 :: (map(times 2) seq_num), (map(times 3) seq_num))
              ⋮

```

- **Lazy evaluation for generating solution using numerical techniques**

Writing algorithms for numerical techniques finding solution(s) by successive refinement till the difference between previous solution and the current solution is negligible is more elegant and concise. Newton Raphson is numerical technique for solving equation of the form $f(x) = 0$. It starts with good initial approximate solution and converges rapidly to the solution. Computing square root of a real number is an application of Newton Raphson method. The following equation is solved for finding square root of a real number say, a .

$$\begin{aligned}
 f(x) &= x^2 - a = 0 \\
 x^2 - a &= 0 \\
 x^2 &= a \\
 x &= \sqrt{a}
 \end{aligned}$$

Consider initial solution to be $x_0 = 1.0$ and next solution is obtained by the following formula.

$$x_i = (x_{(i-1)} + a / x_{(i-1)}) / 2.0$$

Termination condition is: $(x_i - x_{(i-1)}) < .000001$

```

-      fun gen (a, n) = n :: gen(a, (n + a / n) / 2.0);
>      val gen = fn : real * real -> real list
-      fun findroot (a) = gen (a, 1.0);
>      val findroot = fn : real -> real list
-      fun terminate(x::y::xs) = if (x-y) < 0.00001 then x else terminate (y :: xs);
>      val terminate = fn : real list -> real
-      fun sqrt (a) = terminate (findroot (a));
>      val sqrt = fn : real -> real

```

A function *sqrt* calls a function *terminate* which is a higher order function. Argument of *terminate* is another function *findroot* that generate a list of possible solutions by using function *gen*. In eager evaluation, *sqrt* is a non terminating function but it terminates for lazy evaluation.

- **Separation of data and control**

The lazy functions can be written without containing operational control as evaluation is demand-driven. Therefore, Laziness can help in separating data from control. It gives a pipeline approach to a program construction. For example, given the following components:

```

-      fun    cube []      = []

```



```

      | cube (n:: ns) = n*n*n :: cube ns;
>   val cube = fn : int list -> int list
-   fun  double [] = []
      | double (n::ns) = n + n :: double ns;
>   val double = fn : int list -> int list
-   fun positive x = if x > 0 then true else false;
>   val positive = fn : int -> bool
-   fun main () = double (cube(filter positive (get_list)));
>   val main = fn : unit -> int list

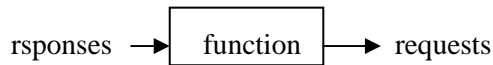
```

We can compose them simply to create a pipeline that filters out positive numbers from a list, cube them and then double them. The function *filters* is defined earlier. The important point is that function *main* has been written without knowing the size of the list. Lazy evaluation has therefore freed the programmer from operational control issues like exiting from a loop in imperative languages. Here *get_list* operates on demand and supplies an element of a list (of unknown size).

This pipeline style generally leads to clearer, more maintainable solutions and prompts the observation that laziness may provide important links in the convergence of functional programming, visual programming and analysis and design methods.

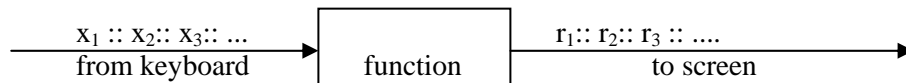
- **Interactive program**

Lazy evaluation forms the basis for effective, elegant interaction. Consider an interaction at the highest level as a function taking responses and returning requests on the demand.



Demand driven evaluation with suspensions can be regarded as a special case of several components of a program working independently and communicating via demand. It gives rise coroutine kind of behaviour whereby each process is driven by demand for some output from another process requiring some input. Processes are suspended as soon as the demand is satisfied but may resume when further demands are placed on them.

A interactive program reads in different values of *X* (requests) and outputs the values (responses) at different stages can easily be expressed as a function which takes a list of values for *X* and produces a list of responses.



In order to generate requests before receiving responses, the program must be lazy with its argument otherwise it would hang forever waiting for responses that haven't yet been requested. Even though there is a large semantic gap between a purely functional program and the outside world. The goal is to provide elegant, purely functional mechanisms for input/output and sequencing. Recent research advances, however, such as the Haskell I/O system [HPW92], have tamed many of these problems and a variety of effective bridges now exist

Simulation of infinite data structures in SML

As already mentioned earlier, evaluation can be delayed in eager languages by using higher order functions. In eager evaluation, only functions can be defined recursively and recursive data structures can not be defined directly because of non termination problem. But infinite data structure can be simulated with eager evaluation by use of higher order functions which expand the structure bit by bit i.e., a data structure containing functions can represent an infinite object such as infinite lists, trees etc.

Lists with lazy evaluation mechanism

In order to avoid unnecessary computation in comparing two lists, one can simulate lazy comparison in SML (in any eager language). We define below the datatype for lazy list whose element is constructed by applying function of the type $(\text{unit} \rightarrow 'a * 'a \text{ lazy_list})$. `E_list` represents an empty list.

```
-      datatype 'a lazy_list = ML_list of (unit ->'a * 'a lazy_list) | E_list;
>      datatype 'a lazy_list = E_list | ML_list of unit -> 'a * 'a lazy_list
```

- *Various functions for lists*

```
-      fun cons x xs = let fun f() = (x, xs) in ML_list f end;
>      val cons = fn : 'a -> 'a lazy_list -> 'a lazy_list
-      fun      head E_list = ~1
              | head (ML_list f) = let val (x, xs) = f() in x end;
>      val head = fn : int lazy_list -> int
-      fun      tail E_list = E_list
              | tail (ML_list f) = let val (x, xs) = f() in xs end;
>      val tail = fn : 'a lazy_list -> 'a lazy_list
-      fun      null E_list = true
              | null (ML_list f) = false;
>      val null = fn : 'a lazy_list -> bool
```

Function for comparing two lists for equality

```
-      infix eq
>      infix eq
-      fun (E_list eq E_list) = true
              | (ML_list f) eq (ML_list g) = let
                                                val (x, xs) = f(); val (y, ys) = g()
                                                in (x = y) andalso (xs eq ys) end
              | ((ML_list f) eq E_list) = false
              | (E_list eq (ML_list f)) = false;
>      val eq = fn : 'a lazy_list * 'a lazy_list -> bool
```

Construction of a list and its operations

```
-      cons 4 E_list;
>      val it = ML_list fn : int lazy_list
-      cons 4 it;
>      val it = ML_list fn : int lazy_list
-      cons 7 it;
>      val it = ML_list fn : int lazy_list
-      val p = cons 8 it;
>      val p = ML_list fn : int lazy_list
```

```

-     head p;
>     val it = 8 : int
-     tail p;
>     val it = ML_list fn : int lazy_list
-     p eq it;
>     val it = false : bool
-     val q = p;
>     val q = ML_list fn : int list1
-     p eq q;
>     val it = true : bool

```

- **Construction of infinite lists:** (list of ones, list of natural numbers etc.)

The same datatype definition is used for constructing infinite lists.

```

-     val inf_ones = let fun f() = (1, ML_list f) in ML_list f end;
>     val inf_ones = ML_list fn : int lazy_list
-     fun gen n = let fun f() = (n, gen (n+1)) in ML_list f end;
>     val gen = fn : int -> int lazy_list
-     val nat = gen 1;
>     val nat = ML_list fn : int lazy_list
-     val from10 = gen 10;
>     val from10 = ML_list fn : int list1

```

Function for getting finite number of elements from infinite list

```

-     fun front n s = get_first(lazy_get n s);
>     val front = fn : int -> 'a lazy_list -> 'a list

```

where *get_first* and *lazy_get* are defined below:

```

-     fun      lazy_get 0 s = ([], s)
              | lazy_get n E_list = ([], E_list)
              | lazy_get n (ML_list f) = let val (x, xs) = f();
                                          val (y, ys) = lazy_get (n-1) xs
                                          in (x::y, xs) end;
>     val lazy_get = fn : int -> 'a lazy_list -> 'a list * 'a lazy_list
-     fun get_first (x, y) = x ;
>     val get_first = fn : 'a * 'b -> 'a

```

Extracting finite values using *front* function

```

-     front 5 inf_ones;
>     val it = [1,1,1,1,1] : int list
-     front 10 inf_ones;
>     val it = [1,1,1,1,1,1,1,1,1,1] : int list
-     front 6 nat;
>     val it = [1,2,3,4,5,6] : int list
-     front 10 from10;
>     val it = [10,11,12,13,14,15,16,17,18,19] : int list

```

Tree comparison using lazy evaluation mechanism

With eager evaluation, earlier function for comparing two trees in preorder gives wasteful computation by getting preorders of both the lists and then comparing these lists using list comparison function. If we combine the actions of flattening binary trees and comparing lists, then in case when elements of both the binary trees in the preorder sequence are not equal, remaining portions of the trees could be avoided for computation.

```
- datatype 'a BT = Null | Node of 'a BT * 'a * 'a BT;
> datatype 'a BT = Node of 'a BT * 'a * 'a BT | Null
```

The function `tree_eq` makes use of another function `stack_compare` which compares values in two stacks each consisting of binary trees. Initially stack contains binary trees. Get top element of both the stacks and compare the value at the root. If they are equal then both the trees are split into sub trees (left and right) and stored in their respective stacks. The process is repeated till the stacks are empty or values do not match. The processing of sub trees when the values do not match is delayed.

```
- fun stack_compare [] [] = true
  | stack_compare [] (a::x) = if a = Null then stack_compare [] x else false
  | stack_compare (a::x) [] = if a = Null then stack_compare x [] else false
  | stack_compare (Null::x) y = stack_compare x y
  | stack_compare x (Null::y) = stack_compare x y
  | stack_compare (Node(l1,d1,r1)::x) (Node(l2,d2,r2)::y) =
    if (d1 = d2) then stack_compare (l1 ::r1::x) (l2::r2::y) else false;
> val stack_compare = fn : "a BT list -> "a BT list -> bool
- fun tree_eq t1 t2 = stack_compare [t1] [t2];
> val tree_eq = fn : "a BT -> "a BT -> bool
```

- Creating binary trees as follows

```
- val t = Node(Node(Node (Null,3,Null),2,Null),1,
               Node(Node(Null,5, Null), 4, Node(Null, 6, Null)));
> val t = Node (Node (Node #,2,Null),1,Node (Node #,4,Node #)) : int BT
- val t1 = Node (Node (Null,1,Null),2,Node(Node(Null,4, Null, 3, Node
      (Node(Null,6,Null), 5, Null)));
> val t1 = Node (Node (Null,2,Null),1,Node (Node #,3,Node #)) : int BT
- val x = Node(Node(Node (Null,2,Null),1,Node(Node(Null,5, Null), 4, Null));
> val x = Node (Node (Null,2,Null),1,Node (Node #,4,Null)) : int BT
- val y = Node (Node (Null,1,Null),2,Node(Null, 3, Null));
> val y = Node (Node (Null,1,Null),2,Node (Null,3,Null)) : int BT
```

- Comparing two binary trees

```
- val p = tree_eq t t1;
> val p = true : bool
- val q = tree_eq x y;
> val q = false : bool
```

Construction of infinite trees and their comparisons

Define datatype for binary tree using function as an argument. The binary tree is either Null or (Node f), where f is a function of type unit -> 'a BT * 'a * 'a BT. Using function one can delay evaluation of a node. Node is evaluated only when function is applied.

```
- datatype 'a BT = Null | Node of (unit -> 'a BT * 'a * 'a BT);
> datatype 'a BT = Node of unit -> 'a BT * 'a * 'a BT | Null
```

One can construct a binary search tree using following *delayed_insert* function defined below:

```

- fun delayed_insert(key, Null) = let fun f() = (Null, key, Null)
                                in Node f end
  | delayed_insert (key, Node f) = let val (left, k, right) = f()
                                in if key < k then let
                                  fun g () = (delayed_insert (key,left), k,right) in Node g end
                                else let fun h() = (left, k, delayed_insert (key, right)) in Node
                                  h end end;
> val insert = fn : int * int BT -> int BT

```

- Functions for traversing binary trees

```

- fun preorder Null = []
  | preorder (Node f) = let val (left, data, right) = f() in
                        [data] @ preorder left @ preorder right end;
> val preorder = fn : 'a BT -> 'a list
- fun inorder Null = []
  | inorder (Node f) = let val (left, data, right) = f() in
                        inorder left @ [data] @ inorder right end;
> val inorder = fn : 'a BT -> 'a list
- fun postorder Null = []
  | postorder (Node f) = let val (left, data, right) = f() in
                          postorder left @ postorder right @ [data] end;
> val postorder = fn : 'a BT -> 'a list

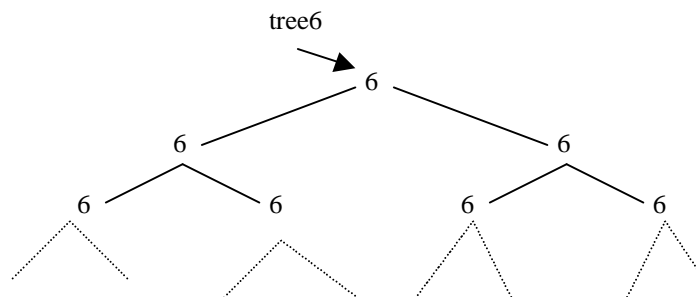
```

Definitions of various infinite binary trees

```

- val tree6 = let fun f () = (Node f, 6, Node f) in Node f end;
> val tree6 = Node fn : int BT

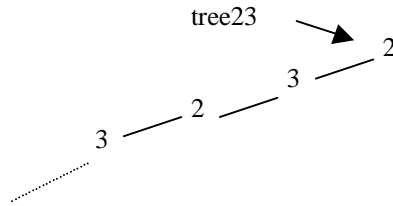
```



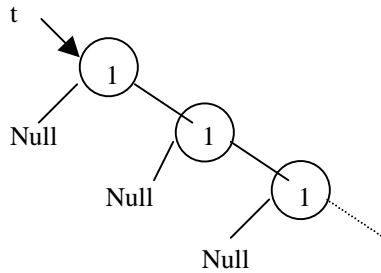
```

- val tree23 = let fun g () = (Node f, 2, Null) and f () = (Node g, 3, Null)
                  in Node g end;
> val tree23 = Node fn : int BT

```



```
- val t = let fun f () = (Null, 1, Node f) in Node f end;
> val t = Node fn : int BT
```



Function for getting fixed number of values from infinite binary tree in preorder sequence.

```
- fun l_get 0 s = ([], s)
  | l_get n Null = ([], Null)
  | l_get n (Node f) = let val (l,d,r) = f();
                       val (d1, xs) = if null l then l_get (n-1) r
                                       else l_get (n-1) l in (d::d1, l) end ;
> val l_get = fn : int -> 'a BT -> 'a list * 'a BT
- fun fst (x, y) = x ;
> val fst = fn : 'a * 'b -> 'a
- fun front n s = fst(l_get n s);
> val front = fn : int -> 'a BT -> 'a list
- front 6 t;
> val it = [1,1,1,1,1,1] : int list
- front 5 tree6;
> val it = [6,6,6,6,6] : int list
- front 8 tree23;
> val it = [2,3,2,3,2,3,2,3] : int list
```

- Comparing two infinite binary tree

```
- fun delayed_stack_eq [] [] = true
  | delayed_stack_eq [] ((Node f)::x) = false
  | delayed_stack_eq ((Node f)::x) [] = false
  | delayed_stack_eq (Null::x) y = delayed_stack_eq x y
  | delayed_stack_eq x (Null::y) = delayed_stack_eq x y
  | delayed_stack_eq ((Node f)::x) ((Node g)::y) = let
      val (l1,d1,r1) = f(); val (l2,d2,r2) = g() in if
      (d1 = d2) then delayed_stack_eq (l1::r1::x)
      (l2::r2::y)
      else false end;
> val equal = fn : 'a BT list -> 'a BT list -> bool
- fun tree_equality t1 t2 = delayed_stack_eq [t1] [t2];
```

```
> val tree_equality = fn : 'a BT -> 'a BT -> bool
- tree_equality t tree6;
> val it = false : bool
```

The function *tree_equality* gives rise false value as soon as it does not match two corresponding values in binary trees but goes into non termination if trees are equal.

Disadvantages of Lazy evaluation

- Although functional languages allow user to ignore the underlying mechanism that manages data, the system must still manage that memory. When laziness is introduced, the situation becomes more complex, and there is conflicting evidence as to whether it helps or hinders memory management. Laziness undoubtedly adds a memory management overhead, although some case studies identify situations when memory management allows lazy functional languages to beat not only eager ones, but imperative languages as well (see [KoOt93]).
- Traditionally, lazy functional languages executes slower than eager ones. This is due to overheads required to handle closures or equivalent constructs. But in some cases, it largely depends on the nature of algorithm. A well-designed lazy programs execute faster than eager ones. Laziness can be exploited to make a simple dynamic-programming algorithm run quickly as explained in [All92]. Execution efficiency can also be viewed as trade off between fast eager code against better engineered lazy code from software engineering point of view.
- Garbage collection is a problem in functional programs which may effect run-time performance of functional programs in terms of both time and space.
- Lazy evaluation computational model is harder to interpret and analyze.