or

d) if `<function value>` is not a function
then lazy evaluate `<argument expression>`to `<argument value>`
and replace all occurences of `(<function expression> <argument expression>)`$_i$
with `(<function value> <argument value>)`
and return `(<function value> <argument value>)`

8.11. Exercises

1) Evaluate the following expressions using normal order, applicative order and lazy evaluation. Explain any differences in the final result and the number of reductions in each case:

```
i) λs.(s s) (λf.λa.(f a) λx.x λy.y)
ii) λx.λy.x λx.x (λs.(s s) λs.(s s))
iii) λa.(a a) (λf.λs.(f (s s)) λx.x)
```

9. FUNCTIONAL PROGRAMMING IN STANDARD ML

9.1. Introduction

ML(Meta Language) is a general purpose language with a powerful functional subset. It is used mainly as a design and implementation tool for computing theory based research and development. It is also used as a teaching language. ML is strongly typed with compile time type checking. Function calls are evaluated in applicative order.

ML originated in the mid 1970's as a language for building proofs in Robin Milner's LCF(Logic for Computable Functions) computer assisted formal reasoning system. SML(Standard ML) was developed in the early 1980's from ML with extensions from the Hope functional language. SML is one of the first programming languages to be based on well defined theoretical foundations.

We won't give a full presentation of SML. Instead, we will concentrate on how SML relates to our approach to functional programming.

SML is defined in terms of a very simple **bare language** which is overlaid with standard **derived forms** to provide a higher level syntax. Here, we will just use these derived forms.

We will explain SML with examples. As the symbol `->` is used in SML to represent the types of functions, we will follow SML system usage and show the result of evaluating an expression as:

```
- <expression>;
> <result>
```

9.2. Types

Types are central to SML. Every object and construct is typed. Unlike Pascal, types need not be made explicit but they must be capable of being deduced statically from a program.

SML provides several standard types, for example for booleans, integers, strings, lists and tuples which we will look at below. SML also has a variety of mechanisms for defining new types but we won't consider these here.

When representing objects, SML always displays types along with values:

```
<value> : <type>
```

For objects which do not have printable value representations SML will still display the types. In particular, function values are displayed as:

```
fn : <type>
```

Types are described by type expressions. We will see how these are constructed as we discuss specific types.

9.3. Basic types - booleans, integers and strings

The type expression for a basic type is the type’s **identifier.**

The boolean type has identifier:

```
bool
```

and values:

```
true false
```

For example:

```
- true;
> true : bool
```

The integer type has identifier:

```
int
```

with positive and negative integer values, for example:

```
- 42;
> 42 : int

- ˜84
˜84 : int
```

Note the use of ˜ as the negative sign.

The string type has identifier:

```
string
```

String values are character sequences within

```
- "Is this a string?";
> "Is this a string?" : string
```

9.4. Lists

In SML, unlike LISP and our approach to λ calculus, a list must contain elements of the same type and end with the empty list. Thus, lists cannot be used to represent records with different type fields.

Lists are written as `,` separated element sequences within `[` and `].` For example:

```
[1,4,9,16,25]

["ant","beetle","caterpillar","dragonfly","earwig"]
```

There is an implied empty list at the end of a list.

```
[]
```

is the empty list.

The type expression for a list depends on the element type:

```
<element type> list
```

The first example above:

```
- [1,4,9,16,25];
> [1,4,9,16,25] : int list
```

is a list of integers. The second example above:

```
- ["ant","beetle","caterpillar","dragonfly","earwig"];
> ["ant","beetle","caterpillar","dragonfly","earwig"] : string list
```

is a list of strings.

Lists may be nested, for example:

```
- [[1,1],[2,8],[3,27],[4,64],[5,125]];
> [[1,1],[2,8],[3,27],[4,64],[5,125]] : (int list) list
```

is a list of integer lists.

Note the use of `(` and `)` to structure the type expression.

9.5. Tuples

An ML **tuple,** like a Pascal `RECORD,` is fixed length sequence of elements of different types, unlike a list which is variable length sequence of elements of the same type. Tuples are written as `,` separated sequences within `(` and `).`

For example, we might represent a stock control record from chapter 7 as:

```
("VDUs",250,120)
```

Tuples may be nested. For example, we might represent a telephone directory entry from chapter 7 as:

```
(("Anna","Able"),"Accounts",101)
```

A tuple’s type is represented by its elements’ types separated by `*s:`

```
<element1 type> * <element2 type> * ...
```

For example:

```
- ("VDUs",250,120);
> ("VDUs",250,120) : string * int * int
```

is a tuple consisting of two integers and a string. For example:

```
- (("Anna","Able"),"Accounts",101);
> (("Anna","Able"),"Accounts",101) : (string * string) * string * int
```

is a tuple consisting of a tuple consisting of two strings, a string and an integer.

These may be used to build tuple lists, for example a stock control list:

```
- [("VDUs",250,120),
     ("mice",155,170),
     ("printers",43,20)];
> [("VDUs",250,120),
   ("mice",155,170),
   ("printers",43,20)] : (string * int * int) list
```

or a telephone directory:

```
- [(("Anna","Able"),"Accounts",101),
     (("Betty","Baker"),"Boiler room",102),
     (("Cloe","Charlie"),"Customer orders",103)];
> [(("Anna","Able"],"Accounts",101),
   (("Betty","Baker"),"Boiler room",102),
   (("Cloe","Charlie"),"Customer orders",103)] : ((string * string) *
                                                    string * int) list
```

Note that if a tuple is defined with expression elements then those expressions are evaluated from left to right. Thus, as tuples are used to specify bound variables for uncurried functions, such functions have a defined actual parameter evaluation order.

9.6. Function types and expressions

A function uses values in an argument **domain** to produce a final value in a result **range.** In SML, a function's type is characterised by its domain and range types:

```
fn : <domain type> -> <range type>
```

Note the use of `->` to indicate a function's domain/range mapping.

Tuples are normally used to enable uncurried functions with multiple bound variables.

In SML, as in λ calculus and LISP, expressions are usually based on prefix notation function applications with the function preceding the arguments:

```
<function expression> <argument expression>
```

Function applications are evaluated in applicative order.

Note that function applications need not be explicitly bracketed but brackets should be used round arguments to avoid ambiguity.

SML enables uncurried binary functions to be used as infix operators so the function name may appear in between the two arguments. They are then typed as if they had tuples for arguments. We won't consider this further here.

Similarly, many standard binary functions are provided as infix operators. They may be treated as prefix functions on tuples by preceding them with:

```
op
```

We will look at this in more detail when we consider standard functions.

9.7. Boolean standard functions

The boolean negation function:

```
not
```

returns the negation of its boolean argument, for example:

```
- not true;
> false : bool
```

Thus, `not`’s type is:

```
- not;
> fn : bool -> bool
```

Conjunction and disjunction are provided through the sequential infix operators:

```
andalso orelse
```

in the derived syntax, for example:

```
- true orelse false;
> true : bool

- true andalso false;
> false : bool
```

SML systems may not be able to display these operators’ types but they are effectively:

```
fn : (bool * bool) -> bool
```

as they both take two boolean arguments, which are treated as a:

```
bool * bool
```

tuple for infix syntax, and return a boolean result.

9.8. Numeric standard functions and operator overloading

SML provides real numbers as well as integers. However, as in many other languages, the same operators are used for both even though they are distinct types. This use of the same operator with different types is known as **operator overloading**.

The addition, subtraction and multiplication infix operators are:

```
+ -  *
```

SML systems may not display their types because they are overloaded. SML literature uses the invented type:

```
num
```

to indicate both integer and real so these operators types might be:

```
fn : (num * num) -> num
```

as they take two numeric arguments, with infix syntax for a tuple, and return a numeric result.

Note that for each operator both arguments must be the same type.

The infix operator:

```
div
```

is for integer division. We can use `op` to convert it to prefix form to display its type:

```
- op div;
> fn : (int * int) -> int
```

Arithmetic expressions are built from these operators with the brackets `(` and `),` for example:

```
- 6 * 7 div (7 - 4) + 28;
> 42 : int
```

Note that there is no strict bracketing. The usual precedence:

```
( )
```

before:

```
div *
```

before:

```
+ -
```

applies.

The numeric negation operator is:

```
~
```

again with effective type:

```
fn : num -> num
```

as it is overloaded for use with integers and reals.

9.9. String standard functions

The binary infix operator:

```
^
```

concatenates two strings together:

```
- op ^;
> fn : (string * string) -> string
```

For example:

```
- "Happy"^" birthday!";
> "Happy birthday!" : string
```

The operator:

```
size
```

returns the size of a string:

```
- size;
> fn : string -> int
```

For example:

```
- size "hello";
> 5 : int
```

Standard functions for turning strings into string lists are discussed below.

9.10. List standard functions

In SML, list operations apply to lists of any types. In SML, an unknown type is denoted by a single letter name preceded by a prime - `’,` for example:

```
’a ’b ’c
```

Thus, we can refer to a list of arbitrary typed objects as having type:

```
’a list
```

In SML, lists are accessed by the head and tail operators:

```
hd tl
```

The head operator returns the head object with type:

```
’a
```

from an arbitrary typed list. Thus, `hd` is of type:

```
- hd;
> fn : (’a list) -> ’a
```

For example:

```
- hd [1,2,3,4,5];
> 1 : int
```

Similarly, the tail operator returns the tail with type:

```
’a list
```

from an arbitrary type list. Thus, `tl` is of type:

```
    - tl;
    > fn : ('a list) -> ('a list)
```

For example:

```
    - tl ["alpha","beta","gamma","delta","epsilon"];
    > ["beta","gamma","delta","epsilon"] : string list
```

The infix list concatenation operator is:

```
    ::
```

Given an object and a list of the same type of object, `::` returns a new list with the object in the head and the object list in the tail. Thus, `::` has type:

```
    - op ::;
    > ('a * ('a list)) -> ('a list)
```

For example:

```
    - 0::[1,2,3,4,5];
    > [0,1,2,3,4,5] : int list
```

The operators `hd, tl` and `::` are said to be **polymorphic** because they apply to a list of any type of object. We will look at polymorphism in slightly more detail later.

9.11. Characters, strings and lists

SML does not provide a separate character type. Instead, a character is a one letter string.

The standard function

```
    ord
```

converts a single character string to the equivalent ASCII code:

```
    - ord;
    > fn : string -> int
```

For example:

```
    - ord "a";
    > 97 : int
```

Similarly, the standard function

```
    chr
```

converts an integer ASCII value into the equivalent single character string:

```
    - chr;
    > fn : int -> string
```

For example:

```
    - chr 54;
    > "6" : string
```

In order to access the individual characters making up a string it must be unpacked into a list of single character strings. The standard function:

```
explode
```

does this:

```
- explode;
> fn : string -> (string list)
```

For example:

```
- explode "hello";
> ["h","e","l","l",’o"] : string list
```

Similarly, the standard function:

```
implode
```

converts a list of strings to a single string:

```
- implode;
> fn : (string list) -> string
```

For example:

```
- implode ["Time ","for ","tea?"];
> "Time for tea?" : string
```

Note that `implode` will join strings of any length.

9.12. Comparison operators

SML provides a variety of overloaded infix comparison operators. Equality and inequality are tested with:

```
= <>
```

and may be used with booleans, integers, strings, lists and tuples.

The less than, less than or equal, greater than and greater than or equal operators:

```
< <= >= >
```

may be used with numbers and strings. For strings, they test for alphabetic order, for example:

```
- "haggis" < "oatcake";
> true : bool
```

SML systems may not display these operators’ types because they are overloaded.

For all these operators, both arguments must be of the same type.

9.13. Functions

Functions have the form:

```
fn <bound variables> => <expression>
```

A bound variable is known as an **alphabetic identifier** and consists of one or more letters, digits and _s starting with a letter, for example:

```
oxymoron Home_on_the_range Highway61
```

A function’s bound variable may be a single bound variable or a tuple of bound variable elements.

For example:

```
- fn x => x+1;
> fn : int -> int
```

increments its argument.

Note that SML deduces that the domain and range are `int` because `+` is used with the `int` argument `1.`

For example:

```
- fn x => fn y => not (x orelse y);
> fn : bool -> (bool -> bool)
```

is the boolean implication function.

Note that `orelse` has a boolean tuple domain so `x` and `y` must both be boolean. Similarly, `not` returns a boolean so the inner function:

```
fn y => not (x orelse y)
```

has type:

```
bool -> bool
```

Hence, the whole function has type:

```
fn : bool -> (bool -> bool)
```

This might have been written with a tuple domain:

```
- fn (x,y) => not (x orelse y);
> fn : (bool * bool) -> bool
```

9.14. Making bound variables’ types explicit

Suppose we try to define a squaring function:

```
fn x => x*x
```

Because `*` is overloaded, SML cannot deduce `x`’s type and will reject this function.

Domain types may be made explicit by following each bound variable with its type. Thus for a single bound variable:

```
(<bound variable> : <type>)
```

is used. For example, an integer squaring function may be defined by:

```
- fn (x:int) => x*x;
> fn : int -> int
```

For a tuple of bound variables:

```
(<bound variable1> : <type1>, <bound variable2> : <type2>, ... )
```

is used. For example, we might define the sum of squares function as:

```
- fn (x:int,y:int) => x*x+y*y;
> fn : (int * int) -> int
```

It is thought to be ‘good practise’ to make all bound variables’ types explicit. This is supposed to make it easier to read functions and to ensure that types are consistent. However, without care, type expressions can become unmanageably long. SML provides ways to name complex types which we will consider in a later section.

9.15. Definitions

Global definitions may be established with:

```
val <name> = <expression>
```

For example:

```
- val sq = fn (x:int) => x*x;
> val sq = fn : int -> int

- val sum_sq = fn (x:int,y:int) => x*x+y*y;
> val sum_sq = fn : (int * int) -> int
```

Note that the SML system acknowledges the definition by displaying the defined name and the expression’s value and/or type.

Defined names may be used in subsequent expressions, for example:

```
- sq 3;
> 9 : int
```

and subsequent definitions, for example:

```
- val sum_sq = fn (x:int,y:int) => (sq x)+(sq y)
> val sum_sq = fn : (int * int) -> int
```

9.16. Conditional expressions

The SML conditional expression has the form:

```
if <expression1>
then <expression2>
else <expression3>
```

The first expression must return a boolean and the option expressions `<expression2>` and `<expression3>` must have the same type.

For example, to find the larger of two integers:

```
- val max = fn (x:int,y:int) => if x>y
                                then x
                                else y;
> val max = fn : (int * int) -> int
```

For example, to define sequential boolean implication:

```
- val imp = fn (x,y) => if x
                        then y
                        else true;
> val imp = fn : (bool * bool) -> bool
```

9.17. Recursion and function definitions

To define recursive functions, the defined name is preceded by:

```
rec
```

For example, to find the length of an integer list:

```
- val rec length = fn (l:int list) => if l = []
                                      then 0
                                      else 1+(length (tl l))
> val length = fn : (int list) -> int
```

As with our λ calculus notation there is a shortened form for function definitions. Instead of `val`:

```
fun
```

is used to introduce the definition, the `fn` is dropped, the bound variables are moved to the left of the `=` and the `=>` is dropped. For recursive definitions, the `rec` is dropped. Thus:

```
fun <name> <bound variables> = <expression> ==

val rec <name> = fn <bound variables> => <expression>
```

For example, to square all the values in an integer list:

```
- fun squarel (l:int list) =
   if l=[]
   then []
   else ((hd l)*(hd l))::(squarel (tl l));
> fun squarel = fn : (int list) -> (int list)
```

For example, to insert a string into an ordered string list:

```
- fun sinsert (s:string,l:string list) =
   if l = []
   then [s]
   else
    if s < (hd l)
    then s::l
    else (hd l)::(sinsert (s,(tl l)));
> val sinsert = fn : (string * (string list)) -> (string list)
```

9.18. Tuple selection

Tuple elements are selected by defining functions with appropriate bound variable tuples. For example, to select the name, department and ’phone number from a telephone directory entry tuple:

```
- fun tname (n:(string * string),d:string,p:int) = n;
> val tname = fn : ((string * string) * string * int) -> (string * string)

- fun tdept (n:(string * string),d:string,p:int) = d;
> val tdept = fn : ((string * string) * string * int) -> string

- fun tno (n:(string * string),d:string,p:int) = p;
> val tno = fn : ((string * string) * string * int) -> int
```

To avoid writing out bound variables which are not used in the function body, SML provides the **wild card** variable:

```
_
```

which behaves like a nameless variable of arbitrary type. For example, we could rewrite the above examples as:

```
- fun tname (n:(string * string),_,_) = n;
> val tname = fn : ((string * string) * ’a * ’b) -> (string * string)

- tname (("Anna","Able"),"Accounts",123);
> ("Anna","Able") : (string * string)

- fun tdept (_,d:string,_) = d;
> val tdept = fn : (’a * string *’b) -> string

- tdept (("Anna","Able"),"Accounts",123);
> "Accounts" : string

- fun tno (_,_,p:int) = p;
> val tno = fn : (’a * ’b * int) -> int

- tno (("Anna","Able"),"Accounts",123);
> 123 : int
```

Note that SML uses `’a` and `’b` to stand for possibly distinct unknown types.

For nested tuple selection, nested bound variable tuples are used. For example, to select the forename and surname from a telephone directory entry:

```
- fun fname ((f:string,_),_,_) = f;
> val fname = fn : ((string * ’a) * ’b * ’c) -> string

- fname (("Anna","Able"),"Accounts",123);
> "Anna" : string

- fun sname ((_,s:string),_,_) = s;
> val fname = fn : ((’a * string) * ’b * ’c) -> string

- sname (("Anna","Able"),"Accounts",123);
> "Able" : string
```

9.19. Pattern matching

SML functions may be defined with bound variable patterns using constants and constructors as well as variables. For example, the head and tail selector functions for integer lists might be defined by:

```
- fun ihd ((h:int)::(t:int list)) = h;
> val ihd = fn : (int list) -> int

- fun itl ((h:int)::(t:int list)) = t;
> val itl = fn : (int list) -> (int list)
```

Note the use of the bound variable pattern:

```
((h:int)::(t:int list))
```

with the list constructor `::`.

Note that this function will crash with an empty list argument as the pattern match will fail.

It is common SML practise to use case style function definitions with pattern matching rather than conditional expressions in a function’s body. These are known as **clausal form** definitions. The general form is:

```
fun <name> <pattern1> = <expression1> |
    <name> <pattern2> = <expression2> |
    ...
    <name> <patternN> = <expressionN>
```

Here, each:

```
<name> <patternI> = <expressionI>
```

defines a case.

Note that the order of the cases is significant.

When a case defined function is applied to an argument, each pattern is matched against the argument in turn, from first to last, until one succeeds. The value of the corresponding expression is then returned.

For example, we might construct a function to return the capital of a Scandinavian country as a sequence of constant cases:

```
- fun capital "Denmark" = "Copenhagen" |
      capital "Finland" = "Helsinki" |
      capital "Norway" = "Oslo" |
      capital "Sweden" = "Stockholm" |
      capital _ = "not in Scandinavia";
> val capital = fn : string -> string
```

Here, an argument is compared with constants until the last case where it is matched with the wild card variable.

For example, we might redefine the integer list length function in terms of a base case for the empty list and a recursive case for a non-empty list:

```
- fun length [] = 0 |
      length (_::(t:int list)) = 1+(length t);
> val length = fn : (int list) -> int
```

Here an argument is compared with the empty list in the first case or split into its head and tail in the second. The head is matched with the wild card variable and lost.

For example, we might generate a list of the first `n` cubes with a base case for when `n` is `0` and a recursive case for positive `n:`

```
- fun cubes 0 = [0] |
      cubes (n:int) = (n*n*n)::(cubes (n-1));
> val cubes = fn : int -> (int list)
```

Here, an argument is compared with `0` in the first case or associated with the bound variable `n` in the second.

For example, we might find the `i`th element in a string list with a base case which fails for an empty list, a base case which returns the head of the list when `i` is `0` and a recursive case for positive `i` with a non empty list:

```
- fun sifind _ [] = "can’t find it" |
      sifind 0 ((h:string)::_) = h |
      sifind (i:int) (_::(t:string list)) = sifind (i-1) t;
> val sfiind = fn : int -> ((string list) -> string)
```

Here, the integer argument is matched with the wild card variable in the first case, compared with `0` in the second and associated with the bound variable `i` in the third. Similarly, the list argument is compared with the empty list in the first case and split into its head and tail in the second and third. In the second case, the tail is matched with with the wild ard variable and lost. In the third case, the head is matched with the wild card variable and lost.

Note that this is a curried function.

Patterns may also be used to specify nameless functions.

9.20. Local definitions

SML uses the `let ... in ...` notation for local definitions:

```
let val <name> = <expression1>
in <expression2>
end
```

This evaluates `<expression2>` with `<name>` associated with `<expression1>`.

For function definitions:

```
let fun <name> <pattern> = <expression1>
in <expression2>
end
```

and the corresponding case form is used.

For example, to sort a list of integers with a local insertion function:

```
- fun sort [] = [] |
      sort ((h:int)::(t:int list)) =
       let fun insert (i:int) [] = [i] |
               insert (i:int) ((h:int)::(t:int list)) =
                if i<h
                then i::h::t
                else h::(insert i t)
       in insert h (sort t)
       end;
> val sort = fn : (int list) -> (int list)
```

9.21. Type expressions and abbreviated types

We will now be a bit more formal about types in SML. We specify a variable's type with a **type expression.** Type expressions are built from **type constructors** like `int,` `real,` `string` and `list.` So far, a type expression may be a single type constructor or a type variable or a function type or a product type or a bracketed type expression or a type variable preceding a type constructor.

SML also enables the use of *abbreviated types* to name type expressions. A name may be associated with a type expression using a *type binding* of the form:

```
type <abbreviation> = <type expression>
```

The `<abbreviation>` is an identifier which may be used in subsequent type expressions.

For example, in the telephone directory example we might use abbreviations to simplify the types used in a directory entry:

```
- type forename = string;
> type forename = string

- type surname = string;
> type surname = string

- type person = forename * surname;
> type person = forename * surname

- type department = string;
> type department = string

- type extension = int;
> type extension = int

- type entry = person * department * extension;
> type entry = person * department * extension
```

New type constructors may be used in subsequent expressions in the same way as predefined types.

Note that a new type constructor is syntactically equivalent to its defining expression. Thus, if we define:

```
- type whole_numb = int;
> type whole_numb = int

- type integer = int;
> type integer = int
```

then values of type `whole_numb`, `integer` and `int` may be used in the same places without causing type errors. This form of type binding just disguises a longer type expression.

9.22. Type variables and polymorphism

SML, like Pascal, is strongly typed. All types must be determinable by static analysis of a program. Pascal is particularly restrictive because there is no means of referring to a type in general. SML, however, allow generalisation through the use of type variables in type expressions where particular types are not significant.

A type variable starts with a `’` and usually has only one letter, for example:

```
’a ’b ’c
```

We have already seen the use of type variables to describe the standard list functions’ types and the use of the wild card variable.

With strong typing but without type variables, generalised functions cannot be described. In Pascal, for example, it is not possible to write general purpose procedures or functions to process arrays of arbitrary types. In SML, though, the `list` type is generalised through a type variable to be element type independent.

Above, we described a variety of functions with specific types. Let us now look at how we can use type variables to provide more general definitions. For example, the head and tail list selector functions might be defined as:

```
- fun hd (h::t) = h;
> val hd = fn : (’a list) -> ’a

- fun tl (h::t) = t;
> val tl = fn : (’a list) -> (’a list)
```

Here, in the pattern:

```
(h::t)
```

there are no explicit types. SML ‘knows’ that `::` is a constructor for lists of any type element provided the head and tail element type have the same type. Thus, if `::` is:

```
(’a * (’a list)) -> (’a list)
```

then `h` must be `’a` and `’t` must be `’a list`. We do not need to specify types here because list construction and selection is type independent.

Note that we could have use a wild card variable for `t` in `hd` and for `h` in `tl.`

For example, we can define general functions to select the elements of a three place tuple:

```
- fun first (x,y,z) = x;
> val first = fn : (’a * ’b * ’c) -> ’a

- fun second (x,y,z) = y
> val second = fn : (’a * ’b * ’c) -> ’b

- fun third (x,y,z) = z
> val third = fn : (’a * ’b * ’c) -> ’c
```

Here in the pattern:

```
(a,b,c)
```

there are no explicit types. Hence, SML assigns the types `’a` to `x,` `’b` to `y` and `’c` to `z.` Here, the element types are not significant. For selection, all that matters is their relative positions within the tuple.

Note that we could have used wild cards for `y` and `z` in `first,` for `x` and `z` in `second,` and for `x` and `y` in `third.`

For example, we can define a general purpose list length function:

```
- fun length [] = 0 |
      length (h::t) = 1+(length t);
> val length = fn : ('a list) -> int
```

There are no explicit types in the pattern:

```
(h::t)
```

and this pattern is consistent if `h` is `'a` and `t` is `'a list` as `(h::t)` is then `'a list`. This is also consistent with the use of `t` as the argument in the recursive call to `length.` Here again, the element types are irrelevant for purely structural manipulations.

This approach can also be used to define type independent functions which are later made type specific.

For example, we might try and define a general purpose list insertion function as:

```
- fun insert i [] = [i] |
      insert i (h::t) =
       if i<h
       then i::h::t
       else h::(insert i t);
```

but this is incorrect although the bound variable typing is consistent if `i` and `h` are both `'a` and `t` is an `'a list`. The problem lies with the use of the comparison operator `<.` This is overloaded so its arguments' types must be made explicit. We could get round this by abstracting for the comparison:

```
- fun insert _ i [] = [i] |
      insert comp i (h::t) =
       if comp (i,h)
       then i::h::t
       else h::(insert comp i t);
> val insert = (('a * 'a) -> bool) -> ('a -> (('a list) -> ('a list)))
```

Here, `comp` needs an `('a * 'a)` argument to be consistent in `insert` and must return a `bool` to satisfy its use in the `if.`

Now, different typed comparison functions may be used to construct different typed insertion functions. For example, we could construct a string insertion function through partial application by passing a string comparison function:

```
fn (s1:string,s2:string) => s1<s2
```

to `insert:`

```
- val sinsert = insert (fn (s1:string,s2:string) => s1<s2);
> val sinsert = fn : string -> ((string list) -> (string list))
```

Here, the comparison function is:

```
(string * string) -> bool
```

so `'a` must be `string` in the rest of the function's type.

For example, we could also construct a integer insertion function through partial application by passing an integer comparison function:

```
fn (i1:integer,i2:integer) => i1<i2
```

to `insert:`

```
- val iinsert = insert (fn (i1:int,i2:int) => i1<i2);
> val iinsert = fn : int -> ((int list) -> (int list))
```

Now, the comparison function is:

```
(int * int) -> bool
```

so `'a` must be `int` in the rest of the function.

Functions which are defined for generalised types are said to be **polymorphic** because they have **many forms**. Polymorphic typing gives substantial power to a programming language and a great deal of research and development has gone into its theory and practise. There are several forms of polymorphism. Strachey distinguishes ‘ad-hoc’ polymorphism through operator overloading from ‘parameterised’ polymorphism through abstraction over types. Cardelli distinguishes ‘explicit’ parameterised polymorphism where the types are themselves objects from the weaker ‘implicit’ polymorphism where type variables are used in type expressions but types are not themselves objects, as in SML. Milner first made type polymorphism in functional languages practical with his early ML for LCF. This introduced polymorphic type checking where types are deduced from type expressions and variable use. Hope and Miranda also have implicit parameterised polymorphic type checking.

9.23. New types

A new **concrete** type may be introduced by a **datatype binding.** This is used to define a new type’s constituent values recursively by

i) listing base values explicitly

ii) defining structured values in terms of base values and other structured values.

The binding introduces new **type constructors** which are used to build new values of that datatype. They are also used to identify and manipulate such values.

At simplest, a datatype binding takes the form:

```
datatype <constructor> = <constructor1> |
                         <constructor2> |
                         ...
                         <constructorN>
```

which defines the base values of type `<constructor>`, an identifier, to be the type constructor identifiers `<constructor1>` or `<constructor2>` etc.

For example, we could define the type `bool` with:

```
- datatype bool = true | false;
> datatype bool = true | false
  con true = true : bool
  con false = false : bool
```

This defines the constructors `true` and `false` for the new type `bool.` In effect, this specifies that an object of type `bool` may have either the value `true` or the value `false.` An equality test for `bool` is also defined so that the values `true` and `false` may be tested explicitly.

For example, a traffic light goes through the stages red, red and amber, green, amber and back to red:

```
- datatype traffic_light = red | red_amber | green | amber;
> datatype traffic_light = red | red_amber | green | amber
  con red = red : traffic_light
  con red_amber = red_amber : traffic_light
  con green = green : traffic_light
  con amber = amber : traffic_light
```

This defines the data type `traffic_light` with the constructors `red`, `red_amber`, `green` and effect,`amber.` In `red`, `red_amber`, `green` and `amber` are the values of the new type: `traffic_light`. An equality test for `traffic_light` values is also defined.

For example, we can now define a function to change a traffic light from one stage to the next:

```
- fun change red       = red_amber |
      change red_amber = green     |
      change green     = amber     |
      change amber     = red;
> val change = fn : traffic_light -> traffic_light
```

The datatype binding is also used to define structured concrete types. The binding form is extended to:

```
datatype <constructor> = <constructor1> of <type expression1> |
                         <constructor2> of <type expression2> |
                         ...
                         <constructorN> of <type expressionN>
```

where the extension `of <type expression>` is optional. This specifies a new type:

```
<constructor>
```

with values of the form:

```
<constructor1>(<value for <type expression1>>)
<constructor2>(<value for <type expression2>>)
etc.
```

`<constructor1>`, `<constructor2>`, etc are functions which build structured values of type `<constructor>`.

For example, integer lists might be defined by:

```
- datatype intlist = intnil | intcons of int * intlist;
> datatype intlist = intnil | intcons of int * intlist
  con intnil = intnil : intlist
  con intcons = fn : (int * intlist) -> intlist
```

Now, `intnil` is an `intlist` and values of the form `intcons(<int value>, <intlist value>)` are `intlist.` That is, `intcons` is a function which builds an `intlist` from an `int` and an `intlist.` For example:

```
- intcons(1,intnil);
> intcons(1,intnil) : intlist

- intcons(1,intcons(2,intnil));
> intcons(1,intcons(2,intnil)) : intlist

- intcons(1,intcons(2,intcons(3,intnil)));
> intcons(1,intcons(2,intcons(3,intnil))) : intlist
```

A datatype constructor may be preceded by a type variable to parameterise the datatype. For example, SML lists might be defined by:

```
- datatype ’a list = lnil | cons of ’a * (’a list);
> datatype ’a list = lnil | cons of ’a * (’a list)
  con lnil : ’a list
  con cons = fn : (’a * ’a list) -> (’a list)
```

This defines `cons` as a prefix constructor.

Type variables in datatype definitions may be set to other types in subsequent type expressions. For example, in:

```
- type intlist = int list
> type intlist = int list
```

the type variable `’a` is set to `int` to use `intlist` to name an integer list type.

SML systems will also deduce the intended type when the constructor from a parameterised data type is used with consistent values Thus, the following are all `string lists`:

```
- cons("ant",lnil);
> cons("ant",lnil) : string list

- cons("ant",cons("bee",lnil));
> cons("ant",cons("bee",lnil)) : string list

- cons("ant",cons("bee",cons("caterpiller",lnil)));
> cons("ant",cons("bee",cons("caterpiller",lnil))) : string list
```

Structured datatype values may also be used in patterns with typed variables and constructors in the tuple following the datatype constructor. It is usual to have separate patterened definitions for base and for structured values.

For example, to sum the elements of an `intlist:`

```
- fun sum intnil = 0 |
      sum (intcons(x:int,y:intlist)) = x + (sum y);
> val sum = fn : intlist -> int

- sum (intcons(9,intcons(8,intcons(7,intnil))));
> 24 : int
```

For examples, to join the elements of a `string list`:

```
- fun join lnil = "" |
      join (cons(s:string,l:(string list))) = sˆjoin l;
> val join = fn : (string list) -> string

- join (cons("here",cons("we",cons("go",lnil))));
> "herewego" : string
```

Note that existing types cannot be used as base types directly. For example, we might try to define a general number type as:

```
- datatype number = int | real;
> datatype number = int | real
  con int = int : number
  con real = int : number
```

but this defines the new type `number` with base constructors `int` and `real` as if they were simple identifiers for constant values instead of types. A structured constructor must be used to incorporate existing types into a new type, for example:

```
- datatype number = intnumb of int | realnumb of real;
> datatype number = intnumb of int | realnumb of real
  con intnumb = fn : int -> number
  con realnumb = fn : real -> number
```

For example:

```
- intnumb(3);
> intnumb(3) : number

- realnumb(3.3);
>realnumb(3.3) : number
```

Note that structure matching must now be used to extract the values from this structured type:

```
- fun ivalue (intnumb(n:int)) = n;
> val ivalue = fn : number -> int

- fun rvalue (realnumb(r:real)) = r;
> val rvalue = fn : number -> real
```

so, for example:

```
- ivalue (intnumb(3));
> 3 : int

- rvalue (realnumb(3.3));
> 3.3 : real
```

9.24. Trees

We looked at tree construction in chapter 7. We will now see how SML concrete datatypes may be used to construct trees. First of all we will consider binary integer trees.

To recap: a binary integer tree is either empty or it is a node consisting of an integer value, a left sub-tree and a right sub-tree. Thus, we can define a corresponding datatype:

```
- datatype inttree = empty | node of int * inttree * inttree;
> datatype inttree = empty | node of int * inttree * inttree
  con empty = empty : inttree
  con node = fn : (int * inttree * inttree) -> inttree
```

To add an integer to an integer binary tree, if the tree is empty then form a new node with the integer as value and empty left and right sub-trees. Otherwise, if the integer comes before the root node value then add it to the left sub-tree and if it comes afyer the root node value then add it to the right subtree:

```
- fun add (v:int) empty = node(v,empty,empty) |
      add (v:int) (node(nv:int,l:inttree,r:inttree)) =
       if v < nv
       then node(nv,add v l,r)
       else node(nv,l,add v r);
> val add = fn : int -> (inttree -> inttree)
```

For example:

```
    - val root = empty;
    > val root = empty : inttree

    - val root = add 5 root;
    > val root = node(5,empty, empty) : inttree

    - val root = add 3 root;
    > val root = node(5,
                       node(3,empty,empty),
                       empty) : inttree

    - val root = add 7 root;
    > val root = node(5,
                       node(3,empty,empty),
                       node(7,empty,empty)) : inttree

    - val root = add 2 root;
    > val root = node(5,
                       node(3,
                             node(2,empty,empty),
                             empty),
                       node(7,empty,empty)) : inttree

    - val root = add 4 root;
    > val root = node(5,
                       node(3,
                             node(2,empty,empty),
                             node(4,empty,empty)),
                       node(7,empty,empty)) : inttree

    - val root = add 9 root;
    > val root = node(5,
                       node(3,
                             node(2,empty,empty),
                             node(4,empty,empty)),
                       node(7,
                             empty,
                             node(9,empty,empty))) : inttree
```

Given an integer binary tree, to construct an ordered list of node values: if the tree is empty then return the empty list; otherwise, traverse the left sub-tree, pick up the root node value and traverse the right subtree:

```
    - fun traverse empty = [] |
          traverse (node(v:int,l:inttree,r:inttree)) =
           append (traverse l) (v::traverse r);
    > val traverse = fn : inttree -> (int list)
```

For example:

```
    - traverse root;
    > [2,3,4,5,7,9] : int list
```

We can rewrite the above datatype to specify trees of polymorphic type by abstracting with the type variable `’a`:

```
    - datatype ’a tree = empty | node of ’a * (’a tree) * (’a tree);
    > datatype ’a tree = empty | node of ’a * (’a tree) * (’a tree)
    con empty = empty : ('a tree)
    con node = fn : ('a * ('a tree) ('a tree)) -> ('a tree)
```

Similarly, we can define polymorphic versions of `add`:

```
- fun add _ (v:'a) empty = node(v,empty,empty) |
      add (less:'a -> ('a -> bool))
          (v:'a)
          (node(nv:'a,l:'a tree,r:'a tree)) =
       if less v nv
       then node(nv,add less v l,r)
       else node(nv,l,add less v r);
> val add = fn : ('a -> ('a -> bool)) ->
                 ('a -> (('a tree) -> ('a tree)))
```

and `traverse`:

```
- fun traverse empty = [] |
      traverse (node(v:'a,l:'a tree,r:'a tree)) =
       append (traverse l) (v::traverse r);
> val traverse = fn : ('a tree) -> ('a  list)
```

Note the use of the bound variable `less` in `add` to generalise the comparison between the new value and the node value.

9.25. λ calculus in ML

We can use SML to represent directly many pure λ functions. For example, the identity function is:

```
- fn x => x;
> fn : 'a -> 'a
```

Note that this is a polymorphic function from the domain `'a` to the same range `'a`.

Let us apply the identity function to itself:

```
- (fn x => x) (fn x => x);
> fn : 'a -> 'a
```

Alas, SML will not display nameless functions.

For example, the function application function is:

```
- fn f => fn x => (f x);
> fn : ('a -> 'b) -> ('a -> 'b)
```

This is another polymorphic function. Here, `f` is used as a function but its type is not specified so it might be `'a -> 'b` for arbitrary domain `'a` and arbitrary range `'b`. `f` is applied to `x` so `x` must be `'a`. The whole function returns the result of applying `f` to `x` which is of type `'b`.

Let us use this function to apply the identity function to itself:

```
- (fn f => fn x => (f x)) (fn x => x) (fn x => x);
> fn : 'a -> 'a
```

Once again, SML will not display the resulting function. Using global definitions does not help here:

```
- val identity = fn x => x;
> val identity = fn : ’a -> ’a

- identity identity;
> fn : ’a -> ’a

- val apply = fn f => fn x => (f x);
> val apply = fn : (’a -> ’b) -> (’a -> ’b)

- apply identity identity;
> fn : ’a -> ’a
```

as applicative order evaluation replaces name arguments with values.

Some of our λ functions cannot be represented directly in SML as the type system won’t allow self application. For example, in:

```
fn s => (s s)
```

there is a type inconsistency in the function body:

```
(s s)
```

Here, the `s` in the function position is untyped so it might be `’a -> ’b`. Thus, the `s` in the argument position should be `’a` but this clashes with the type for the `s` in the function position!

9.26. Other features

There are many aspects of SML which we cannot cover here. Most important are abstract type construction and modularisation techniques and the use of exceptions to change control flow, in particular to trap errors. SML also provides imperative constructs for assignment, I/O and iteration.

9.27. Summary

In this chapter we have:

- surveyed quickly Standard ML(SML)
- seen how to implement algorithms from preceding chapters in SML
- seen that some pure λ functions cannot be represented in SML

9.28. Exercises

1) Write and test functions to:

i) Find y^3 given integer `y`.

ii) Find `x implies y` from `x implies y == not x or y` given `x` and `y`. The function `implies` should be prefix.

iii) Find the smallest of the integers `a`, `b` and `c`.

iv) Join strings `s1` and `s2` together in descending alphabetic order.

v) Find the shorter of strings `s1` and `s2`.

2) Write and test functions to:

i) Find the sum of the integers between `1` and `n`.

ii) Find the sum of the integers between `m` and `n`.

iii) Repeat a string `s` integer `n` times.

3) Write and test functions to:

i) Count the number of negative integers in a list `l`.

ii) Count how often a given string `s` occurs in a list `l`.

iii) Construct a list of all the integers in a list `l` which are greater than a given value `v`.

iv) Merge two sorted string lists `s1` and `s2`. For example:

```
- smerge ["a","d","e"] ["b","c","f","g"];
> ["a","b","c","d","e","f","g"] : string list
```

v) Use `smerge` from iv) above to construct a single sorted string list from a list of sorted string lists. For example:

```
- slmerge [["a","c","i","j"],["b","d","f"],["e","g","h","k"]];
> ["a","b","c","d","e","f","g","h","i","j","k"] : string list
```

vi) Process a list of stock records represented as tuples of:

```
item name     - string
no. in stock  - integer
reorder level - integer
```

to:

a) construct a list of those stock records with the number in stock less than the reorder level. For example:

```
- check [("RAM",9,10),("ROM",12,10),("PROM",20,21)];
> [("RAM",9,10),("PROM",20,21)] : (string * int * int) list
```

b) update a list of stock records from a list of update records represented as tuples of:

```
item name  - string
update no. - integer
```

by constructing a new stock list of records with the number in stock increased by the update number.

The update records may be in any order. There may not be update records for all stock items. There may be more than one update record for any stock item. For example:

```
- update [("RAM",9,10),("ROM",12,10),("PROM",20,21)]
         [("PROM",15),("RAM",12),("PROM",15)];
> [("RAM",21,10),("ROM",12,10),("PROM",50,21)] : (string * int * int) list
```

4) Write functions to:

i) Extract the leftmost `n` letters from string `s`:

```
- left 4 "goodbye";
> "good" : string
```

ii) Extract the rightmost `n` letters from string `s`:

```
- right 3 "goodbye";
> "bye" : string
```

iii) Extract `n` letters starting with the `l`'th letter from string `s`:

```
- middle 2 5 "goodbye";
> "by" : string
```

iv) Find the position of the first occurence of string `s1` in string `s2`:

```
- find "by" "goodbye";
> 5 : int
```

5) The train travelling east from Glasgow Queen's Street to Edinburgh Waverly passes through Bishopbriggs, Lenzie, Croy, Polmont, Falkirk High, Linlithgow and Edinburgh Haymarket. These stations might be represented by the data type:

```
datatype station = Queens_Street | Bishopbriggs |
                   Lenzie | Croy |
                   Polmont | Falkirk_High |
                   Linlithgow | Haymarket | Waverly;
```

Write functions which return the station to the east or the west of a given station, for example:

```
- east Croy;
> Polmont : station

- west Croy;
> Lenzie : station
```

6) The data type:

```
datatype exp = add of exp * exp |
               diff of exp * exp |
               mult of exp * exp |
               quot of exp * exp |
               numb of int;
```

might be used to represent strictly bracketed integer arithmetic expressions:

```
<expression> ::= (<expression> + <expression>) |
                 (<expression> - <expression>) |
                 (<expression> * <expression>) |
                 (<expression> / <expression>) |
                 <integer>
```

so:

```
(<expression1> + <expression2>)  == add(<expression1>,<expression2>)
(<expression1> - <expression2>)  == diff(<expression1>,<expression2>)
(<expression1> * <expression2>)  == mult(<expression1>,<expression2>)
(<expression1> / <expression2>)  == quot(<expression1>,<expression2>)
<integer> == numb(<integer>)
```

For example:

```
1 == numb(1)
(1 + 2) == add(numb(1),numb(2))
((1 * 2) + 3) == add(mult(numb(1),numb(2)),numb(3))
((1 * 2) + (3 - 4)) == add(mult(numb(1),numb(2)),diff(numb(3),numb(4)))
```

Write a function which evaluates an arithemtic expression in this representation, for example:

```
- eval (numb(1));
> 1 : exp

- eval (add(numb(1),numb(2)));
> 3 : exp

- eval (add(mult(numb(1),numb(2)),numb(3)));
> 5 : exp

- eval (add(mult(numb(1),numb(2)),diff(numb(3),numb(4))));
> 1 : exp
```

10. FUNCTIONAL PROGRAMMING AND LISP

10.1. Introduction

LISP(LISt Processor) is a widely used artificial intelligence language. It is weakly typed with run-time type checking. Functions calls are evaluated in applicative order. LISP lacks structure and pattern matching.

Although LISP is not a pure functional language, it has a number of functional features. Like the λ calculus, LISP has an incredibly simple core syntax. This is based on bracketed symbol sequences which may be interpreted as list data structures or as function calls. The shared representation of data and program reputedly makes LISP particularly appropriate for artificial intelligence applications.

The original LISP programming system was devised by McCarthy in the late 1950's as an aid to the Advice Taker experimental artificial intelligence system. McCarthy's early description of LISP was based on a functional formalism influenced by λ calculus, known as **M-expressions(Meta expressions)**. These were represented in an extremely simple **S-expression'(Symbolic expression)** format for practical programming. Contemporary LISP systems are based solely on the S-expression format although other functional languages are reminiscent of the richer M-expressions. We won't consider M-expressions here.

LISP, like BASIC, is not a unitary language and is available in a number of widely differing dialects. The heart of these differences, as we shall see, lies in the way that name/object associations are treated. Here we will consider COMMON LISP which is a modern standard. We will also look briefly at Scheme, a fully functional LISP. Other LISPs include FRANZ LISP which is widely available on UNIX systems, MACLISP and INTERLISP which COMMON LISP subsumes, and Lispkit Lisp which is another fully functional LISP.

It is important to note that LISP has been refined and developed for many years and so is not a very 'clean' language. LISP systems differ in how some aspects of LISP are implemented. Some aspects of LISP are extremely arcane and subject to much disputation amongst the cognoscenti. Furthermore, while LISP has much in common with functional