プログラミング言語Standard ML入門

— Introduction to Standard ML —

Atsushi Ohori

http://www.pllab.riec.tohoku.ac.jp/~ohori/

Preface

This is a set of slides used in my lecture “Programming Methodology”.
(So exists the next slide.)

本資料は，ML の教科書

> プログラミング言語Standard ML入門，大堀 淳，共立出版，2001.

を使用して行った講義資料です．この教科書とともにお使いください．

Programming Methodology?

The right programming methodology is the Holy Grail of the subject. But does such venerated object exist?

The correct answer : No.

A real answer: to use a better programming language such as ML.

The goal of this course : to master ML programming.

What is ML?

Here are some answers:

- It was a Meta Language for LCF system.
- It is a functional programming language.
- It evolved to be a general purpose programming language.
- It is a statically typed language.
- It is a language with firm theoretical basis.

What are the features of ML?

Some of its notable features are:

- user-defined (algebraic) data types
- powerful pattern matching
- automatic memory management
- first-class functions
- integrated module systems
- polymorphism
- automatic type inference

ML is a cool programming language for cool hackers[1] .

[1] A person with an enthusiasm for programming or using computers as an end in itself.

Historical Notes on ML

- 60's: a meta language for the LCF project.
- 1972: Milner type inference algorithm.
- early 80'S: Cardelli's ML implementation using FAM.
- 80's: a series of proposals for Standard ML in "polymorphism".
- 80's: Standard ML of Edinburgh
- 80's: Caml (so called Caml-heavy later) of INRIA
- late 80's: Caml-light by Leroy.
- early 90's: Standard ML of New Jersey by Appel and MacQueen.
- 90's: Objective Caml by Leroy

⋮

- 2008: SML# at Tohoku University. (This is not a joke.)

Current (and neare futrue) Major Dialects of ML

1. Standard ML
   An "official" one described in the following book.

   R. Milner, M. Tofte, R. Harper, and D. MacQueen
   The Definition of Standard ML (revised), MIT Press, 1997.

   This is the one we learn using Standard ML of New Jersey system.

2. Objective Caml
   A French dialect.

   This is also a name of a system developed at INRIA.

   It deviates from the definition, but it offers high-quality implementation.

⋮

3. SML♯ (to be released in near future)

Textbooks and References

Our lecture will roughly be based on

プログラミング言語Standard ML入門，大堀 淳，共立出版.

Other resources

- L. Paulson, "ML for the Working Programmer, 2nd Edition", Cambridge University Press, 1996.
  (a very good textbook covering many topics)
- R. Harper, "Programming in Standard ML".
  http://www.cs.cmu.edu/People/rwh/introsml/
  (a textbook available on-line)
- M. Tofte, "Four Lectures on Standard ML" (SML'90)
  ftp://ftp.diku.dk/pub/diku/users/tofte/FourLectures/sml/
  (good tutorial, but based on the old version)

Some FAQs or Folklore on ML

- ML is based on theory. So it is difficult to learn, isn't it?
  It should be the opposite!

  Well designed machine with firm foundations is usually easier to use.

  In ML, one only need to understand a small number of principles, e.g.

  - expression and evaluation,
  - functions (recursion and first-class functions),
  - types, and
  - datatypes.

  The structure is much simpler than those of imperative languages.

- But, I heard that functional programming is more difficult.
  No. At least, it is simpler and more natural.
  Compare implementations of the factorial function:

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

```
fun f 0 = 1
  | f n = n * f (n - 1)
```

```
f(int n){
   int r;
   while (n != 0) {
      r = r * n;
      n = n - 1; }
   return(n);}
```

  In general, code in ML is simpler and generally less error prone.

- Programming in ML is inefficient due to lots of compile type errors.

  This makes ML a highly productive language!

  For example, consider the following erroneous code

```
fun f L nil = L
  | f L (h::t) =
        f (h@L) t
```

```
Type Error:
circularity
'Z list -> 'Z list
'Z -> 'Z list
```

```
(defun f (l1 l2)
  (if (null l1) l2
      (cons (cdr l1)
         (f (car l1) l2))))
```

```
append
```

  ML reports all the type errors at the compile type.

  This will make ML programming so efficient.

- Programs written in ML are slower.

- ML is a toy for academics,, and not for real production.

⋮

I would like to challenge these beliefs by developing an industrial strength, next generation ML....

Rest of the Contents of the Course

1. Getting Started (1,2)
2. Value bindings and function definitions (2)
3. Recursive and higher-order functions (3)
4. Static type system (4)
5. ML's built-in data types
6. List (5,6)
7. Recursive datatype definitions (7)
8. Imperative features
9. Module systems
10. Basis libraries
11. Programming project

1. GETTING STARTED

Installing SML of NJ

Go to: http://www.smlnj.org/software.html and download the software.

- Windows systems: smlnj.exe (a self-extracting archive.)
- Linux RPM: smlnj-110.0.7-4.i386.rpm
- Other Unix systems:
  Follow the instructions given in the SML NJ page.

It is already installed in JAIST system.

But if you have a PC, do Install the system on it.

How to use ML

To start interactive session:

`% sml`	(invoking SML/NJ compiler)
Standard ML	(opening message)
`-`	(input prompt)

After this, the user and the system iterate the following interaction:

1. the user inputs a program,
2. the system compiles and executes the program, and
3. the system prints the result.

Terminating the session:

- To terminate, type `^D` at the top-level.
- To cancel evaluation, type `^C`.

Expression and Evaluations

The fundamental principle of functional programming:

a program is an expression that evaluates to a value.

So in your interactive session, you do:

- $expr$; (an expression)
$val\ it = value : type$ (the result)

where

- ";" indicates the end of a compilation unit,
- $value$ is a representation of the computed result,
- $type$ is the type ML compiler inferred for your program, and
- `val it` means the system remember the result as "`it`".

Atomic Expressions

In ML, the "hello world" program is simply:

```
% sml
- "Hi, there!";
```

val it = "Hi, there!" : string

In ML, a simple program is indeed very simple!
Compare the above with that of C.

Note:

- Here "Hi, there!" is an expression and therefore a complete program!
- This is an example of atomic expressions or value literals.
- The result of an atomic expression is itself.
- ML always infers the type of a program. In this case, it is string.

Some atomic expressions:

```
- 21;
```
$val\ it = 21 : int$
```
- 3.14;
```
$val\ it = 3.14 : real$
```
- 1E2;
```
$val\ it = 100.0 : real$
```
- true;
```

```
- true;
```
$val\ it = true : bool$
```
- false;
```
$val\ it = false : bool$
```
- #"A";
```
$val\ it = \#"A" : char$

where

- int : integers
- $real$: floating point numbers
- $bool$: boolean values
- $char$: characters

Expressions with Primitive Operations

Simple arithmetic expressions

```
- ~2;
```
val it = ~2 : int
```
- 3 + ~2;
```
val it = 1 : int
```
- 22 - (23 mod 3);
```
val it = 20 : int
```
- 3.14 * 6.0 * 6.0 * (60.0/360.0);
```
val it = 18.84 : real

- `~`n is a negative number, and `~` is the operator to multiply `~1`.
- `mod` is the remainder operator.

Expression can be of any length.

```
- 2 * 4 div (5 - 3)
= * 3 + (10 - 7) + 6;
```

val it = 21 : int

The first $=$ above is the continue-line prompt.

Conditional and Boolean expressions

In ML, a program is an expression, so

a conditional program is also an expression.

```
- if true then 1 else 2;
```
val it = 1 : int
```
- if false then 1 else 2;
```
val it = 2 : int

where `true` and `false` are boolean literals.

Note:

- `if` E_1 `then` E_2 `else` E_3 is an expression evaluates to a value.
- E_1 can be any boolean expressions.

A fundamental principle of a typed language:

expressions can be freely combined as far as they are type correct.

So you can do:

```
- (7 mod 2) = 2;
```
val it = false : bool
```
- (7 mod 2) = (if false then 1 else 2);
```
val it = false : bool
```
- if (7 mod 2) = 0 then 7 else 7 - 1;
```
val it = 60 : int
```
- 6 * 10;
```
val it = 60 : int
```
- (if (7 mod 2) = 0 then 7 else 7 - 1) * 10;
```
val it = 60 : int

Expression "`it`"

The system maintain a special expression `it` which always evaluates to the last successful result.

```
- 31;
```
val it = 31 : int
```
- it;
```
val it = 31 : int
```
- it + 1;
```
val it = 32 : int
```
- it;
```
val it = 32 : int

Characters and Strings

```
- if #"A" > #"a" then ord #"A" else ord #"a";
```

val it = 97 : int

```
- chr 97;
```

val it = #"a" : char

```
- str it;
```

val it = "a" : string

```
- "SML" > "Lisp" ;
```

val it = true : bool

```
- "Standard " ^ "ML";
```

val it = "Standard ML" : string

- $e_1 \;\; e_2$ is function application.
- `ord` e returns the ASCII code of character e.
- `chr` e returns the character of the ASCII code e.

Reading programs from a file

To execute programs in a file, do the following:

`use "`*file*`" ;`

The system then perform the following:

1. open the file
2. compile, evaluate, and print the result for each compilation unit separated by ";"
3. close the file
4. return to the top-level.

Syntax Error and Type Error

As in other language, ML compiler reports syntax error:

```
- (2 +2] + 4);
```

stdIn:1.7 Error: syntax error found at RBRACKET

It also report the type errors as follows:

```
- 33 + "cat";
```

stdIn:21.1-21.11 Error: operator and operand don't agree [literal]
    *operator domain: int * int*
    *operand: int * string*
    in expression:
        33 + "cat"

No type cast or overloading

To achieve complete static typechecking, there is no type cast nor overloading.

```
- 10 * 3.14;
```

stdIn:2.1-2.10 Error: operator and operand don't agree [literal]
*operator domain: int * int*
*operand: int * real*
in expression:
*10 * 3.14*

You need explicit type conversion:

```
- real 10;
```

val it = 10.0 : real

```
- it * 3.14;
```

val it = 31.4 : real

Exercise

1. Install the SML/NJ system and perform the interactive session shown in this note.
2. Predict the result of each of the following expression.

```
44;
it mod 3;
44 - it;
(it mod 3) = 0;
if it then "Boring" else "Strange";
```

   Check your prediction by evaluating them in SML/NJ.
3. In the ASCII code, uppercase letters A trough Z are adjacent and in this order. The same is true for the set of lowercase letters. Assuming only this fact, write an expression that evaluates to an upper case

character `X` to its lower case where `X` is the name bound to some upper case character.

4. In the ASCII code system, the set of uppercase letters and that of lowercase letters are not adjacent. Write an expression that evaluates to the number of letters between these sets. Use this result and write an expression to return a string of the characters between these two sets.

2. VALUE BINDING AND FUNCTION DEFINITIONS

Variable binding

The first step of a large program development is to name an expression and use it in the subsequent context using the syntax:

`val` *name* `=` *exp* `;`

which "binds" *name* to *expr* and store the binding in the top-level environment.

```
- val OneMile = 1.6093;
```
val OneMile = 1.6093 : real
```
- OneMile;
```
val it = 1.6093 : real
```
- 100.0 / OneMile;
```
val it = 62.1388181197 : real

There is no need to declare variables and their types.

```
- val OneMile = 1.609;
```
val OneMile = 1.609 : real
```
- val OneMile = 1609;
```
val OneMile = 1609 : int
```
- OneMile * 55;
```
val it = 88495

Reference to an undeclared variable results in type error.

```
- onemile * 55;
```
stdIn:22.1-22.8 Error: unbound variable or constructor: onemile

Identifiers

A name can be one of the following identifiers:

1. alphabetical
   Starting with uppercase letter, lowercase letter, or #"'", and
   containing only of letters, numerals, #"'", and #"_".
   Identifier starting with #"'" are for names of type variables.

2. symbolic
   a string consisting of the following characters;

```
! % & $ # + - / : < = > ? @
\ ~ ` ^ | *
```

Keywords

```
abstype and andalso as case datatype do else end
eqtype exception fn fun functor handle if in
include infix infixr let local nonfix of op open
orelse raise rec sharing sig signature struct
structure then type val where while with withtype
(  )  [  [  {  }  ,  :  :>  ;  ...  _  |  =  =>  ->
#
```

Function Definitions

A function is defined by fun construct:

`fun` f p `=` $body$ `;`

where

- f is the name of the function,
- p is a formal parameter, and
- $body$ is the function body.

This declaration define a function that takes a parameter named p and return the value $body$ computes.

Simple example:

```
- fun double x = x * 2;
```
val double = fn : int -> int

where

- `val double = fn` indicates that `double` is bound to a function value.
- `int -> int` is a type of functions that takes an integer and returns an integer.

  This function can be used as:

  ```
  - double 1;
  ```
  val it = 2 : int

Applying a Function

Typing principle:

f of type τ_1 `->` τ_2 can be applied to an expression E of type τ_1, yielding a value of type τ_2.

This can be written concisely as:

$$\frac{f : \tau_1 \texttt{ -> } \tau_2 \qquad E : \tau_1}{f\ E : \tau_2}$$

Remember our fundamental principle on typed language:

expressions can be freely combined as far as they are type correct.

So E can be any expression and $f\ E$ can occurs whenever type τ_1 is allowed.

```
- double (if true then 2 else 3);
```

val it = 4 : int

```
- double 3 + 1;
```

val it = 7 : int

```
- double (double 2) + 3;
```

val it = 11 : int

Important note:

Function application $E_1\ E_2$ associates tightest and from the left.

So, `double 3 + 1` is interpreted as `(double 3) + 1`.

Function Definitions with Simple Patterns

```
- fun f (x,y) = x * 2 + y;
```
*val f = fn : int * int -> int*
```
- f (2,3);
```
val it = 7 : int

where

- $(E_1, \ldots, E_n)$ is a tuple,
- τ_1 `*` $\cdots$ `*` τ_n is a tuple type,
- Function type constructor `->` associate weakly than other type constructor, so `int * int -> int` is `(int * int) -> int`.

Typing rule for tuples is:

$$\frac{E_1 : \tau_1 \qquad \cdots \qquad E_n : \tau_n}{(E_1, \ldots, E_n) \;:\; \tau_1 \;\texttt{*}\; \cdots \;\texttt{*}\; \tau_n}$$

So you can form any form of tuples:

```
- ("Oleo",("Kenny","Drew"),1975);
```

$val\ it =$ `("Oleo",("Kenny","Drew"),1975)`

$: string * (string * string) * int$

Evaluation Process of Function Application

The system evaluate $E_1\ E_2$ in the following steps:

1. Evaluate E_1 to obtain a function definition `fn x =>` E_0
2. Evaluate the argument E_2 to a value v_0.
3. Extend the environment in which the function is defined with the binding $x \mapsto v_0$,
4. evaluate the function body E_0 under the extended environment and obtain the result value v.
5. v is the result of $E_1\ E_2$.

Evaluation process:

```
Eval({}, double (double 2) + 3)
    | Eval({}, double (double 2))
    |     | Eval({}, (double 2))
    |     |    | Eval({}, 2) = 2
    |     |    | Eval({x ↦ 2}, x * 2)
    |     |    |    | Eval({x ↦ 2}, x) = 2
    |     |    |    | Eval({x ↦ 2}, 2) = 2
    |     |    |    | Eval({x ↦ 2}, 2 * 2) = 4
    |     |    |    = 4
    |     |    = 4
    |     | Eval({x ↦ 4}, x * 2)
    |     |    | Eval({x ↦ 4}, x) = 4
    |     |    | Eval({x ↦ 4}, 2) = 2
    |     |    | Eval({x ↦ 4}, 4 * 2) = 8
    |     |    = 8
    |     = 8
    | Eval({}, 3) = 3
    | Eval({}, 8 + 3) = 11
    = 11
```

3. RECURSIVE AND HIGHER-ORDER FUNCTIONS

Recursive Functions

A function definition

`fun` f p `=` $body$

is recursive if $body$ contains f, the function being defined by this definition.

A simple example: the factorial function

$$0! = 1$$
$$n! = n \times (n-1)!$$

```
fun f n =
    if n = 0 then 1
    else n * f (n - 1)
```

Many complex problems can naturally be solved recursively.

How to design a recursive function:

$$0! = 1$$
$$n! = n \times (n-1)!$$

1. Write the trivial case.
   `if n = 0 then 1`
2. Decompose a complex case into smaller problems, and solve the smaller problems by using the function you are defining.
   `f (n - 1)`
3. Compose the results to obtain the solution.
   `n * f (n - 1)`

How to analyze a recursive function definition:

```
fun f n =
    if n = 0 then 1
    else n * f (n - 1)
```

1. Assume that f computes the desired function.
   `f n` is $n!$
2. Show that $body$ is correct under the assumption above.
   `f 0` is 0!.
   By the assumption, `f (n - 1)` is $(n-1)!$.
   So `f n` is $n!$. So the body of the above definition is correct.
3. If the above step succeed then f indeed computes the desired function.

Fibonacci sequence is defined as follows:

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1} \qquad (n \geq 2) \end{aligned}$$

The following function computers this sequence.

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

Tail Recursive Functions

Consider the definition again:

```
fun f n =
    if n = 0 then 1
    else n * f (n - 1)
```

This function uses $O(n)$ stack space. However, we can write a C code that only using a few variables.

Is ML inefficient?

The answer is the obvious one:

Efficient ML programs are efficient, and inefficient ML programs are inefficient.

A recursive function definition `fun` f p `=` $body$ is tail recursive if all the calls of f in $body$ are in tail calls positions in $body$.

[ ] is a tail call position in T:

$$
\begin{aligned}
T \;:=\; [\,] \mid \texttt{if}\ E\ \texttt{then}\ T\ \texttt{else}\ T \mid (E\texttt{;}\ \cdots\texttt{;}\ T) \\
\mid \texttt{let}\ decs\ \texttt{in}\ T\ \texttt{end} \\
\mid \texttt{case}\ e\ \texttt{of}\ p_1\ \texttt{=>}\ T\ \texttt{|}\ \cdots\ \texttt{|}\ p_n\ \texttt{=>}\ T
\end{aligned}
$$

Examples of tail calls:

- `f` e
- `if` e_1 `then` `f` e_2 `else` `f` e_3
- `let` $decls$ `in` `f` e `end`

Tail recursive functions do not consume stack space and therefore more efficient.

Tail recursive version of `fact`:

```
fun fact (n, a) = if n = 0 then a
                  else fact (n - 1, n * a);
```
*val fact = fn : int * int -> int*

```
fun factorial n = fact (n,1);
```
val factorial = fn : int -> int

fact is an auxiliary function for factorial.

Let Expression

The following let expression introduces local definitions

```
let
    sequence of val or fun definitions
in
    exp
end
```

Simple example:

```
let
    val x = 1
in
    x + 3
end;
```

val it = 4 : int

A function is usually defined using this construct as:

```
- fun factorial n =
      let
         fun fact n a = if n = 0 then a
                        else fact (n - 1) (n * a)
      in
         fact n 1
      end;
```

val factorial = fn : int -> int

Notes on `let` *decl* `in` *exp* `end`:

- This is an expression.
- The type and the result of this expression is those of *exp*.

```
- let
     val pi = 3.141592
     fun f r = 2.0 * pi * r
  in
     f 10.0
  end * 10.0;
```

val it = 628.3184 : real

Local Definitions

The following local special form also introduce local definitions

```
local
    declList1
in
    declList2
end
```

- Only the definitions in $declList2$ are valid after this declaration.
- $declList1$ is used in $declList2$.

```
- local
     fun fact n a = if n = 0 then a
                    else fact (n - 1) (n*a)
  in
     fun factorial n = fact n 1
  end;
```

val factorial = fn : int -> int

```
- fact;
```

stdIn:10.1-10.4 Error: unbound variable or constructor fact

Mutually Recursive Functions

Functions/problems are often mutually recursive.

Consider that you deposit some money x under the condition that each year's interest rate is determined by some function F from the previous year's balance.

Let

- A_x^n be the amount after n year deposit
- I_x^n the $n'th$ year's interest rate.

They satisfy the following equations:

$$I_x^n = F(A_x^{n-1}) \qquad (n \geq 1)$$
$$A_x^n = \begin{cases} x & (n = 0) \\ A_x^{n-1} \times (1.0 + I_x^n) & (n \geq 1) \end{cases}$$

which can be directly programed as follows:

```
- fun I(x,n) = F(A(x,n - 1))
  and A(x,n) = if n = 0 then x
               else A(x,n-1)*(1.0+I(x,n));
val I = fn : real * int -> real
val A = fn : real * int -> real
```

Recursion and Efficiency

Straightforward implementation of recursive functions results in inefficient programs due to repeated calls on same arguments.

Consider again:

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

In computing fib n,

- fib n is called once; fib (n - 1)) is called once
- fib (n - 2)) is called twice; fib (n - 3)) is called 3 times
- fib (n - 4)) is called 5 times; fib (n - 5)) is called 8 times

⋮

Question: How may times fib 0 is called?

To avoid repeated computation, we consider a function G_k

$$G_k(F_{n-1}, F_n) = F_{n+k}$$

which takes two consecutive Fibonacci numbers (F_{n-1}, F_n) and returns the Fibonacci number F_{n+k}.

G satisfies the following recursive equations:

$$\begin{aligned} G_0(a, b) &= a \\ G_1(a, b) &= b \\ G_{k+1}(a, b) &= G_k(b, a + b) \end{aligned}$$

```
fun fastFib n =
    let
       fun G(n,a,b) =
           if n = 0 then a
           else if n = 1 then b
           else G(n-1,b,a+b)
    in
      G(n,1,1)
    end
```

Only one call for `G(n,a,b)` for each n, so this function computes F_n in $O(n)$ time.

The same is true for `A` and `I` in

```
fun I(x,n) = F(A(x,n - 1))
and A(x,n) = if n = 0 then x
                  else A(x,n-1)*(1.0+I(x,n));
```

In computing A(x,n),

- A(x,n) is called once,
- A(x,n - 1)) is called twice
- A(x,n - 2)) is called 4 times

  ⋮

Repeated computation is avoided by computing A_x^n and I_x^n using a function G defined as

$$G_k(A^n, I^n) = (A^{n+k}, I^{n+k})$$

which satisfies:

$$\begin{aligned} G_0(a, i) &= (a, i) \\ G_{k+1}(a, i) &= G_k(a \times (1 + I), F(a)) \end{aligned}$$

So we can define:

```
local
     fun G(n,a,i) = if n = 0 then (a,i)
                    else G(n-1,a * (1.0 + i)),F a)
in
     fun A(n,x) = #1 (G(n,x F(x)))
     fun I(n,x) = #2 (G(n,x F(x)))
end
```

Higher-Order Functions

(1) Functions can return a function

Consider a function taking 2 arguments:

```
- fun Power(m,n) = if m = 0 then 1
                   else n * Power(m - 1,n);
```
*val Power = fn : int * int -> int*

```
- Power(3,2);
```
val it = 8 : int

But in ML, we can also define:

```
- fun power m n = if m = 0 then 1
                  else n * power (m - 1) n;
```
val power = fn : int -> int -> int

power is a function that takes one argument m and returns a function that takes an argument n and returns m^n.

So we can write:

```
- val cube = power 3;
```

val cube = fn : int -> int

```
- cube 2;
```

val it = 8 : int

Note:

- function application associates to the left so power (m - 1) n is (power (m - 1)) n.
- -> associates to the right so int -> int -> int is int -> (int -> int).

(2) Functions can take functions as its arguments

To understand such a function, let us first consider the following function:

```
- fun sumOfCube n = if n = 1 then cube 1
                    else cube n + sumOfCube (n - 1);
```

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

```
- sumOfCube 3;
```

val it = 36 : int

We note that `sumOfCube` is a special case of the computation:

$$\sum_{k=1}^{n} f(k) = f(1) + f(2) + \cdots + f(n)$$

In ML, we can directly define such computation as a program.

```
- fun summation f n = if n = 1 then f 1
                      else f n + summation f (n - 1);
```

val summation = fn : (int -> int) -> int -> int

Remember:

- (int -> int) -> int -> int is (int -> int) -> (int -> int).
- summation is a function that
  - takes a value of type int -> int, and
  - return a value of type int -> int.

So summation is a function that takes a function as its argument and returns a function.

Remember our principle on typing:

expressions can be freely combined as far as they are type correct.

We can use summation as far as the usage is type consistent.

```
- val newSumOfCube = summation cube;
```

val newSumOfCube = fn : int -> int

```
- newSumOfCube 3;
```

val it = 36 : int

We can also write:

```
- val sumOfSquare = summation (power 2);
```

val sumOfSquare = fn : int -> int

```
- sumOfSquare 3;
```

val it = 14

```
- summation (power 4) 3;
```

val it = 98 : int

Function Expressions

In ML, a function is a value, and therefore representable by expressions.

A function expression of the form

$$\texttt{fn}\ p\ \texttt{=>}\ body$$

denotes the function that takes p and returns the value denoted by $body$.

```
- fn x => x + 1;
val it = fn : int -> int
- (fn x => x + 1) 3 * 10;
val it = 40 : int
```

Using this function expression, we can write a program to generate a function.

To see its usefulness, consider:

```
fun f n m = (fib n) mod m = 0;
```

which computes fib n for each different m. So

```
val g = f 35;
(g 1,g 2,g 3,g 4,g 5);
```

compute fib 35 5 times.
This is clearly redundant, and should be avoided.

We can factor out the computation fib 35 as

```
fun f n = let val a = (fib n)
          in fn m => a mod m = 0
          end
```

This performs the following computation:

1. receives n
2. computes fib n and binds m to the result,
3. return a function `fn m => a mod m = 0`.

So after this,

```
val g = f 35;
(g 1,g 2,g 3,g 4,g 5);
```

will not compute `fib 35` again.

Static Scoping

General principle: defining a name hide the previous definition.

A simple example:

```
- let val x = 3 in (fn x => x + 1) x end;
```
val it = 4 : int

Names are defined in

- `val` definition
- `fun` definition
- `fn` x `=>` e expression

`val` declaration

$\texttt{val}\ x_1\ \texttt{=}\ exp_1$
$\texttt{and}\ x_2\ \texttt{=}\ exp_2$
$\vdots$
$\texttt{and}\ x_n\ \texttt{=}\ exp_n\ \texttt{;}$

The scope of $x_1, \ldots, x_n$ is the expressions that follow this definition.

```
val x = 1
val y = x + 1
val x = y + 1
and y = x + 1
```

will binds `x` to 3 and `y` to 2.

Function declarations

$$
\begin{array}{l}
\texttt{fun}\ \ f_1\ \ p^1_1\ \ \cdots\ \ p^1_{k_1}\ \ \texttt{=}\ \ exp_1 \\
\texttt{and}\ \ f_2\ \ p^2_1\ \ \cdots\ \ p^2_{k_2}\ \ \texttt{=}\ \ exp_2 \\
\qquad \vdots \\
\texttt{and}\ \ f_n\ \ p^n_1\ \ \cdots\ \ p^n_{k_n}\ \ \texttt{=}\ \ exp_n\ \ \texttt{;}
\end{array}
$$

The scope of $f_1, \ldots, f_n$ are

- $exp_1, \ldots, exp_n$, and
- the expressions that follow this definition.

Note:

In static scoping, new bindings only hid previous binding and do not change them.

```
- val x = 10;
```
val x = 10 : int
```
- val y = x * 2;
```
val y = 20 : int
```
- val x = 20;
```
val x = 20 : int
```
- y;
```
val it = 20 : int

This is also true for function definitions.

```
- val x = 10;
val x = 10 : int
- val y = 20;
val y = 20 : int
- fun f x = x + y;
val f = fn : int -> int
- f 3;
val it = 23 : int
- val y = 99;
val y = 99 : int
- f 3;
val it = 23 : int
```

Binary Operators

In ML, operators are functions and functions only take one arguments.

Operator expressions of the form

e_1 `op` e_2

is a *syntactic sugar* for

$op(e_1, e_2)$

The following declaration defines operator syntax:

- `infix` n $id_1 \cdots id_n$: left associative operator of strength n.
- `infixr` n $id_1 \cdots id_n$: right associative operator of strength n.

The system pre-defined the following

```
infix 7 * /
infix 6 + -
```

You can also define your one operators as:

```
- infix 8 Power;
```
infix 8 Power
```
- 2 Power 3 + 10;
```
val it = 19 : int

`op` *id* temporarily invalidate operator declarations:

```
- Power;
```
stdIn:4.1 Error: nonfix identifier required
```
- op Power;
```
*val it = fn : int * int* `->` *int*
```
- op Power (2,3);
```
val it = 9 : int

Exercise Set (2)

1. For each of the following, write down a recursive equation, and a recursive program corresponding to the equation.

   (1) $S_n = 1 + 2 + \cdots + n$

   (2) $S_n = 1 + (1+2) + (1+2+3) + \cdots + (1+2+\cdots+n)$

2. Write a tail recursive definition for each of the above.

3. Let us represent a 2×2 matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ as (a, b, c, d). Write a function `matrixPower(n,A)` that takes any matrix A and an integer n, and returns A^n.

4. From the definition of Fibonacci numbers we have:

$$\begin{aligned} F_n &= F_n \\ F_{n+1} &= F_{n-1} + F_n \end{aligned}$$

Let

$$G_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$$

Then we have:

$$G_n = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} G_{n-1}$$

and therefore

$$G_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdots \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}}_{n \text{ times}} \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Thus if $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ then we have:

$$G_n = A^n \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Using the above facts, define a function to compute F_n using `matrixPower`

5. Define a tail recursive version of `summation`.
6. For each of the following, define a function using `summation`.

   (1) $f(x) = 1 + 2 + \cdots + n$
   (2) $f(n) = 1 + 2^2 + 3^2 + \cdots + n^2$
   (3) $f(n) = 1 + (1 + 2) + (1 + 2 + 3) + \cdots + (1 + 2 + \cdots + n)$
7. Define

```
summation' : (int -> real) -> int -> real
```

   that computes $\Sigma_{k=1}^{n} f(k)$ for a given function of type `int -> real`.
8. Let $f(x)$ be a function on real numbers.

$$\int_a^b f(x)dx$$

can be approximated by

$$\sum_{k=1}^{n}\left(f\left(a+\frac{k(b-a)}{n}\right)\times\frac{b-a}{n}\right)$$

for some large n.

Define a function `integral` that takes f, n, a and b, and compute the above value.

You will need a function `real : int -> real` that converts a value of type `int` to the corresponding value of `real`.

9. `summation` can be regarded as a special case of the following more general computation scheme:

$$\Lambda_{k=1}^{n}(h, f, z) = h(f(n), \cdots, h(f(1), z)\cdots)$$

For example, $\Sigma_{k=1}^{n} f(k)$ can be defined as $\Lambda_{k=1}^{n}(+, f, 0)$.

Write a higher-order function

```
accumulate h z f n
```

that compute $\Lambda_{k=1}^{n}(h, z, f)$.

10. Define `summation` using `accumulate`.

11. Define each of the following function using `accumulate`.

    (1) $f_1(n) = 1 + 2 + \cdots + n$

    (2) $f_2(n) = 1 \times 2 \times \cdots \times n$

    (3) $f_3(n, x) = 1 \times x^1 + 2 \times x^2 + \cdots + n \times x^n$

12. We can regard a function f of type `int -> bool` as a set of elements for which f returns true. For example,

```
fn x => x = 1 orelse x = 2 orelse x = 3
```

can be regarded as a representation of the set $\{1, 2, 3\}$. In this, exp_1 `orelse` exp_2 is logical or of exp_1 and exp_2. An expression to denote the logical and of exp_1 and exp_2 is exp_1 `andalso` exp_2.

Based on the above observation, write the following functions for manipulating sets:

(1) `emptyset` to denote the emptyset.

(2) `singleton` that returns the singleton set $\{n\}$ of a given n.

(3) `insert` to insert an element n to a set S.

(4) `member` to test whether an element n is in a set S.

(5) set theoretic functions: `union`, `intersection`, `difference`.

4. TYPE SYSTEM OF ML

Type Inference and Type Checking

The two most notable features of ML type system:

1. It automatically infers a type for any expression.
2. It supports polymorphism.

Automatic Type Inference

As we have seen, ML programs do not require type declarations.

```
fun products f n = if n = 0 then 1
                   else f n * products f (n - 1)
```

which is equivalent to

```
(defun products (f n)
   (if (<= n 0) 1
       (* (funcall f n) (products f (- n 1)))))
```

But ML infers its type!

val products = fn : (int -> int) -> int -> int

This achieves complete static type checking without type annotation:

```
fun factorial n = products n (fn x => x);
```

stdIn:1.19-1.40 Error: operator and operand don't agree

operator domain: int

operand: 'Z -> 'Z

in expression: products n ((fn x => x))

This is in contrast to untyped language like LISP:

```
- (defun factorial (n)
         (products n #'(lambda (x) x)))
```

factorial

⋮

```
(··· (factorial 4) ···)
```

Worng type argument: integer-or-marker-p (lambda (x) x)

Examples of type errors:

- `3 * 3.14`
- `fun f1 x = (x 1,x true)`
- `fun f2 x y = if true then x y else y x`
- `fun f3 x = x x`

Polymorphism

A program has many different types.

```
fun id x = x
```

`id` has infinitely many types such as

```
int -> int
```

or

```
string -> string.
```

or any other types of the form τ`->` τ.

ML infers a type that represent all the possible types of a given program.

val id = fn : 'a `->` *'a*

where `'a` is a *type variable* representing arbitrary types.

Type variables are *instantiated* when the program is used.

```
- id 21;
```
val it = 21 : int
```
- id "Standard ML";
```
val it = "Standard ML" : string
```
- id products;
```
val it = fn : (int -> int) -> int -> int
```
- fn x => id id x
```
val it = fn : 'a -> 'a

More examples:

```
- fn x => id id x;
```
val it = fn : 'a -> 'a
```
- fun twice f x = f (f x);
```
val twice = fn : ('a -> 'a) -> 'a -> 'a

```
- twice cube 2;
val it = 512 : int
- twice (fn x => x ^ x) "ML";
val it = "MLMLMLML" : string
- fn x => twice twice x;
val it = fn : ('a -> 'a) -> 'a -> 'a
- it (fn x => x + 1) 1;
val it = 5 : int
```

ML infers a polymorphic type by computing a most general solution of type constraints.
Let us trace the process for `twice` ML proceeds roughly as follows.

1. Assign types to each sub-expressions

$$\texttt{fun}\ \underline{\texttt{twice}}_{\tau_1}\ \underline{\texttt{f}}_{\tau_2}\ \underline{\texttt{x}}_{\tau_3} = \underline{\texttt{f}\ \ \underline{\texttt{(f x)}}_{\tau_4}}_{\tau_5};$$

2. Since `twice` is a function that takes `f` and `x`, we must have:

$$\tau_1 = \tau_2\texttt{->}\tau_3\texttt{->}\tau_5$$

3. For $\underline{\texttt{(f x)}}_{\tau_4}$ to be type correct, we must have;

   (1) `f`'s type τ_2 must be a function type of the form α `->` β

   (2) α must be equal to `x`'s type τ_3,

   (3) β must be equal to the result type τ_4 of `(f x)`.

   So we have the following equation:

$$\tau_2 = \tau_3\ \texttt{->}\ \tau_4$$

4. Similarly, for $\underline{\texttt{f}\ \underline{\texttt{(f x)}}_{\tau_4}}_{\tau_5}$ to be type correct, we must have the equation:

$$\tau_2 = \tau_4 \texttt{->} \tau_5$$

5. The desired type is a most general solution of the following type equations:

$$\begin{aligned}\tau_1 &= \tau_2 \texttt{->} \tau_3 \texttt{->} \tau_5 \\ \tau_2 &= \tau_3 \texttt{->} \tau_4 \\ \tau_2 &= \tau_4 \texttt{->} \tau_5\end{aligned}$$

   One such solution is

$$\tau_1 = (\tau_3 \texttt{->} \tau_3) \texttt{->} \tau_3 \texttt{->} \tau_3$$

Explicit Type Declaration

You can specify types as in:

```
- fun intTwice f (x:int) = f (f x);
```

val intTwice = fn : (int -> int) -> int -> int

or

```
fun intTwice (f:int -> int) x = f (f x)
fun intTwice f x = f (f x) : int
fun intTwice f x : int = f (f x)
```

Type specification is weaker than function application, so the last example means

`fun ((intTwice f) x):int = f (f x)`.

Type specification can be polymorphic

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

However, you cannot specify a more general type than the actual type.

Overloaded Identifiers

Some system defined identifiers have multiple definitions, and disambiguated by the context.

```
- 1 < 2;
val it = true : bool
- "dog" < "cat";
val it = false : bool
- fun comp x y = x < y;
val comp = fn : int -> int -> bool
```

If there is no context to disambiguate, then the default is chosen.

Some of overloaded identifiers:

id	type scheme	possible t	default
`div`	t `*` t `->` t	`word`,`int`	`int`
`mod`	t `*` t `->` t	`word`,`int`	`int`
`+,-`	t `*` t `->` t	`word`,`real`,`int`	`int`
`<,>,<=,>=`	t `*` t `->` `bool`	`word`,`real`,`int`,`string`,`char`	`int`

Value Polymorphism

Currently, polymorphism is restricted to values:

$$V ::= c \mid x \mid (V, \ldots, V) \mid \texttt{fn}\ p\ \texttt{=>}\ e$$

So

```
(fn x => x, fn y => y);
```

*it = poly-record : ('a -> 'a) * ('b -> 'b)*

is OK, but

```
- fun f n x = (n,x);
```

*val f = fn : 'a -> 'b -> 'a * 'b*

```
- f 1;
```

stdIn:21.1-21.4 Warning: type vars not generalized because of value restriction are instantiated to dummy types (X1,X2,...)
*val it = fn : ?.X1 -> int * ?.X1*

Note on "Value Polymorphism"

This is a "theoretical compromise", and is not particularly nice.

There are several alternative solutions. One is to use

rank 1 polymorphism

where one can do the following

```
- fun f n x = (n,x);
val f = fn : ['a.'a -> ['b.'b ->'a * 'b]]
- f 1;
val it = fn : ['a.'a -> int * 'a]
```

We will come back to this point later.

Equality Types

Equality function "=" tests whether two expressions have the same value or not. However,

we cannot compute equality of functions or reals.

So "=" is restricted to those types on which equality is computable.

```
- op =;
```

*val it = fn : ''a * ''a ->bool*

where ''a is a equality type variable ranging over equality types

$$\delta ::= int \mid bool \mid char \mid string \mid \delta * \cdots * \delta \mid ref\,\tau$$

ref τ is a reference type of τ we shall learn later.

Exercise Set (3)

1. For each of the following expressions, if it is type correct then infer its most general type, and if it does not have any type then explain briefly the cause of the type error.

   (1) `fun S x y z = (x z) (y z)`

   (2) `fun K x y = x`

   (3) `fun A x y z = z y x`

   (4) `fun B f g = f g g`

   (5) `fun C x = x C`

   (6) `fun D p a b = if p a then (b,a) else (a,b)`

2. What are the type of the following expressions?

```
f x = f x;
f 1;
```

3. Explain the type and the behavior of the following function.

```
local
      fun K x y = x
in
      fun f x = K x (fn y => x (x 1))
end
```

4. As you can see above, it is possible to constrain the type of expression without using explicit type declaration. Let τ be a type not containing any type variable, and suppose E is an expression of type τ.

   (1) Let exp be an expression whose type is more general than that of E. Give an expression whose behavior is the same as that of exp but have the type τ.

   (2) Define a function that behaves as the identity function and its most general type is τ `->` τ.

5. Functions can represent any data structures, including integers. Let us do some arithmetic using functions. Consider the `twice` again:

```
val twice = fn f => fn x => f (f x)
```

This represents the notion of "doing something *two* times".

To verify this, we can see by applying it as

```
- twice (fn x => "*" ^ x) "";
val it = "**" : string
- twice (fn x => "<" ^ x ^ ">") "";
val it = "<<>>" : string
```

and of course

```
- twice (fn x => x + 1) 0;
val it = 2 : int
```

So we can consider twice as (a representation of) the number 2!

So we can define numbers as

```
val one = fn f => fn x => f x
val two = fn f => fn x => f (f x)
val three= fn f => fn x => f (f (f x))
val four= fn f => fn x => f (f (f (f x)))
   ⋮
```

and a utility function

```
fun show n = n (fn x => x + 1) 0
```

Let us call these functional representation *numerals*.

In this exercise, we define various operations on natural numbers based on this idea.

(1) Define a numeral for the number zero.

(2) Define a function `succ` that represents "add one" function. For example, it should behave as:

```
- show(succ one);
```
val it = 2 : int

(3) Define functions `add`, `mul`, and `exp` to compute addition, multiplication and exponentiation. For example, we should have:

```
- show(add one two);
```
val it = 3 : int
```
- show(mul (add one two) three);
```
val it = 9 : int
```
- show(exp two three);
```
val it = 8 : int

PREDEFINED DATA TYPES

Unit Type

`unit` type contain `()` as its only value. It is used for functions whose return values are not important:

```
val use :  string -> unit
val print :  string -> unit
```

For example:

```
- use "emptyFile";
```
val it = () : unit
```
- print "Hi!";
```
Hi!val it = () : unit

Other functions related to unit are:

```
infix 0 before
val before : 'a * unit -> 'a
val ignore : 'a -> unit
```

For example:

```
- 1 before print "One\n";
```

One

val it = 1 : int

```
- ignore 3;
```

val it = () : unit

Booleans : `eqtype bool`

`bool` type contains `true` and `false`.

The following negation function is defined:

```
val not : bool -> bool
```

In addition, the following special forms are provided:

exp_1 `andalso` exp_2 (conjunction)
exp_1 `orelse` exp_2 (disjunction)
`if` exp `then` exp_1 `else` exp_1 (conditional)

These special forms suppress unnecessary evaluation.

- exp_1 `andalso` exp_2 evaluates exp_2 only if exp_1 is true.
- exp_1 `orelse` exp_2 evaluates exp_2 only if exp_1 is false.
- `if` exp_1 `then` exp_2 `else` exp_3 evaluate only one of exp_2 or exp_3 and not both.

Integers: `eqtype int`

`int` represents 2's complement representations of integers in $-2^{30} \leq n \leq 2^{30} - 1$. One bit less than in other languages is current ML's another defect.

Integer literals

- usual decimal forms e.g. `123` etc.
- hexadecimal notations of the form `0x`nnn where n is digit (`0` `9`) or letters from `a` (or `A`) to `f` (or `F`).

Negative literals are written as `~123`:

```
- 123;
val it = 123 : int
- ~0x10;
val it = ~16 : int
```

Primitive operations on integers.

```
exception Overflow                  (overflow exception)
exception Div                       (division by zero)
val ~ : int -> int                  (negation)
val * : int * int -> int
val div : int * int -> int
val mod : int * int -> int
val + : int * int -> int
val - : int * int -> int
val >  : int * int -> bool
val >= : int * int -> bool
val <  : int * int -> bool
val <= : int * int -> bool
val abs : int -> int
```

Reals : `type real`

`real` is type of real (rational) numbers in floating point representations.

`real` literals

- digit sequence containing decimal points e.g. `3.14`
- scientific expression e.g. $x\mathrm{E}n$ for $x \times 10^n$.

```
- 3.14;
```
val it = 3.14 : real

```
- val C = 2.9979E8
```
val it = 299790000.0 : real

```
val ~ : real -> real              (negation)
val + : (real * real) -> real
val - : (real * real) -> real
val * : (real * real) -> real
val / : (real * real) -> real
val > : (real * real) -> bool
val < : (real * real) -> bool
val >= : (real * real) -> bool
val <= : (real * real) -> bool
val abs : real -> real
val real : int -> real
val floor : real -> int
val ceil : real -> int
val round : real -> int
val trunc : real -> int
```

Some simple examples:

```
- val a = 1.1 / 0.0;
```
val a = inf : real
```
- a * ~1.0;
```
val it = ~inf : real
```
- a / it;
```
val it = nan : real

where

- nan : "not a number" constant
- inf : infinity

Characters : `eqtype char`

ASCII representation of characters.

Character literals: `#"c"` where c may be one of

`\a`	`worning(ASCII 7)`
`\b`	`backspace(ASCII 8)`
`\t`	`horizontal tab(ASCII 9)`
`\n`	`new line(ASCII 10)`
`\v`	`vertical tab(ASCII 11)`
`\f`	`home feed (ASCII 12)`
`\r`	`carrige return(ASCII 13)`
`\^`c	`control character` c
`\"`	`character "`
`\\`	`character \`
`\`ddd	`the character whose code is` ddd `in decimal`

Operations on characters:

```
exception Chr
val chr : int -> char
val ord : char -> int
val str : char -> string
val <= : char * char -> bool
val <  : char * char -> bool
val >= : char * char -> bool
val >  : char * char -> bool
```

Strings : `eqtype string`

String literals : "...." which may contain special character literals and \.

Multiple line notation:

```
- "This is a single \
    \string constant.";
```

val it = "This is a single string constant." : string

```
exception Substring
val size : string -> int
val substring : string * int * int -> string
val explode : string -> char list     (to a list of characters)
val implode : char list -> string     (from a list of characters)
val concat : string list -> string
val <= : string * string -> bool
val <  : string * string -> bool
val >= : string * string -> bool
val >  : string * string -> bool
val ^  : string * string -> string     (concatenation)
val print : string -> unit
```

```
- "Standard ML";
```

val it = "Standard ML" : string

```
- substring(it,9,2);
```

val it "ML" : string

```
- explode it;
```

val it = [#"M",#"L"] : char list

```
- map (fn x => ord x + 1) it;
```

val it = [78,77] : int list

```
- map chr it;
```

val it = [#"N",#"M"] : char list

```
- implode it;
```

val it = "NM" : string

```
- it < "ML";
```

val it = false : bool

Programming Example

Problem: find a maximal common substring of two strings s_1 and s_2.

Data representation:

Represent a common substring of s_1 and s_2 as $(start_1, start_2, l)$. $start_1$ and $start_2$ are the starting positions of s_1 and s_2

We search all the possible pairs $(start_1, start_2)$ and find a maximal l.

Search strategy:

Maintaining the maximum substring $(start_1, start_2, max)$ found so far, and update this information during the search.

Procedure:

1. Set the starting position $(from_1, from_2)$ to $(0, 0)$, and set the maximum common substring to $(start_1, start_2, max)$ to $(0, 0, 0)$.
2. Repeat the following process.
   - 2.1. If the position $(from_1, from_2)$ is out of the strings, then we are done. Return the maximum substring $(start_1, start_2, max)$ so far found as the result.
   - 2.2. If $(from_1, from_2)$ is within the strings, then computes the length of the longest substring starting from this position.
   - 2.3. If $n > max$ then update $(start_1, start_2, max)$ to $(from_1, from_2, n)$.
   - 2.4. Update the starting position $(from_1, from_2)$ to the next position, and continue.

```
fun match s1 s2 =
  let val maxIndex1 = size s1 - 1
      val maxIndex2 = size s2 - 1
      fun nextStartPos (i,j) = ···
      fun findMatch (from1,from2) (start1,start2,max) =
          if from1 > maxIndex1 orelse from2 > maxIndex2 then
             (start1,start2,max)
          else let fun advance n = ···
                   val newSize = advance 0
               in  if max < newSize then
                      findMatch (nextStartPos(from1,from2))
                                (from1,from2,newSize)
                   else findMatch (nextStartPos(from1,from2))
                                  (start1,start2,max)
               end
  in findMatch (0,0) (0,0,0) end
```

RECORDS

Record Types and Record Expressions

Syntax of record types

$\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$

where

- $l_1, \cdots, l_n$ are pairwise distinct labels,
- a label l_i can be either an identifier or a numbers, and
- $\tau_1, \cdots, \tau_n$ can be any types.

Example

```
type malt = {Brand:string, Distiller:string,
             Region:string, Age:int}
```

Syntax of record expressions

$\{l_1$ `=` exp_1`,`$\cdots$`,`l_n `=` $exp_n\}$

where exp_i can be any expression.

Typing rule for records:

$$\frac{exp_1 : \tau_1 \qquad \cdots\ exp_n : \tau_n}{\{l_1 = exp_1, \cdots, l_n = exp_n\} : \{l_1 : \tau_1, \cdots, l_n : \tau_n\}}$$

```
- val myMalt= {Brand = "Glen Moray",
               Distiller = "Glenlivet",
               Region = "the Highlands", Age = 28};
```

val myMalt = {Age = 28, Brand = "Glen Moray",
Distiller = "Glenlivet",
Region = "the Highlands" }
: {Age : int, Brand : string, Distiller : string, Region : string}

According to our principle:

expressions can be freely combined as far as they are type correct.

records are freely combined with any other constructs:

```
- fun createGlen'sMalt (name,age) =
      {Brand = name, Distiller = "Glenlivet",
       Region = "the Highlands", Age = age};
```

val createGlen'sMalt =
   *fn : 'a * 'b* -> *{Age:'b, Brand:'a, Distiller:string, Region:string}*

Operations on Records

$\#l\ \ e$ extract the l field of e.

```
- #Distiller myMalt;
val it = "Glenlivet" : string
- fun oldMalt (x:{Brand:'a, Distiller:'b,
                 Region:'c, Age:int}) =
      #Age x > 18;
val oldMalt = fn : {Age:int, Brand:'a, Distiller:'b, Region:'c} -> bool
- oldMalt myMalt;
val it = true : bool
```

Functions with record patters of the form:

$\{l_1$ = pat_1,$\cdots$,l_n = $pat_n\}$
$\{l_1$ = pat_1,$\cdots$,l_n = pat_n, . . .$\}$

extract record fields.

```
- fun oldMalt {Brand, Distiller, Region, Age} = Age > 18
```
val oldMalt = fn : {Age:int, Brand:'a, Distiller:'b, Region:'c} -> *bool*

```
- val {Brand = brand, Distiller = distiller,
       Age = age, Region = region} = myMalt;
```
val age = 28 : int
val brand = "Glen Moray" : string
val distiller = "Glenlivet" : string
val region = "the Highlands" : string
```
- val {Region = r, ...} = myMalt;
```
val r = "the Highlands" : string
```
- val {Region, ...} = myMalt;
```
val Region = "the Highlands" : string
```
- val distiller = (fn {Distiller,...} => Distiller) myMalt;
```
val distiller = "Glen Moray" : string
```
- fun getRegion ({Region, ...}:malt) = Region;
```
val getRegion = fn : {Age:'a, Brand:'b, Distiller:'c, Region:'d } -> 'd
```
- getRegion myMalt;
```
val it = "the Highlands" : string

Typing Restriction

The Current ML implementation cannot infer a polymorphic type for function with record operations. So

```
fun name {Name = x, Age =a} = x
```

is ok but

```
fun name x = #Name x
```

stdIn:17.1-17.21 Error: unresolved flex record
(can't tell what fields there are besides #name)

Note: This is a defect of the current ML.

In the language we are developing, you can do

```
- fun name x = #Name x;
```

val name : ['a,'b.'a#Name:'b -> 'b]

Tuples

Tuples such as

```
- val p = (2,3) ;
```
*val p = (2,3) : int * int*

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

```
- f p ;
```
val it = 5 int

are special case of records

tuple notations	the corresponding record expressions
(exp_1, exp_2, $\cdots$, exp_n)	{1=exp_1, 2=exp_2, $\cdots$, n=exp_n}
τ_1 * τ_2 * $\cdots$ * τ_n	{1:τ_1, 2:τ_2, $\cdots$, n:τ_n}
(pat_1, pat_2, $\cdots$, pat_n)	{1=pat_1, 2=pat_2, $\cdots$, n=pat_n}

Indeed you can write:

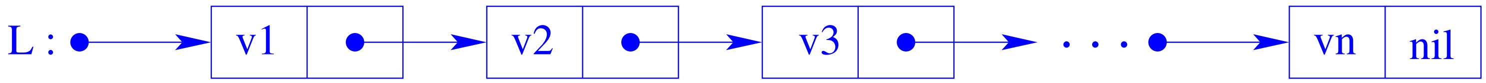
```
- #2 p;
```
val it = 3 : int
```
- val {1=x,2=y} = p;
```
val x = 2

val y = 3
```
- f {1=1,2=2};
```
val it = 3 : int

PROGRAMMING WITH LISTS

List Structure

A list is a finite sequence of values connected by pointers:



From this structure, one can see the following properties:

1. A list is represented by a pointer, irrespective of its length.
2. The empty list is a special pointer called `nil`.
3. Elements in a list is accessed from the top by traversing the pointers.
4. Removing the top element from a list results in a list; adding an element to an list results in a list.

Mathematical Understanding of Lists

A list can be regarded as a nested pair of the form

$$(\mathrm{v1}, (\mathrm{v2}, \cdots (\mathrm{vn}, \mathrm{nil}) \cdots))$$

Let A be the set from which elements $v_1, v_2, \ldots, v_n$ are taken. Also let Nil to be the set $\{nil\}$. Define cartesian product $A \times B$ of two sets A, B as

$$A \times B = \{(a, b) | a \in A, b \in B\}$$

Then the set of n elements lists can be the following set:

$$\underbrace{A \times (A \times (\cdots (A}_{n\text{個の}A} \times Nil) \cdots))$$

So the set of all finite lists can be given as a solution to the following equation on sets:

$$L = Nil \cup (A \times L)$$

We can obtain the solution in the following steps:

1. Define a sequence of sets X_i as follows:

$$X_0 = Nil$$
$$X_{i+1} = X_i \cup (A \times X_i)$$

   If the above equation has a solution for L, then we can verify that for each i

$$X_i \subseteq L$$

2. Using this sequence of sets. we define

$$X = \bigcup_{i \geq 0} X_i$$

   It is easily verify that this set satisfies the equation, and therefore a solution of the equation.

List Type : τ `list`

τ list is a type of lists of element of type τ where τ can be any type. For example:

- `int list` : integer lists.
- `int list list` : lists of integer lists.
- `(int -> int) list` : lists of integer functions.

Using these constructors, list of elements $v_1, \cdots, v_n$ is written as

v_1 `::` v_2 `::` $\cdots$ `::` v_n `::` `nil`

The following shorthand (syntactic sugar) is also supported.

$$\texttt{[]} \Longrightarrow \texttt{nil}$$

$$\texttt{[}exp_1\texttt{,}exp_2\texttt{,}\cdots\texttt{,}exp_n\texttt{]} \Longrightarrow exp_1 \texttt{ :: } exp_2 \texttt{ :: } \cdots \texttt{ :: } exp_n \texttt{ :: nil}$$

Simple list expressions:

```
- nil;
val it = [] : 'a list
- 1 :: 2 :: 3 :: nil;
val it = [1,2,3] : int list
- [[1],[1,2],[1,2,3]];
val it = [[1],[1,2],[1,2,3]] : int list list
- [fn x => x];
val it = [fn] : ('a -> 'a) list
```

Simple List Creation

An example: create a list of `[1,2,3,4,...,`n`]` for a give n.

The first try:

```
- fun mkList n = if n = 0 then nil
                 else n ::  f (n - 1);
```
val mkList = fn : int `->` *int list*
```
- mkList 3;
```
val it = [3,2,1] : int list

The second try

```
- fun mkList n m = if n = 0 then nil
                   else (m - n) ::  f (n - 1) m
```
val mkList = fn : int `->` *int*`->` *int liist*
```
- mkList 3 4;
```
val it = [1,2,3] : int list

A better solution:

```
- fun mkList n =
      let
         fun f n L = if n = 0 then L
                     else f (n - 1) (n::L)
      in
         f n nil
      end;
```

val mkList = fn : int `->` *intlist*

```
- mkList 3;
```

val it = [1,2,3] : int list

This is clearer and efficient.

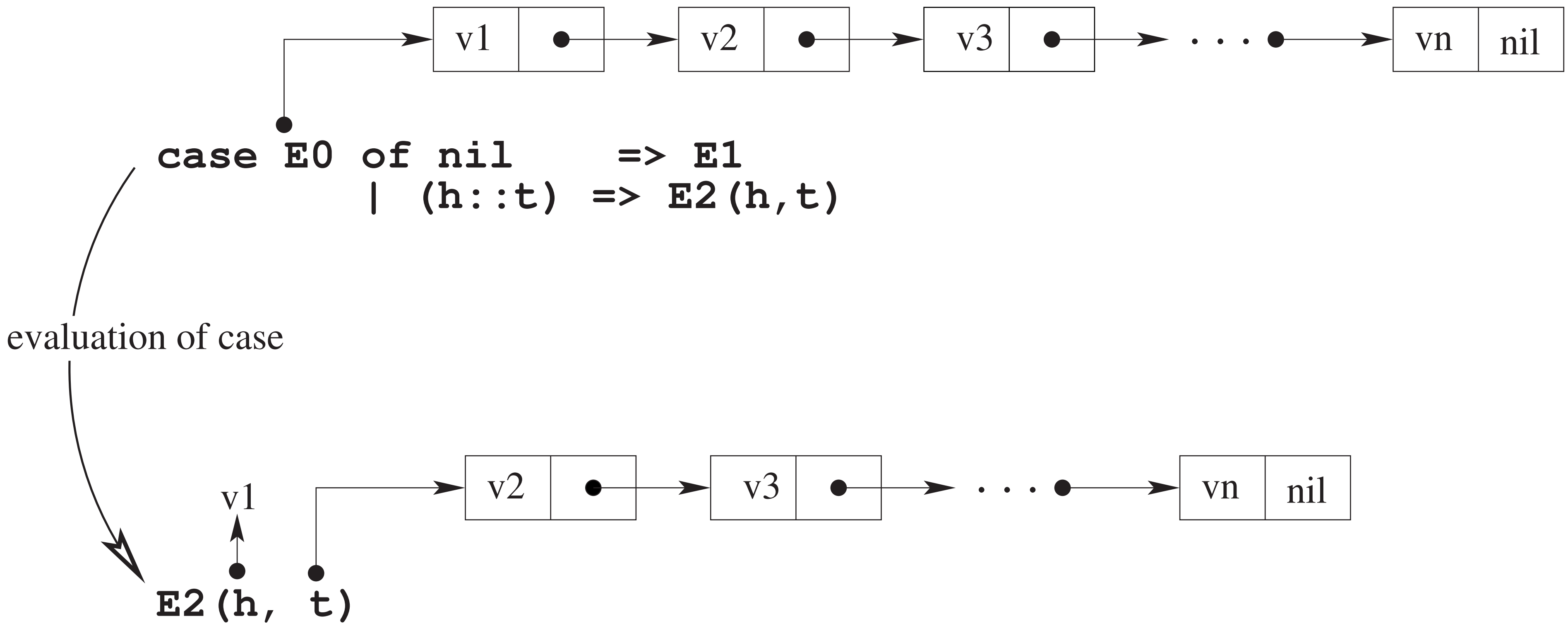
Decomposing a List with Patters

The basic operation on a list is pattern matching:

```
case E0 of nil => E1
         | (h::t) => E2(h,t)
```

which performs the following actions

1. evaluate E_0 and obtain a list L
2. if L is `nil` then evaluate $E1$.
3. if L is of the form `h::t`, then evaluate $E2(\mathtt{h}, \mathtt{t})$ under the binding where `h` is the head of L and t is the tail of L.


v1
v2
v3
vn
nil
case E0 of nil => E1
| (h::t) => E2(h,t)
evaluation of case
v1
v2
v3
vn
nil
E2(h, t)

```
- case nil of nil => 0 | (h::t) => h;
```
val it = 0 : int

```
- case [1,2] of nil => 0 | (h::t) => h;
```
val it = 1 : int

A simple program using pattern matching:

```
fun length L = case L of nil => 0
                       | (h::t) => 1 + length t;
```

The general form of pattern matching

`case` exp `of` pat_1 `=>` exp_1 `|` pat_2 `=>` exp_2 `|` $\cdots$ `|` pat_n `=>` exp_n

where pat_i is a pattern consisting of

- constants,
- variables,
- data structure construction (e.g. list). For lists, we can include patterns of the forms:
  - `nil`,
  - pat_1::pat_2,
  - `[`pat_1`,`$\cdots$`,`pat_n`]`

```
fun zip x = case x of (h1::t1,h2::t2) =>
                          (h1,h2) :: zip (t1,t2)
                    | _ => nil
fun unzip x = case x of (h1,h2)::t =>
                            let val (L1,L2) = unzip t
                            in (h1::L1,h2::L2)
                            end
                      | _ => (nil,nil)
```

where "_ " is the anonymous pattern matching any value.

Patterns can overlap and can be non-exhaustive.

```
fun last L = case L of [x] => x
                     | (h::t) => last t
```

Useful shorthand:

$$\begin{array}{l} \texttt{fun}\ f\ \ pat_1\ \texttt{=}\ exp_1 \\ \quad \texttt{|}\ f\ \ pat_2\ \texttt{=}\ exp_2 \\ \quad \vdots \\ \quad \texttt{|}\ f\ \ pat_n\ \texttt{=}\ exp_n \end{array} \quad \Longrightarrow \quad \begin{array}{l} \texttt{fun}\ f\ \texttt{x = case x of} \\ \qquad\qquad\quad\ \ pat_1\ \texttt{=>}\ exp_1 \\ \qquad\qquad\quad \texttt{|}\ pat_2\ \texttt{=>}\ exp_2 \\ \qquad\qquad\quad \vdots \\ \qquad\qquad\quad \texttt{|}\ pat_n\ \texttt{=>}\ exp_n \end{array}$$

$$\begin{array}{l} \texttt{fn}\ pat_1\ \texttt{=>}\ exp_1 \\ \ \ \texttt{|}\ pat_2\ \texttt{=>}\ exp_2 \\ \ \ \vdots \\ \ \ \texttt{|}\ pat_n\ \texttt{=>}\ exp_n \end{array} \quad \Longrightarrow \quad \begin{array}{l} \texttt{fn x => case x of} \\ \qquad\qquad\quad\ \ pat_1\ \texttt{=>}\ exp_1 \\ \qquad\qquad\quad \texttt{|}\ pat_2\ \texttt{=>}\ exp_2 \\ \qquad\qquad\quad \vdots \\ \qquad\qquad\quad \texttt{|}\ pat_n\ \texttt{=>}\ exp_n \end{array}$$

Some examples:

```
fun length nil = 0
  | length (h::t) = 1 + length t

fun fib 0 = 1
  | fib 1 = 1
  | fib n = fib (n - 1) + fib (n - 1)
```

Built-in Functions for Lists

```
infixr 5 @
exception Empty
val null : 'a list -> bool
val hd : 'a list -> 'a
val tl : 'a list -> 'a list
val @ : 'a list * 'a list -> 'a list
val rev : 'a list -> 'a list
val length : 'a list -> int
val map : ('a -> 'b) -> 'a list -> 'b list
```

```
- map (fn x => (x,x)) [1,2,3,4];
```

*val it = [(1,1),(2,2),(3,3),(4,4)] : int * int list*

General List Processing Function

Consider the function definition:

```
fun sumList nil = 0
  | sumList (h::t) = h + sumList t
```

This can be regarded as a special case of the following procedure.

1. If the give list is `nil` then return some predefined value Z.
2. If the list is of the form $h::t$ then compute a value for t and obtain the result R.
3. Compose the final result from h and R by applying some function f.

Indeed, if we take $Z = 0$ and $f(h, R) = h + R$ then we get `sumList` above.

The following higher-function perform this general computation.

```
fun foldr f Z nil = Z
  | foldr f Z (h::t) = f(h,foldr f Z t)
```

Individual list processing functions can be obtained by specifying f and Z as

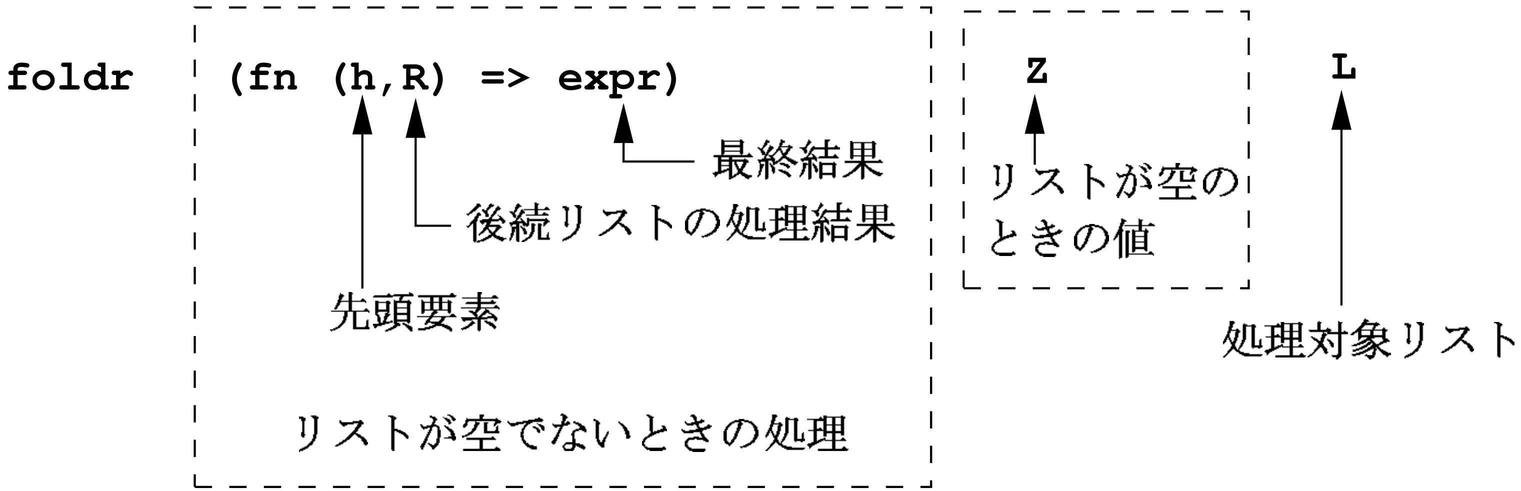
`foldr (fn (`h`,`R`) =>` exp`)` Z

where

- Z is the value for `nil`.
- `fn (`h`,`R`) =>` exp computes the final result form the result R of the tail list and the head.

```
- val sumList = foldr (fn (h,R) => h + R) 0 ;
```

val sum : int list -> int



Programming Example

The transitive closure R^+ of a given finite relation R is defined as:

$$R^+ = \{(x, y) | \exists n \exists z_1 \cdots \exists z_n \, x = z_1, y = z_n, (z_1, z_2) \in R, \cdots, (z_{n-1}, z_n) \in R\}$$

We develop a program to compute the transitive closer of R in the following steps.

1. Rewrite the definition of R^+ to a constructive one.
   Considering n of elements in R^+, R^+ can be re-written as:
   $$R^+ = R^1 \cup R^2 \cup \cdots \cup R^N$$
   where N can be take as the largest possible one.
2. Give a recursive definition of R^k.

$$\begin{aligned} R^1 &= R \\ R^k &= R \times R^{k-1} \qquad (k \geq 2) \end{aligned}$$

where $\times$ is the following operation

$$R \times S = \{(x, y) | \exists a \ (x, a) \in R, (a, y) \in S\}$$

3. Write a function `timesRel` and `powerRel` to compute $R \times S$ and R^n respectively.

```
fun timesRel (R,S) =
    foldr (fn ((x,a),r) =>
                foldr (fn ((b,y),rs) =>
                           if a=b then (x,y)::rs
                           else rs)
                r S)
    nil R;
fun powerRel r 1 = r
  | powerRel r n = timesRel (r,powerRel r (n - 1));
```

4. Use `accumulate` to define R^+.
   $R^+ = R^1 \cup \cdots \cup R^N$ is $\Lambda_{k=1}^{N}(h, f, z) = h(f(N), \cdots, h(f(1), z) \cdots)$
   with h to be the set union and $f(k) = R^k$, $z = \emptyset$.

```
fun tc R =
    accumulate (op @) nil (powerRel R) (length R)
```

```
- tc [(1,2),(2,3),(3,4)];
```

*val it = [(1,4),(1,3),(2,4),(1,2),(2,3),(3,4)]: (int * int) list*

Exercise Set (4)

1. Define the following functions directly by recursion.

   (1) `sumList` to compute the sum of a given integer list.

   (2) `member` to test whether a given element is in a given list.

   (3) `unique` to remove duplicate elements from a given list.

   (4) `filter` that takes a function P of type `'a -> bool` and a list of type `'a list` and return the list of elements for which P returns `true`.

   (5) `flatten` to convert a list of lists to a list as follows:

   ```
   - flatten [[1],[1,2],[1,2,3]];
   ```
   val it = [1,1,2,1,2,3] : int list

   (6) `splice` that takes a list L of strings and a string d and return the string obtained by concatenating the strings in L using d as a delimiter. For example, you should have:

```
- splice (["","home","ohori","papers","mltext"],"/");
```
val it = "/home/ohori/papers/mltext" : string

2. Define the following functions using `foldr`.

   (1) Each of the following `map`, `flatten`, `member`, `unique`, `prefixList`, `permutations`

   (2) `forall` which takes a list and a predicate P (a function that return `bool`) and checks whether the list contains an element that is true of P, and
   `exists` which takes a list and a predicate P (a function that return `bool`) and check whether all the elements of the list is true of P By definition, for any P, `exists` P `nil` is `false` and `forall` P `nil` is `true`.

   2.1. `prefixSum` which computes the prefix sum of a given integer lists. Here, the prefix sum of $[a_1, a_2, \cdots, a_n]$ is

$$[a_1, a_1 + a_2, \cdots, a_1 + \cdots a_{n-1}, a_1 + \cdots + a_n].$$

3. `foldr` performs the following computation

$$\texttt{foldr}\ f\ \ Z\ \ [a_1, a_2, \cdots, a_n] = f(a_1, f(a_2, f(\cdots, f(a_n, Z)\cdots)))$$

SML also provides `foldl` that performs the following computation.

$$\texttt{foldr}\ f\ \ Z\ \ [a_1, \cdots, a_{n-1}, a_n] = f(a_n, f(a_{n-1}, f(\cdots, f(a_1, Z)\cdots)))$$

(1) Give a definition of `foldl`.

(2) Define `rev` using `foldl`.

4. We say that (a representation of) a relation R is in *normal form* if R does not contain duplicate entry.

(1) Give an example of R in normal form such that `tc` R is not in normal form.

(2) Rewrite `tc` so that it always return a normal relation for any normal input.

Define the following functions on relations:

(1) `isRelated` which test whether a given pair (a, b) is related in a given R.

(2) `targetOf` which returns the set $\{x \mid (a, x) \in R\}$ for a given relation R and a given point a.

(3) `sourceOf` which returns the set $\{x \mid (x, a) \in R\}$ for a given R and a.

(4) `inverseRel` to compute the inverse R^{-1} of R.

DATATYPE DEFINITIONS

`Datatype` Statement Define a New Type

Example: Binary Trees

The set of binary trees over τ is defined inductively as:

1. The empty tree is a binary tree.
2. If v is a value of τ and T_1 and T_2 are binary trees over τ, then (v, T_1, T_2) is a binary tree.

This is defined as:

```
- datatype 'a tree =
           Empty
         | Node of 'a * 'a tree * 'a tree;
```

*datatype 'a tree = Empty | Node of 'a * 'a tree * 'a tree*

After this definition, `Empty` and `Node` are used as data constructors.

```
- Empty;
```

val it = Empty : 'a tree

```
- Node;
```

*val it = fn : 'a * 'a tree * 'a tree -> 'a tree*

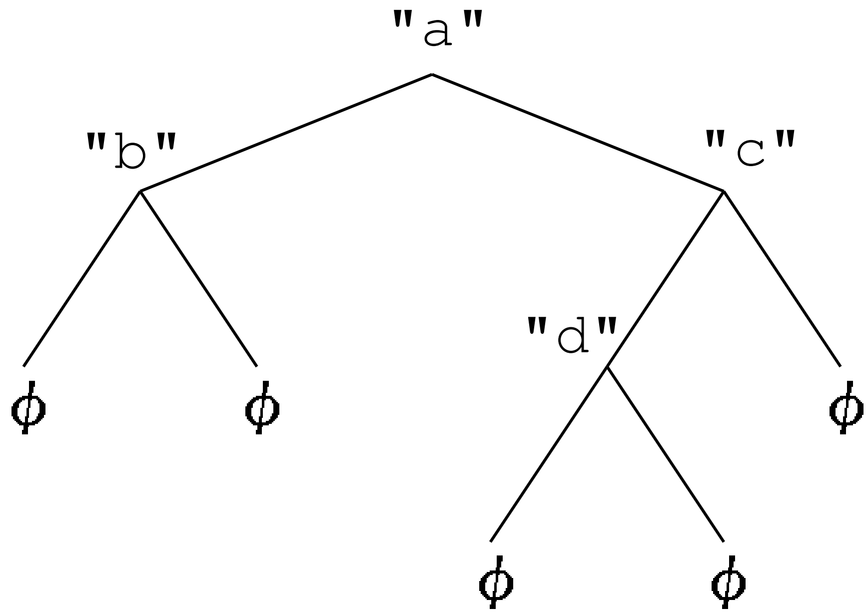
```
- Node (1,Empty,Empty);
```

val it = Node (1,Empty,Empty) : int tree

```
- Node ((fn x => x),Empty,Empty);
```

val it = Node (fn,Empty,Empty) : ('a -> 'a) tree

```
Node("a",Node("b",Empty,Empty),
         Node("c",Node("d",Empty,Empty),Empty))
```



(ϕは空の木を表す)

The pre-order representation `a(b()())(c(d()())())`.

Constructing a binary tree from a string-encoded tree.
pre-order encoding of a tree

- `Empty` is represented as the empty string.
- `Node(`a`,`L`,`R`)` is represented as $a(S_L)(S_R)$ where S_L and S_R are pre-order representation of R and L.

This representation is obtained by traversing a tree in the following order:

1. the root,
2. the left subtree,
3. the right subtree.

We can write down a tree construction function from a pre-order encoding as:

```
fun fromPreOrder s =
   let fun decompose s = ...
            (* decompose a string regarding ‘(’ and ‘)’ as delimiters.*)
   in if s = "" then Empty
      else let val (root,left,right) = decompose s
            in Node(root,fromPreOrder left,fromPreOrder right)
   end
```

`decompose` performs the following action.

```
- decompose "a(b()())(c(d()())())";
val it = ("a","b()()","c(d()())()") : string * string * sting
```

```
fun decompose s =
    let fun searchLP s p = ...
            (* return the first left paren from the position p.  *)
        fun searchRP s p n = ...
            (* return the position of the nth right paren from p *)
        val lp1 = searchLP s 0
        val rp1 = searchRP s (lp1+1) 0
        val lp2 = searchLP s (rp1+1)
        val rp2 = searchRP s (lp2+1) 0
    in (substring (s,0,lp1),
       substring (s,lp1+1,rp1-lp1-1),
       substring (s,lp2+1,rp2-lp2-1))
    end
```

The general structure of `datatype`

$$
\begin{array}{lll}
\texttt{datatype} & \multicolumn{2}{l}{typeSpec\ \texttt{=}} \\
 & & Con_1\ \langle\texttt{of}\ \ type_1\rangle \\
 & \texttt{|} & Con_2\ \langle\texttt{of}\ \ type_2\rangle \\
 & & \vdots \\
 & \texttt{|} & Con_n\ \langle\texttt{of}\ \ type_n\rangle
\end{array}
$$

- $typeSpec$ specify the type to be defined
- the right hand side of = specify the type of each component

Using Data Structures with Patter Matching

In `case` expression of the form

`case` exp `of` pat_1 `=>` exp_1 `|` pat_2 `=>` exp_2 `|` $\cdots$ `|` pat_n `=>` exp_n

pat_i can contain:

1. variables,
2. constants,
3. any data constructors defined in `datatype` declarations,
4. anonymous pattern `_`

```
- case Node ("Joe",Empty,Empty) of Empty => "empty"
                                 | Node (x,_,_) => x;
```

val it = "Joe" : string

Computing the height of a tree:

1. the height of the empty tree (`Empty`) is 0.
2. the height of a tree of the form `Node(a,`L`,`R`)` is 1 + max(the height of R, the height of L)

So we can code this as:

```
- fun height t =
      case t of Empty => 0
              | Node (_,t1,t2) => 1 + max(height t1, height t2
```

val height = fn : 'a tree -> int

Some example using pattern matching

```
fun height Empty = 0
  | height (Node(_,t1,t2)) = 1 + max(height t1, height t2)

fun toPreOrder Empty = ""
  | toPreOrder (Node(s,lt,rt)) =
     s ^ "(" ^ toPreOrder lt ^ ")"
     ^ "(" ^ toPreOrder rt ^ ")"
```

System Defined Data Types

Lists

```
infix 5 ::
datatype 'a list = nil | :: of 'a * 'a list
```

Blooleans

```
datatype bool = true | false
```

Special forms for `bool` are defined as:

exp_1 `andalso` exp_2 $\Longrightarrow$ `case` exp_1 `of true =>` exp_2
`| false => false`

exp_1 `orelse` exp_2 $\Longrightarrow$ `case` exp_1 `of false =>` exp_2
`| true => true`

`if` exp `then` exp_1 `else` exp_2 $\Longrightarrow$ `case` exp `of true =>` exp_1
`| false =>` exp_2

Other system defined types:

```
datatype order = EQUAL | GREATER | LESS
datatype 'a option = NONE | SOME of 'a
exception Option
val valOf : 'a option -> 'a
val getOpt : 'a option * 'a -> 'a
val isSome : 'a option -> bool
```

Programming Examples : Dictionary

```
type 'a dict = (string * 'a) tree
val enter : string * 'a * 'a dict -> 'a dict
val lookUp : string * 'a dict -> 'a option
```

- `enter` returns a new dictionary by extending with a given entry.
- `lookUp` returns the value of a given key (if exists).

To implement a dictionary efficiently, we use a binary tree as a binary search tree where the following property hold: for every node of the form `Node(`key`,`L`,`R`)`

1. any key in L is smaller than key, and
2. any key in R is larger than key.

```
fun enter (key,v,dict) =
    case dict of
         Empty => Node((key,v),Empty,Empty)
       | Node((key',v'),L,R) =>
       if key = key' then dict
       else if key > key' then
            Node((key',v'),L, enter (key,v,R))
       else Node((key',v'),enter (key,v,L),R)


fun lookUp (key,Empty) = NONE
  | lookUp (key,Node((key',v),L,R)) =
      if key = key' then SOME v
      else if key > key' then lookUp (key,R)
      else lookUp (key,L)
```

Infinite Data Structure

Obviously, we can only deal with finite structure. So

```
fun fromN n = n :: (fromN (n+1));
```

is useless.

The key technique: delay the evaluation until requested.

the mechanism to delay evaluation : `fn () =>` exp

Simple example of delaying evaluation:

```
fun cond c a b = if c then a () else b();
```

val cond = fn : bool `->` *(unit* `->` *unit)*`->` *(unit* `->` *unit)*`->` *unit*

```
cond true (fn () => print "true") (fn () => print "false");
```

a datatype for infinite lists:

```
datatype 'a inflist =
         NIL | CONS of 'a * (unit -> 'a inflist)
```

Examples:

```
fun FROMN n = CONS(n,fn () => FROMN (n+1));
- FROMN 1;
```

val it = CONS (1,fn) : int inflist

```
- val CONS(x,y) = it;
```

val x = 1 : int

val y = fn : unit -> int inflist

```
- y ();
```

val it = CONS (2,fn) : int inflist

Functions for `inflist`

```
fun HD (CONS(a,b)) = a
fun TL (CONS(a,b)) = b()
fun NULL NIL = true | NULL _ = false
```

Examples:

```
- val naturalNumbers = FROMN 0;
```

val naturalNumbers = CONS (0,fn) : int inflist

```
- HD naturalNumbers;
```

val it = 0 : int

```
- TL naturalNumbers;
```

val it = (1,fn) : int inflist

```
- HD (TL(TL(TL it)));
```

val it = 4 : int

How to design a program on `inflist`:

1. Design a program for ordinary lists using `hd`, `tl` and `null`.
2. Replace list processing primitives as:

```
 hd  ==> HD
 tl  ==> TL
null ==> NULL
h::t ==> CONS(h,fn () => t)
```

Example

```
fun NTH 0 L = HD L
  | NTH n L = NTH (n - 1) (TL L)
```

int -> 'a inflist -> -> 'a

```
- NTH 100000000 naturalNumbers;
```

val it = 100000000 : int

A more complicated example:
For finite lists:

```
fun filter f l = if null l then nil
                 else if f (hd l) then
                   hd l ::  (filter f (tl l))
                 else filter f (tl l);
```

For infinite lists:

```
fun FILTER f l = if NULL l then NIL
                 else if f (HD l) then
                   CONS(HD l,fn () => (FILTER f (TL l)))
                 else FILTER f (TL l);
```

Sieve of Eratosthenes:
For the infinite list of integers starting from 2, repeat the following:

1. Remove the first number and output it.
2. Remove all the elements that are divisible by the first number.

An example code:

```
fun SIFT NIL = NIL
  | SIFT L =
    let val a = HD L
    in CONS(a, fn () =>
                SIFT (FILTER (fn x => x mod a <> 0)
                             (TL L)))
    end
```

```
- val PRIMES = SIFT (FROMN 2);
```
val PRIMES = CONS(2,fn) : int inflist

```
- TAKE 20 PRIMES;
```
val it = [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
: int list;

```
- VIEW (10000,10) PRIMES;
```
val it = [104743,104759,104761,104773,104779,104789,
104801,104803,104827,104831] : int list

IMPERATIVE FEATURES

References

`'a ref` type and `ref` data constructor

```
infix 3 :=
val ref :  'a -> 'a ref
val !  :  'a ref -> 'a
val := :  'a ref * 'a -> unit
```

Simple examples:

```
- val x = ref 1;
```
val x = ref 1 : int ref
```
- !x;
```
val it = 1 : int
```
- x:=2;
```
val it = () : unit
```
- !x;
```

$$val\ it = 2 : int$$

References are imperative data structure, for which the order of evaluation is significant.

The order of evaluation of ML

- In `let val` x_1 `=` exp_1 $\cdots$ `val` x_n `=` exp_n `in` exp `end`, ML evaluates $exp_1, \ldots, exp_n$ in this order and then evaluate exp.
- Tuples `(`exp_1`,`$\cdots$`,`exp_n`)` and records $\{l_1$ `=` exp_1`,`$\cdots$`,` l_n `=` $exp_n\}$ are evaluated left to right.
- For function applications epx_1 exp_2, ML first evaluates exp_1 to obtain a function `fn` x `=>` exp_0, and then evaluates exp_2 to obtain the result v, and finally it evaluates exp_0 with x bound to v.

Constructs for controlling the order of evaluation:

- `(`exp_1`;` $\cdots$`;` exp_n`)`
- exp_1 `before` exp_2
- `while` exp_1 `do` exp_2

Programming Example

`gensym` : a program to create a new name.

- It has type `gensym : unit -> string`.
- It returns a new name every time when it is called.
- It generates alphabet strings in the following order: `"a"`, `"b"`, $\cdots$, `"z"`, `"aa"`, `"ab"`, $\cdots$, `"az"`, `"ba"`, $\cdots$

We use the following data:

- a reference data `state` to represent the last name generated.
- a function `next` to generate a representation of the next string from `state`.
- a function `toString` to convert `state` to the string it represents.

Then we can code the function as:

```
local
  val state = ···
  fun toString = ···
  fun next s = ···
in
   fun gensym() = (state:=next (!state);
                   toString (!state))
end
```

Internal representation of a string:

represent a string $s_n s_{n-1} \cdots s_1$ of length k by the reverse list $[ord(s_1), ord(s_2), \cdots, ord(s_n)]$ of character codes.

`next` "increments" this list as:

```
val state = ref nil :  int list ref
fun next nil = [ord #"a"]
  | next (h::t) = if h = ord #"z" then
                     ord #"a" ::  (next t)
                  else (h+1::t)
```

References and Referential Transparency

The basic principle of mathematical reasoning on programs:

> the meaning of a program is determined by the meaning of its components

In other words,

> the meaning of a program does not change when we replace some part of it with an equivalent one.

This property is called referential transparency, which is a special form of the basis of mathematical reasoning:

> two equal expressions can substitute each other without affecting the meaning of a statement.

For example:

`(fn x => x = x)` exp

and

exp `=` exp

have the same meaning.

However, with references, ML does not have this property,

```
- (fn x => x = x) (ref 1);
```
$val\ it = true : bool$
```
- ref 1 = ref 1;
```
$val\ it = false : bool$

References and Value Polymorphism

`ref` has type `'a -> 'a ref`, but since `ref` *exp* is not a value expression, you cannot make a reference to a polymorphic function.

```
- ref (fn x => x);
```

stdIn:19.1-19.16 Warning: type vars not generalized because of value restriction are instantiated to dummy types (X1,X2,...)
val it = ref fn : (?.X1 -> ?.X1) ref

If we allow

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

then we get a problem such as:

```
val polyIdRef = ref (fn x => x);
polyIdRef := (fn x => x + 1);
(!polyIdRef) "You can't add one to me!";
```

Value polymorphism is introduced to prevent this kind of inconsistency.

Exception Handling

Exception : disciplined "goto".

`exception` defines new exception.

`exception` $exnId$

`exception` $exnId$ `of` τ

`raise` generates exception.

`raise` $exnId$

`raise` $exnId$ exp

Example:

```
- exception A;
exception A
- fun f x = raise A;
val f = fn : 'a -> 'b
- fn x => (f 1 + 1, f "a" andalso true);
val it = fn : 'a -> int * bool
```

System defined exceptions

exception name	meaning
`Bind`	bind failure
`Chr`	illegal character code
`Div`	divide by 0
`Empty`	illegal usage of `hd` and `tl`
`Match`	pattern matching failure
`Option`	empty option data
`Overflow`	
`Size`	array etc too big
`Subscript`	index out of range

$$
\begin{array}{lll}
exp\ \texttt{handle} & exnPat_1 \ \texttt{=>}\ handler_1 \\
 & \texttt{|}\ exnPat_2 \ \texttt{=>}\ handler_2 \\
 & \quad\vdots \\
 & \texttt{|}\ exnPat_n \ \texttt{=>}\ handler_n
\end{array}
$$

Exception handling:

1. An exception condition is detected.
2. The system searches exception handlers in the reverse order of pending evaluation.
3. If it finds a handler, it tries matching the exception with exception patterns
4. If the system finds the matching pattern then it evaluate the corresponding hander and restart the normal evaluation.
5. If no matching pattern is found then the system aborts the evaluation and returns the top level.

```
- exception Undefined;
exception Undefined
- fun strictPower n m = if n = 0 andalso m = 0 then
                             raise Undefined
                          else power n m;
val strictPower = fn : int -> int -> int
- 3 + strict_power 0 0;
uncaught exception Undefined
- 3 + (strictPower 0 0 handle Undefined => 1);
val it = 4 : int
```

Programming Example (1)

```
type 'a dict = (string * 'a ) tree
val enter : (string * 'a) * 'a dict -> 'a dict
val lookUp : string * 'a dict -> 'a option
exception NotFound
fun lookUp (key,Empty) = raise NotFound
  | lookUp (key,Node((key',v),L,R)) =
      if key = key' then v
      else if key > key' then lookUp (key,R)
      else lookUp (key,L)
val lookUp : string * 'a dict -> 'a
fun assoc (nil,dict) = nil
  | assoc ((h::t),dict) =
      (h, lookUp (h,dict)):: assoc (t,ditc)
      handle NotFound => (print "Undefined key."; nil)
```

Programming Example (2)

```
fun lookAll key dictList =
  let exception Found of 'a
      fun lookUp key Empty = ()
        | lookUp key (Node((key',v),L,R)) =
            if key = key' then raise Found v
            else if key > key' then lookUp key R
                 else lookUp key L
      in (map (lookUp key) dictList; raise NotFound)
         handle Found v => v
      end
```

Exercise Set (5)

1. Complete the function `decompose`:

```
- decompose "a(b()())(c(d()())())";
```

$val\ it = (\text{"}a\text{"},\text{"}b()()\text{"},\text{"}c(d()())()\text{"})\ :\ string\ *\ string\ *\ sting$

2. In addition to pre-order encoding, there are also the following encodings for trees:

   - post-order encoding
     This is the string encoding obtained by traversing a tree in the following order:
     (1) the left subtree,
     (2) the right subtree,
     (3) the root.
   - in-order encoding

This is the string encoding obtained by traversing a tree in the following order:

(1) the left subtree,

(2) the root,

(3) the right subtree.

Write the following functions.

- `fromPostOrder` that constructs a tree from a given post-order representation.
- `fromInOrder` that constructs a tree from a given post-order representation.
- `toPostOrder` that return a post-order encoding of a given a tree.
- `toInOrder` that return a post-order encoding of a given a tree.

3. Define the following functions on binary trees.

   - `nodes` that returns the total number of nodes in a given tree.
   - `sumTree` that computes the sum of all the values in a given integer tree.
   - `mapTree : ('a -> 'b) -> 'a tree -> 'b tree` that returns the tree obtained by applying a given function to each node value of a given tree.

4. Analogous to `foldr` for lists, let us define a higher-order function `treeFold` for recursive processing of trees.

   It takes the following arguments:

   (1) a tree t to be processed,

   (2) a value z that should be returned if the given tree is the empty tree,

(3) a function f that computes the final result for a tree of the form `Node(`x`,`L`,`R`)`.

and should have the following structure and type:

```
- fun treeFold f z Empty = z
    | treeFold f z (Node (x,L,R)) = ...
      ...
```

val treeFold = fn : $($*'a* `*` *'b* `*` *'b* `->` *'b*$)$ `->` *'b* `->` *'a tree* `->` *'b*

(1) Complete the definition of `treeFold`.

(2) Re-define `nodes`, `sumTree`, and `mapTree` using `treeFold`.

5. Define the following function of type `'a list -> 'a option` :

(1) `car` that returns the head of a given list.

(2) `cdr` that returns the tail of a given list.

(3) `last` that returns the last element of a given list.

6. Write a function `makeDict : (string * 'a) list -> 'a dict` that return a dictionary consisting of the data in a given list. Test the function `lookUp` using `makeDict`.

7. Define the following functions

```
type ('a,'b) dict = ('a * 'b) tree
val makeEnter : ('a * 'a -> order)
                -> 'a * 'b * ('a,'b) dict
                -> ('a,'b) dict
val makeLookUp : ('a * 'a -> order)
                 -> 'a * ('a,'b) dict -> 'b
```

where `makeEnter` takes a function to compare two keys and returns a function that enters a key-value pair to a dictionary, and `makeLookup` takes a function to compare two keys and returns a function that looks up a dictionary.

8. Define `enter` and `lookUp` functions for a dictionary of type `(int,string) dict` where keys are integers and values are strings. Test your functions.

9. Define `evenNumbers` of infinite list of even numbers from `naturalNumbers` using `FILTER`, and do some simple tests. For example, you should have the following:

```
- NTH 10000000 evenNumbers;
```
val it = 20000000 : int

10. Define the following functions on infinite lists:

    - `DROP : int -> 'a inflist -> 'a inflist` that returns the list obtained from a given infinite list by removing the first n elements.

- `TAKE : int -> 'a inflist -> 'a list` that returns the list of the first n elements in a given infinite list.
- `VIEW : int * int -> 'a inflist -> 'a list` that returns the list of m elements starting from the nth the element in a given infinite list.

11. Complete the definition of `genSym`.

12. Define a function

```
makeGensym  : char list -> unit -> string
```

that takes a list of characters and returns a function that generates a string in the order similar to gensym using the given characters. For example, `makeGensym [#"S",#"M",#"L"]` will return a function that generates `"S"`, `"M"`, `"L"`,
`"SS"`, `"SM"`, `"SL"`, `"MS"`, `"MM"`, `"ML"`, `"LS"`, $\cdots$.

13. Change the definition of `enter` so that it raises an exception `DuplicateEntry` if the value being entered is already in a given tree.

14. Write a function that computes the product of all the elements in a given integer list. Your function should terminates the process as soon as it detects 0.

MODULE SYSTEM

Define a module using `structure`

```
structure id = struct
                   various definitions
               end
```

where definitions can contain:

- `val` declarations
- `fun` declarations
- `exception` declarations
- `datatype` declarations
- `type` declarations
- `structure` declarations

```
structure IntQueue = struct
  exception EmptyQueue
  type queue = int list ref
  fun newQueue() = ref nil :  queue
  fun enqueue (item,queue) =
      queue := item ::  (!queue)
  fun removeLast nil = raise EmptyQueue
    | removeLast [x] = (nil,x)
    | removeLast (h::t) =
      let val (t',last) = removeLast t
      in (h::t',last)
      end
  fun dequeue queue =
      let val (rest,last) = removeLast (!queue)
      in (queue:=rest; last)
      end
end
```

A module has a signature

The following `signature` (type information) is inferred for `IntQueue`:

```
structure IntQueue :
  sig
    type queue = int list ref
    exception EmptyQueue
    val dequeue : 'a list ref -> 'a
    val enqueue : 'a * 'a list ref -> unit
    val newQueue : unit -> queue
    val removeLast : 'a list -> 'a list * 'a
  end
```

Using a Module

Two ways to use a module:

1. Explicitly specifying structure name.
   id_2 in a module named id_1 is called as "id_1.id_2". If structure definitions are nested, then specify the nested sequence as $id_1 . id_2 \cdots id_{n-1} . id_n . id$.

```
- val q = IntQueue.newQueue();
val q = ref [] : queue
- map (fn x => IntQueue.enque(x,q)) [1,3,5];
val it = [(),(),()] : unit list
- IntQueue.dequeue q;
val it = 1 : int
```

2. Using the open primitive
   The effect of `open` id is to import all the component defined in the structure id to the current environment.

```
- open IntQueue;
opening IntQueue
    type queue = int list ref
    exception EmptyQueue
    val newQueue : unit -> queue
    val enqueue : 'a * 'a list ref -> unit
    val dequeue : 'a list ref -> 'a
    val removeLast : 'a list -> 'a list * 'a
- dequeue();
val it = 3 : int
```

Example of module programming: functional queue that replace `IntQueue`.

A functional queue $= (newItems$ list, $oldItems$ list$)$

- $newItems$: the elements recently added to the queue
- $oldItems$: the old elements that are removed soon.
- and the system maintain the invariant

entire queue $=$ $newItems$ @ the reversal of $oldItems$

so that if

$$\begin{aligned} newItems &= [a_1, \cdots, a_n] \\ oldItems &= [b_1, \cdots, b_m] \end{aligned}$$

then

$$L = [a_1, \cdots, a_n, b_m, \cdots, b_1]$$

- `enqueue` adds a an element to the front of $newItems$ list.
- `dequeue` perform the following action depending on the status of the queue.
  - if $oldItems \neq nil$ then removes its head and returns it.
  - if $oldItems = nil$ and $newItems \neq nil$ then rearrange the queue so that it become

    (nil, the reverse of $newItems$ without the top element)

    and return the top of the reversal of $newItems$.
  - if both lists are nil then raise `EmptyQueue` exception

`FastIntQueue` structure.

```
structure FastIntQueue = struct
  exception EmptyQueue
  type queue = int list ref * int list ref
  fun newQueue () = (ref [],ref []) :  queue
  fun enqueue (i,(a,b)) = a := i ::  (!a)
  fun dequeue (ref [],ref []) = raise EmptyQueue
    | dequeue (a as ref L, b as ref []) =
        let val (h::t) = rev L
        in (a:=nil; b:=t; h)
        end
    | dequeue (a,b as ref (h::t)) = (b := t; h)
end
```

We can achieve better performance by simply changing `IntQueue` to `FastIntQueue`,

In order to make this process easy, structure your code as:

```
local
   structure Q = IntQueue
in
   ...  (* code using Q *)
end
```

If you want to change `IntQueue` to `FastIntQueue`, you simply change the declaration

```
  structure Q = IntQueue
```

to

```
  structure Q = FastIntQueue .
```

Specifying Module Signature

As we have learn, Standard ML system infers a signature for a structure.

However, in many cases, inferred signature contain unnecessary details.

1. Unnecessary functions and values
   For example, in `IntQueue`, `removeLast` is unnecessary to export.
2. Implementation details
   For example, type definition `type queue = int list ref` in `IntQueue` should not be disclosed.

These problems can be solved by specifying a signature for a structure as

`signature` *sigId* `= sig` various specs `end`

In *various specs*, you can write

- type definition
- datatype declaration
- variables and their types
- exception declarations
- structures and their signatures

Example: signature for queue module.

```
signature QUEUE = sig
  exception EmptyQueue
  type queue
  val newQueue : unit -> queue
  val enqueue : int*queue -> unit
  val dequeue : queue -> int
end
```

Two Forms of Signature Specification

(1) Transparent Signature Specification

`structure` *structId* `:` *sigSpec* `=` *module*

Example:

```
structure IntQueue : QUEUE =
   struct
     ···the same as before···
   end;
```

structure IntQueue : QUEUE

By this signature specification, any bindings that are not in signature are hidden. But it exports the type information.

```
- val q = IntQueue.newQueue();
val a = ref [] : queue
- IntQueue.enqueue (1,q);
val it = () : unit
- IntQueue.removeLast q;
stdIn:18.1-18.21 Error: unbound variable or constructor:
removeLast in path FastIntQueue.removeLast
- IntQueue.dequeue q;
val it = 1 : int
- IntQueue.enqueue (2,q);
val it = () : unit
- a := [1];
val it = () : unit
- IntQueue.dequeue q;
val it = 1 : int
```

(2) Opaque Signature Specification

The following specification also hide type structure.

`structure` *structId* `:>` *sigSpec* `=` *module*

The types specified as `type` t in the signature can only be used through the functions declared in the signature.

```
- structure AbsIntQueue :> QUEUE = IntQueue;
structure AbsIntQueue : QUEUE
- val q = AbsIntQueue.newQueue();
val q = - : queue
- AbsIntQueue.enque (1,q);
val it = () : unit
- AbsIntQueue.dequeue q;
val it = 1 : int
```

But you cannot do the following:

```
- q := [1] ;
stdIn:26.1-26.9 Error: operator and operand
don't agree [tycon mismatch]
   operator domain: 'Z ref * 'Z
   operand: IntQueue.queue * int list
   in expression: q := 1 :: nil
```

Selective Disclosure of Types

Sometimes it is necessary to disclose only some of the type structures.
Example : queues whose element types can change.
Attempt 1:

```
signature POLY_QUEUE = sig
  exception EmptyQueue
  type elem
  type queue
  val newQueue : unit -> queue
  val enqueue : elem*queue -> unit
  val dequeue : queue -> elem
end
```

You can then defined a queue for integers as:

```
structure IntQueue :> POLY_QUEUE = struct
  type elem = int
  type queue = elem list ref
    ⋮
end
```

and also a queue for strings as:

```
structure StringQueue :> POLY_QUEUE = struct
  type elem = string
  type queue = elem list ref
    ⋮
end
```

Unfortunately these definitions are useless.

The type `elem` is opaque, the following functions can not be used.

```
val enqueue : elem * queue -> unit
val dequeue : queue -> elem
```

To solve this problem, you can disclose some of the types by using `where type` declaration as:

```
- structure CQueue :> POLY_QUEUE
                      where type elem = char
    =struct
      type elem = char
          ⋮
    end;
```

structure CQueue : POLY_QUEUE?

```
- open CQueue;
```

opening CQueue

$exception\ EmptyQueue$
$type\ elem = char$
$type\ queue$
$val\ newQueue : unit$ `->` $queue$
$val\ enqueue : elem * queue$ `->` $unit$
$val\ dequeue : queue$ `->` $elem$

Type `elem` is now an alias of `char`.

Module Programming Example

Breadth-first search algorithm using a queue:
It can be programed using a queue:

1. Enqueue the root in the queue
2. Do the following until the queue become empty:
   2.1. dequeue an element from the queue
   2.2. if it is not the empty tree then process the node and enqueue the two subtrees.

Define a queue for `string tree`

```
structure STQueue :> POLY_QUEUE
          where type elem = string tree
= struct
     type elem = string tree
     type queue = elem list ref * elem list ref
     ...  (* the same as FastIntQueue *)
  end
```

Define a structure BF that performs breadth-first search:

```
structure BF = struct
   structure Q = STQueue
   fun bf t =
       let val queue = Q.newQueue()
           fun loop () =
               (case Q.dequeue queue of
                     Node(data,l,r) =>
                       (Q.enqueue (l,queue);
                        Q.enqueue (r,queue);
                        data::loop())
                | Empty => loop())
               handle Q.EmptyQueue => nil
       in (Q.enqueue (t,queue); loop())
       end
   end
```

Modular Programming Using Functor

A functor is a function to generate a structure

`functor` *functorId* `(`various spec`)` `=` structure definition

various specs can be any element that can be specified in signature including:

- `val` *id* `:` *type*
- `type` *id*
- `eqtype` *id*
- `datatype`
- `structure` *structId* `:` *sigSpec*
- `exception` *exnId* `of` *type*

Example: parametric queue

```
functor QueueFUN(type elem) :> POLY_QUEUE
                                  where type elem = elem
   = struct
        type elem = elem
        type queue = elem list ref * elem list ref
           ...
           ...  (* same as FastIntQueue *)
           ...
   end
```

Functions can be applied to their arguments

functorId(*various declarations*)

Example:

```
structure ITQuene = QueueFUN(type elem = int tree)
```

Common usage of functions : a function to create a structure using other structures:

```
signature BUFFER = sig
   exception EndOfBuffer
   type channel
   val openBuffer :  unit -> channel
   val input :  channel -> char
   val output :  channel * char -> unit
end
```

```
functor BufferFUN(structure CQueue :  POLY_QUEUE
                            where type elem = char)
        :> BUFFER =
struct
   exception EndOfBuffer
   type channel = CQueue.queue
   fun openBuffer () = CQueue.newQueue()
   fun input ch = CQueue.dequeue ch
   handle CQueue.EmptyQueue => raise EndOfBuffer
   fun output(ch,c) = CQueue.enqueue (c,ch)
end
```

The generated structures can be used just as ordinary structures:

```
- structure CQueue = QueueFUN(type elem = char);
```

structure CQueue

```
- structure Buffer =
          BufferFUN(structure CQueue = CQueue);
```

structure Buffer : BUFFER

SYNTAX OF THE STANDARD ML

Notations

definition	:=	structure1	explanation1
	|	structure2	explanation2
		⋮	
	|	stricture	explanation

⟨optional⟩ is optional

$[E_1, \cdots, E_n]$	one of E_1 through E_n
E*	0 or more E
$E+$	1 or more E

Constant and Identifiers

$$
\begin{array}{lcll}
\mathit{scon} & := & \mathit{int} \mid \mathit{word} \mid \mathit{real} \mid \mathit{string} \mid \mathit{char} & \text{(constant)} \\
\mathit{int} & := & \langle\texttt{~}\rangle[\mathsf{0\text{-}9}]+ & \text{(decimals)} \\
 & \mid & \langle\texttt{~}\rangle\texttt{0x}[\mathsf{0\text{-}9},\texttt{a-f},\texttt{A-F}]+ & \text{hexa decimal notation} \\
\mathit{word} & := & \texttt{0w}[\mathsf{0\text{-}9}]+ & \text{(unsigned decimal)} \\
 & \mid & \texttt{0wx}[\mathsf{0\text{-}9},\texttt{a-f},\texttt{A-F}]+ & \text{(unsigned hexadecimal)} \\
\mathit{real} & := & \mathit{integers}\ \texttt{.}[\mathsf{0\text{-}9}]+\ [\texttt{E},\texttt{e}]\langle\texttt{~}\rangle[\mathsf{0\text{-}9}]+ & \text{(reals)} \\
 & \mid & \mathit{integers}\ \texttt{.}[\mathsf{0\text{-}9}]+ & \\
 & \mid & \mathit{integers}\ [\texttt{E},\texttt{e}]\langle\texttt{~}\rangle[\mathsf{0\text{-}9}]+ & \\
\mathit{char} & := & \texttt{\#"}[\mathit{printable},\mathit{escape}]\texttt{"} & \text{(characters)} \\
\mathit{string} & := & \texttt{"}\ [\mathit{printable},\mathit{escape}]*\ \texttt{"} & \text{(strings)}
\end{array}
$$

printable is the set of printable characters except for \ and ".

$escape$:= `\a` warning (ASCII 7)
| `\b` backspace(ASCII 8)
| `\t` tab(ASCII 9)
| `\n` new line(ASCII 10)
| `\v` vertical tab(ASCII 11)
| `\f` home feed(ASCII 12)
| `\r` return(ASCII 13)
| `\^`C control character C
| `\\` `\` itself
| `\"` `"` itself
| `\`ddd character having the code ddd in decimal
| `\f` $\cdots$ `f\` ignore f$\cdots$f where f is some format character
| `\u`xxx unicode

Classes of Identifiers Used in the Syntax Definition

class	contents	note
vid	variables	long
$tyvar$	type variables startintg with '	
$tycon$	type constructors	long
lab	record labels	
$strid$	structure names	long
$sigid$	signature names	alphanumeric
$funid$	functor names	alphanumeric

Long identifiers

$$longX ::= X \mid strid_1 . \cdots . strid_n . X$$

Syntax of the Core ML

Some auxiliary definitions:

$$
\begin{array}{rcll}
xSeq & ::= & x & \text{one lement} \\
& | & & \text{empty sequence} \\
& | & (x_1, \cdots, x_n) & \text{finite sequence } (n \geq 2)
\end{array}
$$

Syntax for Expressions

$$
\begin{array}{lcl}
\mathit{exp} & ::= & \mathit{infix} \\
 & \mid & \mathit{exp}\ \texttt{:}\ \mathit{ty} \\
 & \mid & \mathit{exp}\ \texttt{andalso}\ \mathit{exp} \\
 & \mid & \mathit{exp}\ \texttt{orelse}\ \mathit{exp} \\
 & \mid & \mathit{exp}\ \texttt{handle}\ \mathit{match} \\
 & \mid & \texttt{raise}\ \mathit{exp} \\
 & \mid & \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{exp}\ \texttt{else}\ \mathit{exp} \\
 & \mid & \texttt{while}\ \mathit{exp}\ \texttt{do}\ \mathit{exp} \\
 & \mid & \texttt{case}\ \mathit{exp}\ \texttt{of}\ \mathit{match} \\
 & \mid & \texttt{fn}\ \mathit{match} \\
\mathit{match} & ::= & \mathit{mrule}\ \langle \texttt{|}\ \mathit{match} \rangle \\
\mathit{mrule} & ::= & \mathit{pat}\ \texttt{=>}\ \mathit{exp}
\end{array}
$$

Function Applications and Infix Expressions

$$
\begin{array}{lcll}
\textit{appexp} & ::= & \textit{atexp} & \\
 & \mid & \textit{appexp}\ \textit{atexp} & \text{(left associative)} \\
\textit{infix} & ::= & \textit{appexp} & \\
 & \mid & \textit{infix}\ \textit{vid}\ \textit{infix} &
\end{array}
$$

Atomic Expressions

$$
\begin{array}{rcl}
atexp & ::= & scon \\
 & \mid & \langle \texttt{op} \rangle\ longvid \\
 & \mid & \{\langle exprow \rangle\ \} \\
 & \mid & \texttt{()} \\
 & \mid & \texttt{(}exp_1\texttt{,}\cdots\texttt{,}exp_n\texttt{)} \\
 & \mid & \texttt{[}exp_1\texttt{,}\cdots\texttt{,}exp_n\texttt{]} \\
 & \mid & \texttt{(}exp_1\texttt{;}\cdots\texttt{;}exp_n\texttt{)} \\
 & \mid & \texttt{let}\ dec\ \texttt{in}\ exp_1\texttt{;}\cdots\texttt{;}exp_n\ \texttt{end} \\
 & \mid & \texttt{(}exp\texttt{)} \\
exprow & ::= & lab\ \texttt{=}\ \ exp\ \langle\texttt{,}\ \ exprow\rangle
\end{array}
$$

Patterns

$$
\begin{array}{lcl}
atpat & ::= & scon \\
 & | & \langle \mathsf{op} \rangle\ longvid \\
 & | & \{ \langle patrow \rangle\ \} \\
 & | & \mathtt{()} \\
 & | & \mathtt{(}pat_1\,\mathtt{,}\cdots\mathtt{,}\,pat_n\mathtt{)} \\
 & | & \mathtt{[}pat_1\,\mathtt{,}\cdots\mathtt{,}\,pat_n\mathtt{]} \\
 & | & \mathtt{(}pat\mathtt{)} \\
patrow & ::= & \ldots \\
 & | & lab\ \mathsf{=}\ \ pat\ \langle \mathtt{,}\ patrow \rangle \\
 & | & \mathsf{vid}\langle \mathtt{:}\,ty \rangle\ \langle \mathtt{as}\ pat \rangle\ \langle \mathtt{,}\,patrow \rangle \\
pat & ::= & atpat \\
 & | & \langle \mathtt{op} \rangle\ longvid\ atpat \\
 & | & pat\ vid\ pat \\
 & | & pat\ \mathtt{:}\ ty \\
 & | & \langle \mathtt{op} \rangle\ pat\ \langle \mathtt{:}\ \ ty \rangle\ \mathtt{as}\ \mathtt{pat}
\end{array}
$$

Types

$$
\begin{array}{lcl}
ty & ::= & tyvar \\
 & | & \{\langle tyrow \rangle\} \\
 & | & tySeq\ longtycon \\
 & | & ty_1\ \mathtt{*} \cdots \mathtt{*}\ ty_n \\
 & | & ty\ \mathtt{->}\ ty \\
 & | & \mathtt{(}ty\ \mathtt{)} \\
tyraw & ::= & lab\ \mathtt{:}\ \langle \mathtt{,}\ tyraw \rangle
\end{array}
$$

Declarations (1)

$$
\begin{array}{lcl}
\mathit{dec} & ::= & \texttt{val}\ \mathit{tyvarSeq}\ \mathit{valbind} \\
 & | & \texttt{fun}\ \mathit{tyvarSeq}\ \mathit{funbind} \\
 & | & \texttt{type}\ \mathit{tybind} \\
 & | & \texttt{datatype}\ \mathit{datbind}\ \langle \texttt{withtyp}\ \mathit{tybind} \rangle \\
 & | & \texttt{datatype}\ \mathit{tycon}\ \texttt{=}\ \mathit{longtycon} \\
 & | & \texttt{exception}\ \mathit{exbind} \\
 & | & \texttt{local}\ \mathit{dec}\ \texttt{in}\ \mathit{dec}\ \texttt{end} \\
 & | & \texttt{opne}\ \mathit{longstrid}_1 \cdots \mathit{longstrid}_n \\
 & | & \mathit{dec}\ \texttt{;}\ \mathit{dec} \\
 & | & \texttt{infix}\ \langle d \rangle\ \mathit{vid}_1 \cdots \mathit{vid}_n \\
 & | & \texttt{infixr}\ \langle d \rangle\ \mathit{vid}_1 \cdots \mathit{vid}_n \\
 & | & \texttt{nonfix}\ \mathit{vid}_1 \cdots \mathit{vid}_n \\
\mathit{valbind} & ::= & \mathit{pat}\ \texttt{=}\ \mathit{exp}\ \langle \texttt{and}\ \mathit{valbind} \rangle \\
 & | & \texttt{rec}\ \mathit{valbind}
\end{array}
$$

Declarations (2)

$$
\begin{array}{lcll}
\mathit{funbind} & ::= & \langle\texttt{op}\rangle\ \mathit{vid}\ \mathit{atpat}_{11} \cdots \mathit{atpat}_{1n}\ \langle\texttt{:}\ \mathit{ty}\rangle\ \texttt{=}\ \mathit{exp}_1 & (m, n \geq 1) \\
 & & \texttt{|}\ \langle\texttt{op}\rangle\ \mathit{vid}\ \mathit{atpat}_{21} \cdots \mathit{atpat}_{2n}\ \langle\texttt{:}\ \mathit{ty}\rangle\ \texttt{=}\ \mathit{exp}_2 & \\
 & & \texttt{|}\ \cdots & \\
 & & \texttt{|}\ \langle\texttt{op}\rangle\ \mathit{vid}\ \mathit{atpat}_{m1} \cdots \mathit{atpat}_{mn}\ \langle\texttt{:}\ \mathit{ty}\rangle\ \texttt{=}\ \mathit{exp}_m & \\
\mathit{tybind} & ::= & \mathit{tyvarSeq}\ \mathit{tycon}\ \texttt{=}\ \mathit{ty}\ \langle\texttt{and}\ \mathit{tybind}\rangle & \\
\mathit{datbind} & ::= & \mathit{tyvarSeq}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind}\ \langle\texttt{and}\ \mathit{datbind}\rangle & \\
\mathit{conbind} & ::= & \langle\texttt{op}\rangle\ \mathit{vid}\ \langle\texttt{of}\ \mathit{ty}\rangle\ \langle\texttt{|}\ \mathit{conbind}\rangle & \\
\mathit{exbind} & ::= & \langle\texttt{op}\rangle\ \mathit{vid}\ \langle\texttt{of}\ \mathit{ty}\rangle\ \langle\texttt{and}\ \mathit{exbind}\rangle & \\
 & | & \langle\texttt{op}\rangle\ \mathit{vid}\ \texttt{=}\ \langle\texttt{op}\rangle\ \mathit{longvid}\ \langle\texttt{and}\ \mathit{exbind}\rangle &
\end{array}
$$

Declarations (3)

$$
\begin{array}{rcl}
\mathit{todec} & ::= & \mathit{strdec}\ \langle \mathit{topdec} \rangle \\
 & | & \mathit{sigdec}\ \langle \mathit{topdec} \rangle \\
 & | & \mathit{fundec}\ \langle \mathit{topdec} \rangle
\end{array}
$$

Structures

$$
\begin{array}{lcl}
\mathit{strdec} & ::= & \mathit{dec} \\
& | & \texttt{structure}\ \mathit{strbind} \\
& | & \texttt{local}\ \mathit{strdec}\ \texttt{in}\ \mathit{strdec}\ \texttt{end} \\
& | & \mathit{strdec}\ \langle \texttt{;} \rangle\ \mathit{strdec} \\
\mathit{strbind} & ::= & \mathit{strid}\ \texttt{=}\ \mathit{strexp}\ \langle \texttt{and}\ \mathit{strbind} \rangle \\
& | & \mathit{strid}\ \texttt{:}\ \mathit{sigexp}\ \texttt{=}\ \mathit{strexp}\ \langle \texttt{and}\ \mathit{strbind} \rangle \\
& | & \mathit{strid}\ \texttt{:>}\ \mathit{sigexp}\ \texttt{=}\ \mathit{strexp}\ \langle \texttt{and}\ \mathit{strbind} \rangle \\
\mathit{strexp} & ::= & \texttt{struct}\ \mathit{strdec}\ \texttt{end} \\
& | & \mathit{longstrid} \\
& | & \mathit{strexp}\ \texttt{:}\mathit{sigexp} \\
& | & \mathit{strexp}\ \texttt{:>}\mathit{sigexp} \\
& | & \mathit{funid}\ \texttt{(}\mathit{strid}\ \texttt{:}\mathit{sigexp}\texttt{)} \\
& | & \mathit{funid}\ \texttt{(}\mathit{strdec}\ \texttt{)}
\end{array}
$$

Signature

$$
\begin{array}{lcl}
\textit{sigdec} & ::= & \texttt{signature}\ \textit{sigbind} \\
\textit{sigexp} & ::= & \texttt{sig}\ \textit{spec}\ \texttt{end} \\
 & \mid & \textit{sigid} \\
 & \mid & \textit{sigid}\ \texttt{where type} \\
 & & \quad \textit{tyvarSeq}\ \textit{longtycon} \texttt{=}\ \textit{ty} \\
\textit{sigbind} & ::= & \textit{sigid}\ \texttt{=}\ \textit{sigexp}\ \langle \texttt{and}\ \textit{sigbind} \rangle
\end{array}
$$

Specifications (1)

$$
\begin{array}{rcl}
\textit{spec} & ::= & \texttt{val}\ \textit{valdesc} \\
 & \mid & \texttt{type}\ \textit{typdesc} \\
 & \mid & \texttt{eqtype}\ \textit{typdesc} \\
 & \mid & \texttt{datatype}\ \textit{datdesc} \\
 & \mid & \texttt{datatype}\ \textit{tycon}\ \texttt{=}\ \texttt{datatype}\ \textit{longtycon} \\
 & \mid & \texttt{exception}\ \textit{exdesc} \\
 & \mid & \texttt{structure}\ \textit{strdesc} \\
 & \mid & \texttt{include}\ \textit{sigexp} \\
 & \mid & \textit{spec}\ \langle\texttt{;}\rangle\ \textit{spec} \\
 & \mid & \textit{spec}\ \texttt{sharing type} \\
 & & \textit{longtycon}_1\ \texttt{=}\ \cdots\ \texttt{=}\textit{longtycon}_n \\
 & \mid &
\end{array}
$$

Specifications (2)

$$
\begin{array}{lcl}
\mathit{valdesc} & ::= & \mathit{vid}\ \texttt{:}\ \mathit{ty}\ \langle \texttt{and}\ \mathit{valdesc} \rangle \\
\mathit{typdesc} & ::= & \mathit{tyvarSeq}\ \mathit{tycon}\ \langle \texttt{and}\ \mathit{typdesc} \rangle \\
\mathit{datdesc} & ::= & \mathit{tyvarSeq}\ \mathit{tycon}\ \texttt{=}\ \mathit{condesc}\ \langle \texttt{and}\ \mathit{datdesc} \rangle \\
\mathit{condesc} & ::= & \mathit{vid}\ \langle \texttt{of}\ \mathit{ty} \rangle\ \langle \texttt{|}\ \mathit{condesc} \rangle \\
\mathit{exdesc} & ::= & \mathit{vid}\ \langle \texttt{of}\ \mathit{ty} \rangle\ \langle \texttt{and}\ \mathit{exdesc} \rangle \\
\mathit{strdesc} & ::= & \mathit{strid}\ \texttt{:}\ \mathit{sigexp}\ \langle \texttt{and}\ \mathit{strdesc} \rangle
\end{array}
$$

Functors

$$
\begin{array}{lcl}
\mathit{fundec} & ::= & \texttt{functor}\ \mathit{funbind} \\
\mathit{funbind} & ::= & \mathit{funid}\ \texttt{(}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \langle \texttt{:}\mathit{sigexp}\rangle\ \texttt{=}\ \mathit{strexp}\ \langle \texttt{and}\ \mathit{funbind}\rangle \\
 & | & \mathit{funid}\ \texttt{(}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\texttt{:>}\mathit{sigexp}\ \texttt{=}\ \mathit{strexp}\ \langle \texttt{and}\ \mathit{funbind}\rangle \\
 & | & \mathit{funid}\ \texttt{(}\ \mathit{spec}\ \texttt{)}\ \langle \texttt{:}\mathit{sigexp}\rangle\ \texttt{=}\ \mathit{strexp}\ \langle \texttt{and}\ \mathit{funbind}\rangle \\
 & | & \mathit{funid}\ \texttt{(}\ \mathit{spec}\ \texttt{)}\ \langle \texttt{:>}\mathit{sigexp}\rangle\ \texttt{=}\ \mathit{strexp}\ \langle \texttt{and}\ \mathit{funbind}\rangle
\end{array}
$$

STANDARD ML BASIS LIBRARY

The Contents of the Library

Mandatory Library

structure名	signature名	remark
Array	ARRAY	polymorphic array
BinIO	BIN_IO	binary IO
BinPrimIO	PRIM_IO	low level IO
Bool	BOOL	
Byte	BYTE	
Char	CHAR	
CharArray	MONO_ARRAY	
CharVector	MONO_VECTOR	
CommandLine	COMMAND_LINE	command line parameters
Date	DATE	
General	GENERAL	

structure name	signature mane	remark
IEEEReal	IEEE_REAL	
Int	INTEGER	
IO	IO	
LargeInt	INTEGER	
LargeReal	REAL	
LargeWord	WORD	
List	LIST	
ListPair	LIST_PAIR	
Math	MATH	
Option	OPTION	
OS	OS	system interface
OS.FileSys	OS_FILE_SYS	
OS.IO	OS_IO	

structure name	signature name	remark
OS.Path	OS_PATH	
OS.Process	OS_PROCESS	
Position	INTEGER	
Real	REAL	
SML90	SML90	for compatibility
String	STRING	
StringCvt	STRING_CVT	string utilities
Substring	SUBSTRING	substrings
TextIO	TEXT_IO	
TextPrimIO	PRIM_IO	
Time	TIME	
Timer	TIMER	
Vector	VECTOR	

structure name	signature name	remark
Vector	VECTOR	
Word	WORD	
Word8	WORD	
Word8Array	MONO_ARRAY	
Word8Vector	MONO_VECTOR	

How to Use a Library

Their signature tell how to use them.

Find out the signature of a library module:

```
- signature X = MATH;
```

signature X =

    sig

        type real

        val pi : real

        val e : real

        val sqrt : real -> *real*

        val sin : real -> *real*

        val cos : real -> *real*

        val tan : real -> *real*

        val asin : real -> *real*

```
    val acos : real -> real
    val atan : real -> real
    val atan2 : real * real -> real
    val exp : real -> real
    val pow : real * real -> real
    val ln : real -> real
    val log10 : real -> real
    val sinh : real -> real
    val cosh : real -> real
    val tanh : real -> real
end
```

It is convenient to print out the signatures of the following basic libraries:

- `BOOL`
- `CHAR`
- `INTEGER`
- `REAL`
- `STRING`
- `LIST`
- `ARRAY`

which serve as “reference cards”.

Boolean `Bool : BOOL`

```
signature BOOL =
  sig
    datatype bool = false | true
    val not : bool -> bool
    val toString : bool -> string
    val fromString : string -> bool option
    val scan : (char,'a) StringCvt.reader -> (bool,'a) StringC
  end
```

Character Char : CHAR

```
signature CHAR =
  sig
    eqtype char
    val chr : int -> char
    val ord : char -> int
    val minChar : char
    val maxChar : char
    val maxOrd : int
    val pred : char -> char
    val succ : char -> char
    val < : char * char -> bool
    val <= : char * char -> bool
    val > : char * char -> bool
    val >= : char * char -> bool
```

```
val compare : char * char -> order
val scan : (char,'a) StringCvt.reader -> (char,'a) StringC
val fromString : string -> char option
val toString : char -> string
val fromCString : string -> char option
val toCString : char -> string
val contains : string -> char -> bool
val notContains : string -> char -> bool
val isLower : char -> bool
val isUpper : char -> bool
val isDigit : char -> bool
val isAlpha : char -> bool
val isHexDigit : char -> bool
val isAlphaNum : char -> bool
val isPrint : char -> bool
```

```
  val isSpace : char -> bool
  val isPunct : char -> bool
  val isGraph : char -> bool
  val isCntrl : char -> bool
  val isAscii : char -> bool
  val toUpper : char -> char
  val toLower : char -> char
end
```

Strings String : STRING

```
signature STRING =
  sig
    type string
    val maxSize : int
    val size : string -> int
    val sub : string * int -> char
    val substring : string * int * int -> string
    val extract : string * int * int option -> string
    val concat : string list -> string
    val ^ : string * string -> string
    val str : char -> string
    val implode : char list -> string
    val explode : string -> char list
    val fromString : string -> string option
```

```
  val toString : string -> string
  val fromCString : string -> string option
  val toCString : string -> string
  val map : (char -> char) -> string -> string
  val translate : (char -> string) -> string -> string
  val tokens : (char -> bool) -> string -> string list
  val fields : (char -> bool) -> string -> string list
  val isPrefix : string -> string -> bool
  val compare : string * string -> order
  val collate : (char * char -> order) -> string * string ->
  val <= : string * string -> bool
  val < : string * string -> bool
  val >= : string * string -> bool
  val > : string * string -> bool
end
```

Integer Modules Int : INTEGER

```
signature INTEGER =
  sig
    eqtype int
    val precision : Int31.int option
    val minInt : int option
    val maxInt : int option
    val toLarge : int -> Int32.int
    val fromLarge : Int32.int -> int
    val toInt : int -> Int31.int
    val fromInt : Int31.int -> int
    val ~ : int -> int
    val * : int * int -> int
    val div : int * int -> int
    val mod : int * int -> int
```

```
val quot : int * int -> int
val rem : int * int -> int
val + : int * int -> int
val - : int * int -> int
val abs : int -> int
val min : int * int -> int
val max : int * int -> int
val sign : int -> Int31.int
val sameSign : int * int -> bool
val > : int * int -> bool
val >= : int * int -> bool
val < : int * int -> bool
val <= : int * int -> bool
val compare : int * int -> order
val toString : int -> string
```

```
  val fromString : string -> int option
  val scan : StringCvt.radix
             -> (char,'a) StringCvt.reader -> (int,'a) Strin
  val fmt : StringCvt.radix -> int -> string
end
```

Reals Real : REAL

```
signature REAL =
  sig
    type real
    structure Math :
      sig
        type real
        val pi : real
        val e : real
        val sqrt : real -> real
        val sin : real -> real
        val cos : real -> real
        val tan : real -> real
        val asin : real -> real
        val acos : real -> real
```

```
    val atan : real -> real
    val atan2 : real * real -> real
    val exp : real -> real
    val pow : real * real -> real
    val ln : real -> real
    val log10 : real -> real
    val sinh : real -> real
    val cosh : real -> real
    val tanh : real -> real
  end
val radix : int
val precision : int
val maxFinite : real
val minPos : real
val minNormalPos : real
```

```
val posInf : real
val negInf : real
val + : real * real -> real
val - : real * real -> real
val * : real * real -> real
val / : real * real -> real
val *+ : real * real * real -> real
val *- : real * real * real -> real
val ~ : real -> real
val abs : real -> real
val min : real * real -> real
val max : real * real -> real
val sign : real -> int
val signBit : real -> bool
val sameSign : real * real -> bool
```

```
val copySign : real * real -> real
val compare : real * real -> order
val compareReal : real * real -> IEEEReal.real_order
val < : real * real -> bool
val <= : real * real -> bool
val > : real * real -> bool
val >= : real * real -> bool
val == : real * real -> bool
val != : real * real -> bool
val ?= : real * real -> bool
val unordered : real * real -> bool
val isFinite : real -> bool
val isNan : real -> bool
val isNormal : real -> bool
val class : real -> IEEEReal.float_class
```

```
val fmt : StringCvt.realfmt -> real -> string
val toString : real -> string
val fromString : string -> real option
val scan : (char,'a) StringCvt.reader
           -> (real,'a) StringCvt.reader
val toManExp : real -> {exp:int, man:real}
val fromManExp : {exp:int, man:real} -> real
val split : real -> {frac:real, whole:real}
val realMod : real -> real
val rem : real * real -> real
val nextAfter : real * real -> real
val checkFloat : real -> real
val floor : real -> int
val ceil : real -> int
val trunc : real -> int
```

```
val round : real -> int
val realFloor : real -> real
val realCeil : real -> real
val realTrunc : real -> real
val toInt : IEEEReal.rounding_mode
            -> real -> int
val toLargeInt : IEEEReal.rounding_mode
                 -> real -> Int32.int
val fromInt : int -> real
val fromLargeInt : Int32.int -> real
val toLarge : real -> Real64.real
val fromLarge : IEEEReal.rounding_mode
                -> Real64.real -> real
val toDecimal : real -> IEEEReal.decimal_approx
val fromDecimal : IEEEReal.decimal_approx -> real
```

```
  sharing type Math.real = real
end
```

Lists List : LIST

```
signature LIST =
  sig
    datatype 'a list = :: of 'a * 'a list | nil
    exception Empty
    val null : 'a list -> bool
    val hd : 'a list -> 'a
    val tl : 'a list -> 'a list
    val last : 'a list -> 'a
    val getItem : 'a list -> ('a * 'a list) option
    val nth : 'a list * int -> 'a
    val take : 'a list * int -> 'a list
    val drop : 'a list * int -> 'a list
    val length : 'a list -> int
    val rev : 'a list -> 'a list
```

```
  val @ : 'a list * 'a list -> 'a list
  val concat : 'a list list -> 'a list
  val revAppend : 'a list * 'a list -> 'a list
  val app : ('a -> unit) -> 'a list -> unit
  val map : ('a -> 'b) -> 'a list -> 'b list
  val mapPartial : ('a -> 'b option) -> 'a list -> 'b list
  val find : ('a -> bool) -> 'a list -> 'a option
  val filter : ('a -> bool) -> 'a list -> 'a list
  val partition : ('a -> bool) -> 'a list -> 'a list * 'a li
  val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
  val foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
  val exists : ('a -> bool) -> 'a list -> bool
  val all : ('a -> bool) -> 'a list -> bool
  val tabulate : int * (int -> 'a) -> 'a list
end
```

General structures

This structure is already opened at the top-level.

```
signature GENERAL =
  sig
    type unit
    type exn
    exception Bind
    exception Chr
    exception Div
    exception Domain
    exception Fail of string
    exception Match
    exception Overflow
    exception Size
    exception Span
```

```
  exception Subscript
  datatype order = EQUAL | GREATER | LESS
  val !  :  'a ref -> 'a
  val := :  'a ref * 'a -> unit
  val o :  ('a -> 'c) * ('b -> 'a) -> 'b -> 'c
  val before :  'a * unit -> 'a
  val ignore :  'a -> unit
  val exnName :  exn -> string
  val exnMessage :  exn -> string
end
```

The Top-Level Environment

Built-in primitive types

type	structure
`eqtype unit`	`General`
`eqtype int`	`Int`
`eqtype word`	`Word`
`type real`	`Real`
`eqtype char`	`Char`
`eqtype string`	`String`
`type substring`	`Substring`
`type exn`	`General`
`eqtype 'a array`	`Array`
`eqtype 'a vector`	`Vector`
`eqtype 'a ref`	

Pre-defined datatypes

type	structure
`datatype bool = false	true`	`Bool`
`datatype 'a option = NONE	SOME of 'a`	`Option`
`datatype order = LESS	EQUAL	GREATER`	`General`
`datatype 'a list = nil	:: of ('a * 'a list)`	`List`

Predefined constants

name	*longvid*
`ref : 'a -> 'a ref`	(built-in primitive)
`! : 'a ref -> 'a`	`General.!`
`:= : 'a ref * 'a -> unit`	`General.:=`
`before : 'a * unit -> 'a`	`General.before`
`ignore : 'a -> unit`	`General.ignore`
`exnName : exn -> string`	`General.exnName`
`exnMessage : exn -> string`	`General.exnMessage`
`o : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b`	`General.o`
`getOpt : ('a option * 'a) -> 'a`	`Option.getOpt`
`isSome : 'a option -> bool`	`Option.isSome`
`valOf : 'a option -> 'a`	`Option.valOf`
`not : bool -> bool`	`Bool.not`

name	defined in
`real : int -> real`	`Real`
`trunc : real -> int`	`Real`
`floor : real -> int`	`Real`
`ceil : real -> int`	`Real`
`round : real -> int`	`Real`
`ord : char -> int`	`Char`
`chr : int -> char`	`Char`
`size : string -> int`	`String`
`str : char -> string`	`String`
`concat : string list -> string`	`String`
`implode : char list -> string`	`String`
`explode : string -> char list`	`String`
`substring : string * int * int -> string`	`String`

name	defined in
`^ : string * string -> string`	`String`
`null : 'a list -> bool`	`List`
`hd : 'a list -> 'a`	`List`
`tl : 'a list -> 'a list`	`List`
`length : 'a list -> int`	`List`
`rev : 'a list -> 'a list`	`List`
`@ : ('a list * 'a list) -> 'a list`	`List`
`app : ('a -> unit) -> 'a list -> unit`	`List`
`map : ('a -> 'b) -> 'a list -> 'b list`	`List`
`foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b`	`List`
`foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b`	`List`
`print : string -> unit`	`TextIO`
`vector : 'a list -> 'a vector`	`Vector`
`use : string -> unit`	(primitive)

Overloaded identifiers

name	default type
`+ : num * num -> num`	`int * int -> int`
`- : num * num -> num`	`int * int -> int`
`* : num * num -> num`	`int * int -> int`
`div : wordint * wordint -> wordint`	`int * int -> int`
`mod : wordint * wordint -> wordint`	`int * int -> int`
`/ : real * real -> real`	`real * real -> real`
`~ : realint -> realint`	`int -> int`
`abs : realint -> realint`	`int -> int`
`< : numtext * numtext -> bool`	`int * int -> bool`
`> : numtext * numtext -> bool`	`int * int -> bool`
`<= : numtext * numtext -> bool`	`int * int -> bool`
`>= : numtext * numtext -> bool`	`int * int -> bool`

$$
\begin{aligned}
\texttt{text} &:= \{\texttt{string}, \texttt{char}\}\\
\texttt{wordint} &:= \{\texttt{word}, \texttt{int}\}\\
\texttt{realint} &:= \{\texttt{real}, \texttt{int}\}\\
\texttt{num} &:= \{\texttt{word}, \texttt{int}, \texttt{real}\}\\
\texttt{numtext} &:= \{\texttt{string}, \texttt{char}, \texttt{word}, \texttt{int}, \texttt{real}\}
\end{aligned}
$$

Binary Operators Declared at the Top-Level

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

USING ARRAYS

Array type : `eqtype 'a array`

τ `array` is a type for arrays over τ. Equality on arrays are pointer equality.

```
signature ARRAY =
 sig type 'a array
     type 'a vector
     val maxLen : int
     val array : int * 'a -> 'a array
     val fromList : 'a list -> 'a array
     val tabulate : int * (int -> 'a) -> 'a array
     val length : 'a array -> int
     val sub : 'a array * int -> 'a
     val update : 'a array * int * 'a -> unit
     val copy : {di:int, dst:'a array, len:int option,
                 si:int, src:'a array} -> unit
```

```
    val copyVec : {di:int, dst:'a array, len:int option,
                   si:int, src:'a vector} -> unit
    val app : ('a -> unit) -> 'a array -> unit
    val foldl : ('a * 'b -> 'b) -> 'b -> 'a array -> 'b
    val foldr : ('a * 'b -> 'b) -> 'b -> 'a array -> 'b
    val modify : ('a -> 'a) -> 'a array -> unit
    val appi : (int * 'a -> unit)
               -> 'a array * int * int option -> unit
    val foldli : (int * 'a * 'b -> 'b)
                 -> 'b -> 'a array * int * int option -> 'b
    val foldri : (int * 'a * 'b -> 'b)
                 -> 'b -> 'a array * int * int option -> 'b
    val modifyi : (int * 'a -> 'a)
                 -> 'a array * int * int option -> unit
end
```

Vectors : immutable arrays.

```
signature VECTOR =
  sig
    eqtype 'a vector
    val maxLen : int
    val fromList : 'a list -> 'a vector
    val tabulate : int * (int -> 'a) -> 'a vector
    val length : 'a vector -> int
    val sub : 'a vector * int -> 'a
    val concat : 'a vector list -> 'a vector
    val app : ('a -> unit) -> 'a vector -> unit
    val map : ('a -> 'b) -> 'a vector -> 'b vector
    val foldl : ('a * 'b -> 'b) -> 'b -> 'a vector -> 'b
    val foldr : ('a * 'b -> 'b) -> 'b -> 'a vector -> 'b
    val appi : (int * 'a -> unit)
```

```
             -> 'a vector * int * int option -> unit
  val mapi : (int * 'a -> 'b)
             -> 'a vector * int * int option -> 'b vector
  val foldli : (int * 'a * 'b -> 'b)
               -> 'b -> 'a vector * int * int option -> 'b
  val foldri : (int * 'a * 'b -> 'b)
               -> 'b -> 'a vector * int * int option -> 'b
end
```

Programming Example: Sorting

Sorting Algorithm

The basis of sorting algorithm : divide and conquer.

The best sorting algorithm: quick sort which proceeds as:

1. Select an element p from a sequence.
2. Divide the sequence into S_1 and S_2 such that $p \geq s$ for any $s \in S_1$ and $p < s$ for any $s \in S_2$.
3. Recursively sort S_1 and S_2.
4. Return the concatenation of the sequences S_1, $[p]$ and S_2.

A Naive Implementation

1. Input data into a list.
2. Sort the list with a function something like:

```
fun sort nil = nil
  | sort (p::t) =
    let fun split nil = (nil,nil)
          | split (h::t) =
            let val (a,b) = split t
            in if h > p then (a,h::b) else (h::a,b)
            end
        val (a,b) = split t
    in
        (sort a)@[p]@(sort b)
    end;
```

This is not you should write for an industrial strength sorting program.

Let's run the sort function

Making a test data:

```
fun randomList n =
    let
       val r = Random.rand (0,1)
       fun next x = Random.randInt r
       fun makeList 0 L = L
         | makeList n L = makeList (n - 1) (next () ::  L)
    in
       makeList n
    end
```

- Random.rand : return a "seed" for random number generator.
- Random.randInt : generate one number from the "seed" and update the seed.

Then you can try your sort function by giving a large list.

```
- val data = makeList 1000000;
```
val data = [...] : int list

```
- sort data;
```
val data = [...] : int list

As you can see, the previous sort consume lots of space!

This should not happen.

You need to use an array for an industrial strength sort system.

Signature of Array Sort

```
signature SORT = sig
  val sort : 'a array * ('a * 'a -> order) -> unit
end
```

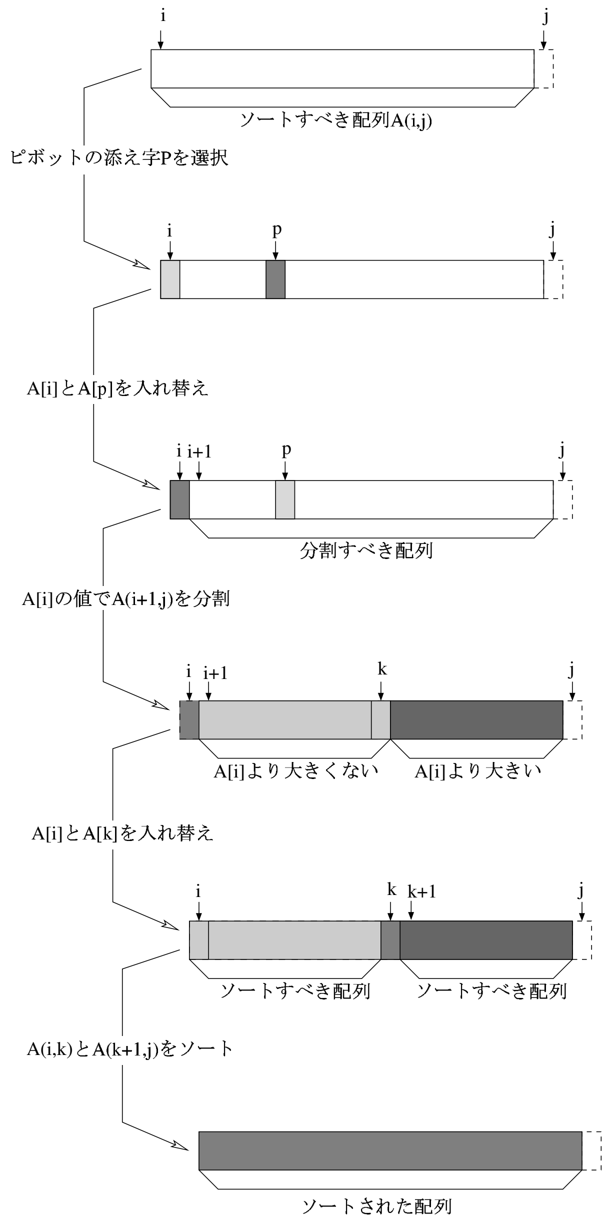
The second parameter is a comparison function. For integers, one can write:

```
fun intcomp (x,y) =
    if x = y then EQUAL
    else if x > y then GREATER
    else LESS
```

Structure of Array Quick Sort

Let $A(i,j)$ be a sub-array from ith element to $j-1$'th element, and $A[i]$ be the i'th element of array A.

1. Let $A(i,j)$ be the sub-array to be sorted.
2. If $j \leq i+1$ then we are done.
3. Select one index p and let $A[p]$ to be the pivot value.
4. Rearrange the array $A(i,j)$ so that there is some k such that elements in $A(i,k)$is less than or equal to $pivot$, $A[k] = A[p]$, and elements in $A(k+1,j)$ are greater than $pivot$.
5. Recursively sort $A(i,k)$ and $A(k+1,j)$.


i
j
ソートすべき配列A(i,j)
ピボットの添え字Pを選択
i
p
j
A[i]とA[p]を入れ替え
i i+1
p
j
分割すべき配列
A[i]の値でA(i+1,j)を分割
i i+1
k
j
A[i]より大きくない
A[i]より大きい
A[i]とA[k]を入れ替え
i
k k+1
j
ソートすべき配列
ソートすべき配列
A(i,k)とA(k+1,j)をソート
ソートされた配列

```
structure ArrayQuickSort : SORT = struct
local open Array
in fun sort (array,comp) =
   let fun qsort (i,j) =
           if j <= i+1 then ()
           else
             let val pivot = ···
                 fun partition (a,b) = ···
                 val k = partition (i+1,j-1)
                 val _ = (* swapping i'th and k'th elements *)
           in (qsort (i,k); qsort (k+1,j))
           end
   in qsort (0,Array.length array)
   end
end
end
```

Pivot selection

1. Select three indexes i_1, i_2 and i_3.
2. Compares $A[i_1],\ \ A[i_2],\ \ A[i_3]$ and select the middle value as the pivot $A[p]$.
3. Swap $A[p]$ and $A[i]$.
4. Partition $A(i+1, j)$ using $A[i]$ as the pivot. Let k be the largest index such that $A[k] \leq pivot$.
5. Exchange $A[i]$ and $A[k]$.

Partitioning a sub-array $A(a, b+1)$

1. Scan from a to the right and find the fist k such that $A[k] > pivot$.
2. Scan from b to the left and find the first l such that $A[l] \leq pivot$.
3. Swap $A[k]$ and $A[l]$
4. Recursively partition $A(k+1, l)$.

```
fun partition (a,b) =
    if b < a then (a - 1)
    else
      let fun scanRight a = ···
          val a = scanRight a
          fun scanLeft b = ···
          val b = scanLeft b
      in if b < a then (a - 1)
         else (swap(a,b);partition (a+1,b-1))
      end
```

Test the Array Sort

Making a test data:

```
fun randomArray n =
    let
       val r = Random.rand (0,1)
       fun next x = Random.randInt r
    in
       Array.tabulate (n,next)
    end
```

Now try the sort function

```
- val a = randomArray 1000000;
```

val a = `[| .... |]` *: int array*

```
ArrayQuickSort.sor(a,intcomp);
```

val it = () : unit

Some Optimization

1. Quicksort is not optimal for small arrays. If the size of the array is 2 or 3, then hand sort the array.
2. If the size of a given array is small (say less than 7), then you should sort it by other simpler sorting method, say insertion sort.
3. Sophisticated pivot selection does not pay off for small arrays. If the given array is not large (say less that 30) then select the first element as the pivot.

USING SYSTEM TIMER

Time and Timer structure

TIME signature

```
signature TIME =
  sig
    eqtype time
    exception Time
    val zeroTime :  time
    val fromReal :  real -> time
    val toReal :  time -> real
    val toSeconds :  time -> LargeInt.int
    val fromSeconds :  LargeInt.int -> time
    val toMilliseconds :  time -> LargeInt.int
    val fromMilliseconds :  LargeInt.int -> time
    val toMicroseconds :  time -> LargeInt.int
    val fromMicroseconds :  LargeInt.int -> time
```

```
  val + :  time * time -> time
  val - :  time * time -> time
  val compare :  time * time -> order
  val < :  time * time -> bool
  val <= :  time * time -> bool
  val > :  time * time -> bool
  val >= :  time * time -> bool
  val now :  unit -> time
  val toString :  time -> string
  val fromString :  string -> time option
end
```

TIMER signature

```
signature TIMER =
  sig
    type cpu_timer
    type real_timer
    val totalCPUTimer :  unit -> cpu_timer
    val startCPUTimer :  unit -> cpu_timer
    val checkCPUTimer :  cpu_timer -> {sys:Time.time,
                                       usr:Time.time,...}
    val totalRealTimer :  unit -> real_timer
    val startRealTimer :  unit -> real_timer
    val checkRealTimer :  real_timer -> Time.time
  end
```

Measure the execution time of a function.

```
fun timeRun f x =
    let
      val timer = Timer.startCPUTimer()
      val _ = f x
      val tm = Timer.checkCPUTimer timer
      val ut = Time.toMicroseconds (#usr tm)
    in LargeInt.toInt ut
    end
```

Then we can measure the time required to sort an array of 1000000 as:

```
timeRun ArrayQuickSort (randomArray(1000000),intcomp);
```

Sort Program Evaluation (1)

The lower bound of the sorting problem is $nlog(n)$.

Check the behavior of the sort function:

```
fun checkTime n =
  let
    val array = genArray n
    val tm = timeRun ArrayQuickSort.sort (array,intcomp)
    val nlognRatio = tm / (nlogn n)
  in
    (n, tm div 1000, nlognRatio)
  end
```

and obtain the time required to sort an array of size n: $c_0 n \log(n)$ (in milliseconds).

Sort Program Evaluation (2)

The previous result depends on the speed of machine etc.

We can factor out the machine etc by doing the following:

use the average time required to compare two element as a unit

The time required to sort an array of size n: $c_1 n \log(n)$ (in the number of comparisons)

Obtain the unit time in your system:

```
fun sample (array,n) =
    let
       val p = Array.sub(array,0)
       val m = n div 2
       fun loop x =
           if x <= m then ()
           else (intcomp(Array.sub(array,n - x),p);
                 intcomp(Array.sub(array,x-1),p);
                 loop (x -1))
    in
       loop n
    end
```

Exercise

1. Write an array sorting module.
2. Write an evaluator function for your sort function which takes a list of numbers for array sizes amd shows the following information:

```
-> evalSort [100000,500000,1000000];
Comparison
 size          time (micro s)   micro s./n
  100000              0         0.00000000
  500000             20         0.04000000
 1000000             30         0.03000000
-----------------------------------------
             avarage            0.02333333
Sorting
 size     time (mil. s)    # of comp / nlogn
```

```
  100000         190          4.90248850
  500000        1140          5.16144689
 1000000        2460          5.28952707
------------------------------------------
             avarage          5.11782082
```

INPUT AND OUTPUT

Stream datatypes : instream and outstream

External data such as files and processes are represented as streams.

Two models of for input streams:

- functional model
- imperative model

The mode of output stream:

- imperative model

A Standard Imperative IO Structure : TextIO

```
signature TEXT_IO =
  sig
    type instream
    type outstream
    val stdIn :  instream
    val stdOut :  outstream
    val stdErr :  outstream
    val openIn :  string -> instream
    val openString :  string -> instream
    val openOut :  string -> outstream
    val openAppend :  string -> outstream
    val closeIn :  instream -> unit
    val closeOut :  outstream -> unit
```

```
  val input :  instream -> string
  val input1 :  instream -> char option
  val inputN : instream * int -> string
  val inputLine :  instream -> string
  val inputAll :  instream -> string
  val canInput :  instream * int -> int option
  val lookahead :  instream -> char option
  val endOfStream :  instream -> bool
  val output :  outstream * string -> unit
  val output1 :  outstream * char -> unit
  val print :  string -> unit
  val outputSubstr :  outstream * substring -> unit
  val flushOut :  outstream -> unit
  ..
end
```

Example

```
open TextIO
fun copyStream ins outs =
    if endOfStream ins then ()
    else case input1 ins of
              SOME c => (output1(outs,c);
                         copyStream ins outs)
            | NONE => copyStream ins outs
fun copyFile inf outf =
    let val ins = openIn inf
        val outs = openOut outf
    in (copyStream ins outs;
        closeIn ins; closeOut outs)
    end
```

```
fun filterStream f ins outs =
    if endOfStream ins then ()
    else case input1 ins of
              SOME c => (output1(outs,f c);
              filterStream f ins outs)
             | NONE => filterStream f ins outs
fun filterFile f inf outf =
    let val ins = openIn inf
        val outs = openOut outf
    in (filterStream f ins outs;
        closeIn ins; closeOut outs)
    end
```

Functional Stream IO

```
signature TEXT_IO =
  sig
    ...
    structure StreamIO :
      sig
        type vector = string
        type elem = char
        type instream
        type outstream
        val input :  instream -> vector * instream
        val input1 :  instream -> (elem * instream) option
        val inputN : instream * int -> vector * instream
        val inputAll :  instream -> vector * instream
```

```
    val canInput : instream * int -> int option
    val closeIn : instream -> unit
    val endOfStream : instream -> bool
    val output : outstream * vector -> unit
    val output1 : outstream * elem -> unit
    val flushOut : outstream -> unit
    val closeOut : outstream -> unit
    val inputLine : instream -> string * instream
    val outputSubstr : outstream * substring -> unit
    ...
  end
val mkInstream : StreamIO.instream -> instream
val getInstream : instream -> StreamIO.instream
val setInstream : instream * StreamIO.instream -> unit
val mkOutstream : StreamIO.outstream -> outstream
```

```
  val getOutstream :  outstream -> StreamIO.outstream
  val setOutstream :  outstream * StreamIO.outstream -> uni
  ...
end
```

An imperative `instream` can be regarded as a reference to functional stream.

For example, the imperative `instream` can be implemented as:

```
type instream = StreamIO.instream ref
fun input1 s = case StreamIO.input1 (!s) of
                    SOME (a,newS) =>(s:=newS; SOME a)
                  | NONE => NONE
```

Functional Stream Example

```
signature ADVANCED_IO =
  sig
    type instream
    type outstream
    val openIn :  string -> instream
    val openOut :  string -> outstream
    val inputN : instream * int -> string
    val lookAheadN : instream * int -> string
    val endOfStream :  instream -> bool
    val canInput :  instream * int -> int option
    val check :  instream -> unit
    val reset :  instream -> unit
    val output :  outstream * string -> unit
```

```
  val redirectIn :  instream * instream -> unit
  val redirectOut :  outstream * outstream -> unit
end
```

```
structure AdvancedIO :> ADVANCED_IO = struct
  structure T = TextIO
  structure S = TextIO.StreamIO
  type instream = T.instream * S.instream ref
  type outstream = T.outstream
  fun openIn f = let val s = T.openIn f
                 in (s,ref (T.getInstream s))
                 end
  fun inputN ((s,_),n) = T.inputN (s,n)
  fun lookAheadN ((s,_),n) = let val ss = T.getInstream s
                             in #1 (S.inputN (ss,n))
                             end
  fun endOfStream(s,_) = T.endOfStream s
  fun canInput ((s,_),n) = T.canInput (s,n)
```

```
  fun check (s,ss) = ss := T.getInstream s
  fun reset (s,ref ss) = T.setInstream (s,ss)
  fun redirectIn ((s1,_),(s2,_)) =
      T.setInstream(s1,T.getInstream s2)
  fun redirectOut (s1,s2) =
      T.setOutstream(s1,T.getOutstream s2)
  val openOut = T.openOut
  val output = T.output
end
```

Programming Example: Lexical Analysis

```
- testLex();
dog
```

ID (dog)

```
(1,2);
```

LPAREN
DIGITS (1)
COMMA
DIGITS (2)
RPAREN
val it = (): unit

Token datatype:

```
datatype token  =
    EOF                                | ID of string
  | DIGITS of string                   | SPECIAL of char
  | BANG            (* ! *)            | DOUBLEQUOTE     (* " *)
  | HASH            (* # *)            | DOLLAR          (* $ *)
  | PERCENT         (* % *)            | AMPERSAND       (* & *)
  | QUOTE           (* ' *)            | LPAREN          (* ( *)
  | RPAREN          (* ) *)            | TILDE           (* ~ *)
  | EQUALSYM        (* = *)            | HYPHEN          (* - *)
  | HAT             (* ^ *)            | UNDERBAR        (* _ *)
  | SLASH           (* \ *)            | BAR             (* | *)
  | AT              (* @ *)            | BACKQUOTE       (* ` *)
  | LBRACKET        (* [ *)            | LBRACE          (* { *)
  | SEMICOLON       (* ; *)            | PLUS            (* + *)
```

```
| COLON          (* : *)   | ASTERISK        (* * *)
| RBRACKET       (* ] *)   | RBRACE          (* } *)
| COMMA          (* , *)   | LANGLE          (* < *)
| PERIOD         (* . *)   | RANGLE          (* > *)
| BACKSLASH      (* / *)  1 | QUESTION        (* ? *)
```

The lexical analysis process:

1. skip preceding space characters
2. determine the kind of token from the first characters
3. read a token

Skipping space characters

```
structure T = TextIO
fun skipSpaces ins =
    case T.lookahead ins of
      SOME c => if Char.isSpace c
                   then (T.input1 ins;skipSpaces ins)
                else ()
    | _ => ()
```

Reading each token:

```
fun getID ins =
    let fun getRest s =
            case T.lookahead ins of
              SOME c => if Char.isAlphaNum c then
                             getRest (s ^  T.inputN(ins,1))
                        else s
            | _ => s
    in ID(getRest "")
    end
```

The lex function:

```
fun lex ins =
    (skipSpaces ins;
     if T.endOfStream ins then EOF
     else let val c = valOf (T.lookahead ins)
          in if Char.isDigit c then getNum ins
             else if Char.isAlpha c then getID ins
             else case valOf (T.input1 ins) of
                    #"!" => BANG
                  | #""" => DOUBLEQUOTE
                  | #"#" => HASH
                  ...  (* other special characters *)
                  | _ => SPECIAL c
          end)
```

A main program:

```
fun testLex () =
    let val token = lex TextIO.stdIn
    in case token of EOF => ()
                   | _ => (print (toString token ^  "\n");
                           testLex ())
    end
```

FORMATING

Substring structure

Intuitively, a substring is a triple (s, i, n)

- s the underlying string
- i start position, and
- n the length of the substring.

Using substring helps making a program faster and more efficient.

```
signature SUBSTRING =
sig type substring
    val base :  substring -> string * int * int
    val string :  substring -> string
    val substring :  string * int * int -> substring
    val extract :  string * int * int option -> substring
    val all :  string -> substring
```

```
val isEmpty : substring -> bool
val getc : substring -> (char * substring) option
val first : substring -> char option
val triml : int -> substring -> substring
val trimr : int -> substring -> substring
val slice : substring * int * int option -> substring
val sub : substring * int -> char
val size : substring -> int
val concat : substring list -> string
val explode : substring -> char list
val isPrefix : string -> substring -> bool
val compare : substring * substring -> order
val splitl : (char -> bool) -> substring
             -> substring * substring
val splitr : (char -> bool) -> substring
```

```
            -> substring * substring
val dropl :  (char -> bool) -> substring -> substring
val dropr :  (char -> bool) -> substring -> substring
val takel :  (char -> bool) -> substring -> substring
val taker :  (char -> bool) -> substring -> substring
val position :  string -> substring -> substring * substr
val span :  substring * substring -> substring
val translate :  (char -> string) -> substring -> string
val fields :  (char -> bool) -> substring -> substring li
val tokens :  (char -> bool) -> substring -> substring li
...  end
```

StringCvt structure

A collection of utility functions for formating.

```
signature STRING_CVT =
sig
  val padLeft :  char -> int -> string -> string
  val padRight :  char -> int -> string -> string
  datatype radix = BIN | DEC | HEX | OCT
  datatype realfmt
    = EXACT
    | FIX of int option
    | GEN of int option
    | SCI of int option
  type ('a,'b) reader = 'b -> ('a * 'b) option
  val splitl :  (char -> bool)
               -> (char,'a) reader -> 'a -> string * 'a
```

```
  val takel :  ( char -> bool)
                 -> (char,'a) reader -> 'a -> string
  val dropl :  (char -> bool) -> (char,'a) reader -> 'a -> 'a
  val skipWS : (char,'a) reader -> 'a -> 'a
  type cs
  val scanString :  ((char,cs) reader -> ('a,cs) reader)
                     -> string -> 'a option
end
```

Reading a data from a string

To read out a data of type τ, you can simply do the following.

1. Use `Substring.getc` as a character reader from `substring`,
2. Define a function

   ```
   scan : (char, substring) reader
          -> (τ,substring) reader.
   ```

3. Apply `scan` to `getc`.

For each atomic type, a `scan` function is already given.

```
Int.scan : StringCvt.radix
           -> (char, 'a) StringCvt.reader
           -> (int, 'a) StringCvt.reader
- fun decScan x = Int.scan StringCvt.DEC x;
```

val decScan = fn : (char,'a) StringCvt.reader
-> (int,'a) StringCvt.reader

This function translates any given character stream to an integer stream.

```
- val intScan = decScan Substring.getc;
```

val intScan = fn : (int,substring) StringCvt.reader

```
val s = Substring.all "123 abc";
```

val s = _ : substring

```
- intScan s;
```

*val it = SOME (123,-) : (int * substring) option*

A Programming Example : URL parser

$$
\begin{array}{rcl}
url & ::= & \texttt{http://}\, domain\, \langle path \rangle\, \langle anchor \rangle \\
 & | & \texttt{file://}\, path \\
 & | & relativePath \\
domain & ::= & id \mid id\,\texttt{.}\,domain \\
path & ::= & \texttt{/}\,relativePath \\
relativePath & ::= & \varepsilon \mid id\,\texttt{/}\,relativePath \\
id & ::= & \text{a string of alpha numeric characters} \\
anchor & ::= & \texttt{\#}\,id
\end{array}
$$

URL parser signature

```
signature PARSE_URL = sig
  exception urlFormat
  datatype url =
        HTTP of {host:string list,
                 path:string list option,
                 anchor:string option}
      | FILE of {path:string list,
                 anchor:string option}
      | RELATIVE of {path:string list,
                     anchor:string option}
  val parseUrl :  string -> url
end
```

```
structure Url:PARSE_URL = struct
  structure SS = Substring
  exception urlFormat
  datatype url = ...
  fun parseHttp s = ...
  fun parseFile s = ...
  fun parseRelative s = ...
  fun parseUrl s =
      let val s = SS.all s
          val (scheme,body) =
              SS.splitl (fn c => c <> #":") s
      in if SS.isEmpty body then
             RELATIVE (parseRelative scheme)
         else case lower (SS.string scheme) of
                   "http" => HTTP (parseHttp body)
```

```
                    | "file" => FILE (parseFile body)
                    | _  => raise urlFormat
          end
    end
```

You can complete the URL parser by writing a conversion function for each given format.

For `http`, you can write a function that performs the following.

1. Check whether the first 3 characters are "`://`".
2. Split the rest into two fields using "`/`" as a delimiter.
3. Decompose the first string into fields using "`.`" as a delimiter, and obtain a list of string representing the host name
4. Decompose the second half into two fields using "#" as a delimiter.
5. Decompose the first half into fields using "`/`" as a delimiter, and obtain a list of string representing a file path.
6. The second half is an anchor string.

```
fun parseHttp s =
 let val s = if SS.isPrefix "://" s then
                SS.triml 3 s
             else raise urlFormat
     fun neq c x = not (x = c)
     fun eq c x = c = x
     val (host,body) = SS.splitl (neq #"/") s
     val domain = map SS.string (SS.tokens (eq #".") host)
     val (path,anchor) =
          if SS.isEmpty body then (NONE,NONE)
          else
            let val (p,a) = SS.splitl (neq #"#") body
            in (SOME (map SS.string
                           (SS.tokens (eq #"/") p)),
                if SS.isEmpty a then NONE
```

```
              else SOME (SS.string (SS.triml 1 a)))
          end
in {host=domain, path=path, anchor=anchor}
end
```

Example : Formated Output

Let us write a general purpose print function corresponding to `printf` in C.

The first task is to design a datatype for format specification including the following information:

- datatype and its representing information
  - integers
    radix (binary, decimal, hexadecimal, octal)
  - reals
    printing format (exact, fixed, general, scientific)
- data alignment
- the data length

We define the following format specification.

```
datatype kind =  INT of StringCvt.radix
               | REAL of StringCvt.realfmt
               | STRING
               | BOOL
datatype align = LEFT | RIGHT
type formatSpec = {kind:kind,
                   width:int option,
                   align:align}
```

The next task is to write a function that takes a format specification and a corresponding data and convert the data into a string.

For this purpose, we need to treat data of different types as data of the same type.

```
datatype argument =
         I of int
       | R of real
       | S of string
       | B of bool
```

We can now write a conversion function as:

```
exception formatError
fun formatData {kind,width,align} data=
 let val body =
          case (kind,data) of
          (INT radix,I i) => Int.fmt radix i
        | (REAL fmt,R r) => Real.fmt fmt r
        | (STRING,S s) => s
        | (BOOL,B b) => Bool.toString b
        | _ => raise formatError
 in case width of
        NONE => body
      | SOME w => (case align of
                      LEFT => StringCvt.padRight
                                #" " w body
```

```
                          | RIGHT => StringCvt.padLeft
                                       #" " w body)
end
```

We also need to write a function for parsing a format specification string.

We consider the following format string:

$$\%[-][ddd]type$$

- $-$ is for left align. The default is right align.
- ddd is the field length. If the given data is shorter than the specified length, then the formatter pads white spaces.
- $type$ is one of the following

`d`	`int` in decimal notation
`x`	`int` in hexadecimal notation
`o`	`int` in octal notation
`f`	`real` in fixed decimal point notation
`e`	`real` in scientific notation
`g`	`real` in the system default representation.

To represent a string having these embedded formating strings, we define the following datatype.

```
datatype format = SPEC of formatSpec
                | LITERAL of string
```

`LITERAL` is a data string to be printed.

A function to parse a format string.

```
fun parse s =
  let val (s1,s) = StringCvt.splitl
                  (fn c=>c <> #"%") SS.getc s
      val prefix = if s1 = "" then nil
                   else [LITERAL s1]
  in if SS.isEmpty s then prefix
     else let val (f,s) = oneFormat s
              val L = parse s
          in prefix@(f::L)
          end
  end
```

A function to convert a formating string into `formatSpec`.

```
fun oneFormat s =
  let val s = SS.triml 1 s
  in if SS.isPrefix "%" s then
        (LITERAL "%",SS.triml 1 s)
     else
      let val (a,s) = if SS.isPrefix "-" s
                        then (LEFT,SS.triml 1 s)
                      else (RIGHT,s)
          val (w,s) = scanInt s
          val (c,s) = case SS.getc s of
                         NONE => raise formatError
                       | SOME s  => s
      in (SPEC {width=w,align=a,
                kind = case c
```

```
                    of #"d" => INT StringCvt.DEC
                     | #"s" => STRING
                     | #"f" => REAL (StringCvt.FIX NONE)
                     | #"e" => REAL (StringCvt.SCI NONE)
                     | #"g" => REAL (StringCvt.GEN NONE)
                     | _ => raise formatError},
             s)
         end
     end
```

The formating function.

```
fun format s L =
    let val FL = parse (SS.all s)
        fun traverse (h::t) L =
            (case h of
                LITERAL s => s ^ (traverse t L)
              | SPEC fmt =>
                   (formatData fmt (List.hd L)
                    ^ (traverse t (List.tl L))))
          | traverse nil l = ""
    in (traverse FL L)
    end
```

A formating module.

```
signature FORMAT =
sig datatype kind =  INT of StringCvt.radix
                   | REAL of StringCvt.realfmt
                   | STRING
                   | BOOL
    datatype align = LEFT | RIGHT
    datatype format =
             LITERAL of string
           | SPEC of {kind:kind,
                      width:int option,
                      align:align}
    datatype argument = I of int
                      | R of real
                      | S of string
```

```
                    | B of bool
    exception formatError
    val format : string -> argument list -> string
    val printf : string -> argument list -> unit
end
```

Exercise

1. Design and implement a balanced binary search tree.
   - Try to implement it as a functor that takes a key type and a comparison function and returns a module.
   - The to implement a polymorphic binary search tree so that it can used for various different values for each given key type.
   - Try to support various useful utility functions such as:
     - a function to create a tree form a list
     - functions to list keys and items, to `foldr` a tree, to `map` a function over a tree, etc.
2. Evaluate your search module by creating trees of various sizes and measuring execution speed of various functions.
3. **Extra Credit.** Add a function to delete a node having a given key.

For Further Study

- MLプログラミング
  L. Paulson, "ML for the Working Programmer, 2nd Edition", Cambridge University Press, 1996.

- ML等の近代的言語動作原理
  大堀，西村，ガリグ，コンピュータサイエンス入門 アルゴリズムとプログラミング言語，岩波書店

- ML等の近代的言語の実装
  X. Leroy, The ZINC experiment : an economical implementation of the ML language,

お願い

MLの教科書（Standard ML入門）やその他MLに関する感想（MLにほしい機能，読んでみたいMLの教材など)があったらohoriまでメールください．

教科書の改善や，新しい教材作成，さらに，開発中の次世代MLなどの参考にさせて頂きます．