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 near future) Major Dialects of ML

1. Standard ML
   An “official” one described in the following book.
   
   R. Milner, M. Tofte, R. Harper, and D. MacQueen
   
   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”,
  (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:

\[
\begin{align*}
0! &= 1 \\
n! &= n \times !(n - 1)
\end{align*}
\]

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

```c
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

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

Type Error: circularity

```lisp
(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
% 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;
  val it = true : bool
- false;
  val it = false : bool
- "A";
  val it = "A" : char

where

- \texttt{int} : integers
- \texttt{real} : floating point numbers
- \texttt{bool} : boolean values
- \texttt{char} : characters
Expressions with Primitive Operations

Simple arithmetic expressions

- $\sim 2$
  
  ```
  val it = \sim 2 : int
  ```

- $3 + \sim 2$
  
  ```
  val it = 1 : int
  ```

- $22 - (23 \mod 3)$
  
  ```
  val it = 20 : int
  ```

- $3.14 \times 6.0 \times 6.0 \times (60.0/360.0)$
  
  ```
  val it = 18.84 : real
  ```

- $\sim n$ is a negative number, and $\sim$ is the operator to multiply $\sim 1$.

- $\text{mod}$ is the remainder operator.
Expression can be of any length.

\[- 2 \times 4 \div (5 - 3)\]
\[= \ast 3 + (10 - 7) + 6;\]

\textit{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:

- \((7 \mod 2) = 2\);
  
  ```ml
  val it = false : bool
  ```

- \((7 \mod 2) = (if \ false \ then \ 1 \ else \ 2)\);
  
  ```ml
  val it = false : bool
  ```

- \(if \ (7 \mod 2) = 0 \ then \ 7 \ else \ 7 - 1\);
  
  ```ml
  val it = 60 : int
  ```

- \(- \ 6 \ * \ 10\);
  
  ```ml
  val it = 60 : int
  ```

- \(if \ (7 \mod 2) = 0 \ then \ 7 \ else \ 7 - 1) \ * \ 10\);
  
  ```ml
  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:

```markdown
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 type checking, 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:

- \( \text{real 10;} \)
  - 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.
   1. 44;
   2. it mod 3;
   3. 44 - it;
   4. (it mod 3) = 0;
   5. if it then "Boring" else "Strange";

Check your prediction by evaluating them in SML/NJ.

3. In the ASCII code, uppercase letters A through 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:

\[ \text{val } \text{name} = \text{exp} ; \]

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 \texttt{fun} construct:

\begin{verbatim}
fun \textit{f} \textit{p} = \textit{body} ;
\end{verbatim}

where

\begin{itemize}
\item \textit{f} is the name of the function,
\item \textit{p} is a formal parameter, and
\item \textit{body} is the function body.
\end{itemize}

This declaration define a function that takes a parameter named \textit{p} and return the value \textit{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:

\[ f \text{ of type } \tau_1 \rightarrow \tau_2 \text{ can be applied to an expression } E \text{ of type } \tau_1, \]
\[ \text{yielding a value of type } \tau_2. \]

This can be written concisely as:

\[
\frac{f : \tau_1 \rightarrow \tau_2 \quad E : \tau_1}{f \ E : \tau_2}
\]

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 \( \tau_1 \) is allowed.
- double (if true then 2 else 3);
  \texttt{val it = 4 : int}
- double 3 + 1;
  \texttt{val it = 7 : int}
- double (double 2) + 3;
  \texttt{val it = 11 : int}

Important note:

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

So, \texttt{double 3 + 1} is interpreted as (\texttt{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,
- \(\tau_1 \cdot \cdots \cdot \tau_n\) is a tuple type,
- Function type constructor \(\rightarrow\) associate weakly than other type constructor, so \(\text{int} \cdot \text{int} \rightarrow \text{int}\) is \((\text{int} \cdot \text{int}) \rightarrow \text{int}\).
Typing rule for tuples is:

\[
\frac{E_1 : \tau_1 \quad \cdots \quad E_n : \tau_n}{(E_1, \ldots, E_n) : \tau_1 \ast \cdots \ast \tau_n}
\]

So you can form any form of tuples:

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

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

: string * (string * string) * int
Evaluation Process of Function Application

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

1. Evaluate $E_1$ to obtain a function definition $\text{fn} \ x \Rightarrow E_0$
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:

1. $\text{Eval}([], \text{double (double 2) + 3})$
2. $\text{Eval}([], \text{double (double 2)})$
3. $\text{Eval}([], \text{(double 2)})$
4. $\text{Eval}([], 2) = 2$
5. $\text{Eval}([x \mapsto 2], x * 2)$
6. $\text{Eval}([x \mapsto 2], x) = 2$
7. $\text{Eval}([x \mapsto 2], 2) = 2$
8. $\text{Eval}([x \mapsto 2], 2 * 2) = 4$
9. $= 4$
10. $\text{Eval}([x \mapsto 4], x * 2)$
11. $\text{Eval}([x \mapsto 4], x) = 4$
12. $\text{Eval}([x \mapsto 4], 2) = 2$
13. $\text{Eval}([x \mapsto 4], 4 * 2) = 8$
14. $= 8$
15. $\text{Eval}([], 3) = 3$
16. $\text{Eval}([], 8 + 3) = 11$
17. $= 11$
3. RECURSIVE AND HIGHER-ORDER FUNCTIONS
Recursive Functions

A function definition

\[ \text{fun } f \ p = \text{body} \]

is recursive if \( \text{body} \) contains \( f \), the function being defined by this definition.

A simple example: the factorial function

\[
0! = 1 \\
n! = n \times (n - 1)!
\]

\[
\text{fun } f \ n = \\
\quad \text{if } n = 0 \text{ then } 1 \\
\quad \text{else } n \times f(n - 1)
\]

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.
   \[ \text{if } n = 0 \text{ then } 1 \]

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.
   \[ n \times f (n - 1) \]
How to analyze a recursive function definition:

```haskell
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:

\[ F_0 = 1 \]
\[ F_1 = 1 \]
\[ F_n = F_{n-2} + F_{n-1} \quad (n \geq 2) \]

The following function computes this sequence.

```haskell
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:

\[
\text{fun } f \ n = \\
\quad \text{if } n = 0 \text{ then } 1 \\
\quad \text{else } n \ast f (n - 1)
\]

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 \( \text{fun } f \ p = \text{body} \) is tail recursive if all the calls of \( f \) in \( \text{body} \) are in tail calls positions in \( \text{body} \).

\( [\ ] \) is a tail call position in \( T \):

\[
T := [\ ] \mid \text{if } E \text{ then } T \text{ else } T \mid (E; \cdots; T) \\
\mid \text{let } \text{decs} \text{ in } T \text{ end} \\
\mid \text{case } e \text{ of } p_1 \Rightarrow T \mid \cdots \mid p_n \Rightarrow T
\]

Examples of tail calls:

- \( f \ e \)
- \( \text{if } e_1 \text{ then } f \ e_2 \text{ else } f \ e_3 \)
- \( \text{let } \text{decls} \text{ in } f \ e \text{ end} \)

Tail recursive functions do not consume stack space and therefore more efficient.
Tail recursive version of \texttt{fact}:

\begin{verbatim}
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
\end{verbatim}

\texttt{fact} is an auxiliary function for \texttt{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:

```plaintext
- 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 \textbf{let } \textit{decl} \textbf{in } \textit{exp} \textbf{end}:

\begin{itemize}
\item This is an expression.
\item The type and the result of this expression is those of \textit{exp}.
\end{itemize}

\begin{verbatim}
- let
  val pi = 3.141592
  fun f r = 2.0 * pi * r
in
  f 10.0
end * 10.0;
val it = 628.3184 : real
\end{verbatim}
Local Definitions

The following local special form also introduce local definitions

```
local
dclList1
in
dclList2
end
```

- Only the definitions in `dclList2` are valid after this declaration.
- `dclList1` is used in `dclList2`. 
- 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

- $A^n_x$ be the amount after $n$ year deposit
- $I^n_x$ the $n'th$ year’s interest rate.
They satisfy the following equations:

\[ I^n_x = F(A^{n-1}_x) \quad (n \geq 1) \]
\[ A^n_x = \begin{cases} 
    x & (n = 0) \\
    A^{n-1}_x \times (1.0 + I^n_x) & (n \geq 1)
\end{cases} \]

which can be directly programmed 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:

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

In computing \( \text{fib } n \),

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

::

Question: How may times \( \text{fib } 0 \) 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:

$$G_0(a, b) = a$$
$$G_1(a, b) = b$$
$$G_{k+1}(a, b) = G_k(b, a+b)$$
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

$$\text{fun } I(x,n) = F(A(x,n - 1))$$
and $$A(x,n) = \text{if } n = 0 \text{ then } x$$
$$\quad \text{else } A(x,n - 1)*(1.0+I(x,n));$$

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
- $\vdots$
Repeated computation is avoided by computing $A^m_x$ and $I^m_x$ using a function $G$ defined as

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

which satisfies:

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

So we can define:

```plaintext
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 \rightarrow int
- Power(3,2);
val it = 8

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 \rightarrow int \rightarrow 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:

```haskell
- 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.

```ml
- 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:

- \((\text{int} \to \text{int}) \to \text{int} \to \text{int}\) is \((\text{int} \to \text{int}) \to (\text{int} \to \text{int})\).

- **summation** is a function that
  - takes a value of type \(\text{int} \to \text{int}\), and
  - return a value of type \(\text{int} \to \text{int}\).

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 \texttt{summation} as far as the usage is type consistent.

- \texttt{val newSumOfCube = summation cube;}
  \texttt{val newSumOfCube = fn : int \rightarrow int}
- \texttt{newSumOfCube 3; val it = 36 : int}

We can also write:

- \texttt{val sumOfSquare = summation (power 2);}
  \texttt{val sumOfSquare = fn : int \rightarrow int}
- \texttt{sumOfSquare 3; val it = 14}
- \texttt{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

\[ \text{fn } p \Rightarrow \text{body} \]

denotes the function that takes \( p \) and returns the value denoted by \( \text{body} \).

- \( \text{fn } x \Rightarrow x + 1; \)
- \( \text{val it = fn : int -> int} \)
- \( (\text{fn } x \Rightarrow x + 1) \ 3 \ * \ 10; \)
- \( \text{val it = 40 : int} \)
Using this function expression, we can write a program to generate a function.

To see its usefulness, consider:

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

which computes \( \text{fib } n \) for each different \( m \). So

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

compute \( \text{fib } 35 \) 5 times.

This is clearly redundant, and should be avoided.
We can factor out the computation \texttt{fib 35} as

\begin{verbatim}
fun f n = let val a = (fib n)
    in fn m => a mod m = 0
    end
\end{verbatim}

This performs the following computation:
1. receives \texttt{n}
2. computes \texttt{fib n} and binds \texttt{m} to the result,
3. return a function \texttt{fn m => a mod m = 0}.

So after this,

\begin{verbatim}
val g = f 35;
(g 1,g 2,g 3,g 4,g 5);
\end{verbatim}

will not compute \texttt{fib 35} again.
Static Scoping

General principle: defining a name hide the previous definition.

A simple example:

```ml
- 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

val $x_1 = \text{exp}_1$
and $x_2 = \text{exp}_2$
:  
and $x_n = \text{exp}_n$ ;

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

\[
\text{fun } f_1 \ p_1^1 \ \cdots \ p_{k_1}^1 = \ exp_1 \\
\text{and } f_2 \ p_1^2 \ \cdots \ p_{k_2}^2 = \ exp_2 \\
\vdots \\
\text{and } f_n \ p_1^n \ \cdots \ p_{k_n}^n = \ exp_n \\
\]

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 \( \rightarrow \) 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 argument.

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 own operators as:

- `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 \(2 \times 2\) matrix \[
\begin{pmatrix}
    a & b \\
    c & d
\end{pmatrix}
\] as \((a, b, c, d)\). Write a function \texttt{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:
   
   \[
   F_n = F_n \\
   F_{n+1} = F_{n-1} + F_n
   \]
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

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

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 $\sum_{k=1}^{n} f(k)$ 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 \texttt{integral} that takes \( f, n, a \) and \( b \), and compute the above value.
You will need a function \texttt{real} : \texttt{int} -> \texttt{real} that converts a value of type \texttt{int} to the corresponding value of \texttt{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, \( \sum_{k=1}^{n} f(k) \) 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,

   \[
   \text{fn x => x = 1 orelse x = 2 orelse x = 3}
   \]

   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.

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

which is equivalent to

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

But ML infers its type!

```ml
val products = fn : (int -> int) -> int -> int
```
This achieves complete static type checking without type annotation:

```haskell
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:

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

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

```haskell
factorial
  :
  (⋯ (factorial 4) ⋯)
Worng type argument: integer-or-marker-p (lambda (x) x)
```
Examples of type errors:

• $3 \times 3.14$

• `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.

```plaintext
fun id x = x
```

`id` has infinitely many types such as

```plaintext
int -> int
```
or

```plaintext
string -> string.
```
or any other types of the form $\tau \rightarrow \tau$.

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

```plaintext
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 = "MLMLMLMLML" : 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
   
   $$\text{fun } \text{twice}_{\tau_1} f_{\tau_2} x_{\tau_3} = f (f x)_{\tau_4 \tau_5};$$

2. Since `twice` is a function that takes \( f \) and \( x \), we must have:
   
   $$\tau_1 = \tau_2 \rightarrow \tau_3 \rightarrow \tau_5$$

3. For \( (f x)_{\tau_4} \) to be type correct, we must have;
   
   (1) \( f \)'s type \( \tau_2 \) must be a function type of the form \( \alpha \rightarrow \beta \)
   
   (2) \( \alpha \) must be equal to \( x \)'s type \( \tau_3 \),
   
   (3) \( \beta \) must be equal to the result type \( \tau_4 \) of \( (f \ x) \).

So we have the following equation:

$$\tau_2 = \tau_3 \rightarrow \tau_4$$
4. Similarly, for $f \ (f \ x)_{\tau_4 \tau_5}$ to be type correct, we must have the equation:

$$\tau_2 = \tau_4 \rightarrow \tau_5$$

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

$$\tau_1 = \tau_2 \rightarrow \tau_3 \rightarrow \tau_5$$

$$\tau_2 = \tau_3 \rightarrow \tau_4$$

$$\tau_2 = \tau_4 \rightarrow \tau_5$$

One such solution is

$$\tau_1 = (\tau_3 \rightarrow \tau_3) \rightarrow \tau_3 \rightarrow \tau_3$$
Explicit Type Declaration
You can specify types as in:

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

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

or

```haskell
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

```haskell
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.

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

If there is no context to disambiguate, then the default is chosen.
Some of overloaded identifiers:

<table>
<thead>
<tr>
<th>id</th>
<th>type scheme</th>
<th>possible t</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>div</td>
<td>$t \times t \rightarrow t$</td>
<td>word, int</td>
<td>int</td>
</tr>
<tr>
<td>mod</td>
<td>$t \times t \rightarrow t$</td>
<td>word, int</td>
<td>int</td>
</tr>
<tr>
<td>+,-</td>
<td>$t \times t \rightarrow t$</td>
<td>word, real, int</td>
<td>int</td>
</tr>
<tr>
<td>$&lt;,&gt;,&lt;=,&gt;=$</td>
<td>$t \times t \rightarrow \text{bool}$</td>
<td>word, real, int, string, char</td>
<td>int</td>
</tr>
</tbody>
</table>
**Value Polymorphism**

Currently, polymorphism is restricted to values:

\[ V ::= c \mid x \mid (V, \ldots, V) \mid \text{fn } p \Rightarrow e \]

So

\[(\text{fn } x \Rightarrow x, \text{fn } y \Rightarrow y)\];

\[\text{it} = \text{poly-record} : (\char Musical note \mathbf{a} \to \char Musical note \mathbf{a}) \times (\char Musical note \mathbf{b} \to \char Musical note \mathbf{b})\]

is OK, but

- \text{fun } f \ n \ x = (n, x);
- \text{val } f = \text{fn} : \char Musical note \mathbf{a} \to \char Musical note \mathbf{b} \to \char Musical note \mathbf{a} \times \char Musical note \mathbf{b}
- \ f \ 1;

\text{stdIn:21.1-21.4 Warning: type vars not generalized because of value restriction are instantiated to dummy types } (X1,X2,\ldots)\]

\[\text{val } \text{it} = \text{fn} : ?.X1 \to \text{int} \times ?.X1\]
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

δ ::= int | bool | char | string | δ * · · · * δ | ref τ

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.

```haskell
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 $\tau$ be a type not containing any type variable, and suppose $E$ is an expression of type $\tau$.

(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 $\tau$.

(2) Define a function that behaves as the identity function and its most general type is $\tau \rightarrow \tau$. 
5. Functions can represent any data structures, including integers. Let us do some arithmetic using functions. Consider the `twice` again:

```plaintext
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

```ml
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

```ml
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:

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

For example:

```ocaml
- 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:

\[ \text{exp}_1 \text{ andalso } \text{exp}_2 \] (conjunction)
\[ \text{exp}_1 \text{ orelse } \text{exp}_2 \] (disjunction)
\[ \text{if } \text{exp} \text{ then } \text{exp}_1 \text{ else } \text{exp}_1 \] (conditional)

These special forms suppress unnecessary evaluation.

- \( \text{exp}_1 \text{ andalso } \text{exp}_2 \) evaluates \( \text{exp}_2 \) only if \( \text{exp}_1 \) is true.
- \( \text{exp}_1 \text{ orelse } \text{exp}_2 \) evaluates \( \text{exp}_2 \) only if \( \text{exp}_1 \) is false.
- \( \text{if } \text{exp}_1 \text{ then } \text{exp}_2 \text{ else } \text{exp}_3 \) evaluate only one of \( \text{exp}_2 \) or \( \text{exp}_3 \) 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 0xnnnn where $n$ is digit (0 9) or letters from a (or A ) to f (or F).

Negative literals are written as $\sim 123$:

- 123;
- `val it = 123 : int`
- $\sim 0x10$;
- `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. \(xE^n\) 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 warning (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 carriage return (ASCII 13)
\^c control character c
\" character "
\\ character \n\\ the character whose code is ddd in decimal
Operations on characters:

```ml
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 $(\text{start}_1, \text{start}_2, l)$.

$\text{start}_1$ and $\text{start}_2$ are the starting positions of $s_1$ and $s_2$

We search all the possible pairs $(\text{start}_1, \text{start}_2)$ and find a maximal $l$.

Search strategy:

Maintaining the maximum substring $(\text{start}_1, \text{start}_2, \text{max})$ 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
    end
  in findMatch (0,0) (0,0,0) end
RECORDS
Record Types and Record Expressions

Syntax of record types

\[ \{ l_1 : \tau_1, \ldots, l_n : \tau_n \} \]

where

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

Example

\[
\text{type malt = \{Brand:string, Distiller:string, Region:string, Age:int\}}
\]
Syntax of record expressions

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

where \(exp_i\) can be any expression.

Typing rule for records:

\[
\begin{array}{c}
\text{exp}_1 : \tau_1 \quad \cdots \quad \text{exp}_n : \tau_n \\
\end{array}
\]

\[
\{l_1 = \text{exp}_1, \cdots, l_n = \text{exp}_n\} : \{l_1 : \tau_1, \cdots, l_n : \tau_n\}
\]

\[\text{- val myMalt} = \{\text{Brand} = "Glen Moray", \]
\[\text{Distiller} = "Glenlivet", \]
\[\text{Region} = "the Highlands", \text{Age} = 28\};\]

\[\text{val myMalt} = \{\text{Age} = 28, \text{Brand} = "Glen Moray", \]
\[\text{Distiller} = "Glenlivet", \]
\[\text{Region} = "the Highlands" \} \]
\[: \{\text{Age} : \text{int}, \text{Brand} : \text{string}, \text{Distiller} : \text{string}, \text{Region} : \text{string}\}\]
According to our principle:

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

records are freely combined with any other constructs:

```ocaml
- 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 patterns of the form:

\[
\{ l_1 = \text{pat}_1, \ldots, l_n = \text{pat}_n \} \\
\{ l_1 = \text{pat}_1, \ldots, l_n = \text{pat}_n, \ldots \}
\]

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

<table>
<thead>
<tr>
<th>Tuple Notations</th>
<th>The Corresponding Record Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(exp_1, exp_2, ⋮, exp_n)</code></td>
<td><code>{1=exp_1, 2=exp_2, ⋮, n=exp_n}</code></td>
</tr>
<tr>
<td><code>τ_1 * τ_2 * ⋮ * τ_n</code></td>
<td><code>{1:τ_1, 2:τ_2, ⋮, n:τ_n}</code></td>
</tr>
<tr>
<td><code>(pat_1, pat_2, ⋮, pat_n)</code></td>
<td><code>{1=pat_1, 2=pat_2, ⋮, n=pat_n}</code></td>
</tr>
</tbody>
</table>
Indeed you can write:

- `#2 p;`
  
\[
\text{val it = 3 : int}
\]

- `val \{1=x,2=y\} = p;`
  
\[
\text{val x = 2}
\]
\[
\text{val y = 3}
\]

- `f \{1=1,2=2\};`
  
\[
\text{val it = 3 : int}
\]
PROGRAMMING WITH LISTS
List Structure

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

\[ L : v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \ldots \rightarrow v_n \rightarrow \text{nil} \]

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

$$(v_1, (v_2, \cdots (v_n, \text{nil}) \cdots))$$

Let $A$ be the set from which elements $v_1, v_2, \ldots, v_n$ are taken. Also let $\text{Nil}$ to be the set $\{\text{nil}\}$. 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:

$$A \times (A \times (\cdots (A \times \text{Nil}) \cdots))$$

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

$$L = \text{Nil} \cup (A \times L)$$
We can obtain the solution in the following steps:

1. Define a sequence of sets $X_i$ as follows:

$$X_0 = \text{Nil}$$

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

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: $\tau$ list

$\tau$ list is a type of lists of element of type $\tau$ where $\tau$ can be any type. For example:

- $\text{int list}$: integer lists.
- $\text{int list list}$: lists of integer lists.
- $(\text{int -> int})$ list: lists of integer functions.

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

$$v_1 :: v_2 :: \cdots :: v_n :: \text{nil}$$

The following shorthand (syntactic sugar) is also supported.

$$[] \implies \text{nil}$$

$$[exp_1, exp_2, \cdots, exp_n] \implies exp_1 :: exp_2 :: \cdots :: exp_n :: \text{nil}$$
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 \([1,2,3,4,\ldots,n]\) for a given \(n\).

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 list
  - 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 Patterns

The basic operation on a list is pattern matching:

\[
\text{case } E_0 \text{ of } \text{nil} \Rightarrow E_1 \\
\quad \mid (h::t) \Rightarrow E_2(h,t)
\]

which performs the following actions

1. evaluate \( E_0 \) and obtain a list \( L \)
2. if \( L \) is \( \text{nil} \) then evaluate \( E_1 \).
3. if \( L \) is of the form \( h::t \), then evaluate \( E_2(h, t) \) under the binding where \( h \) is the head of \( L \) and \( t \) is the tail of \( L \).
\[
\text{case } E_0 \text{ of } \text{nil} \Rightarrow E_1 \\
| (h::t) \Rightarrow E_2(h,t)
\]
A simple program using pattern matching:

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

val it = 0 : int
val it = 1 : int
The general form of pattern matching

```
case exp of  pat₁ => exp₁  |  pat₂ => exp₂  |  ⋯  |  patₙ => expₙ
```

where \( patᵢ \) is a pattern consisting of

- constants,
- variables,
- data structure construction (e.g. list). For lists, we can include patterns of the forms:
  - \texttt{nil},
  - \texttt{pat₁::pat₂},
  - \texttt{[pat₁, ⋯, patₙ]}
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:

\[
\begin{align*}
\text{fun } f \quad &\text{pat}_1 = \exp_1 \\
| \quad &f \quad \text{pat}_2 = \exp_2 \\
\vdots \\
| \quad &f \quad \text{pat}_n = \exp_n
\end{align*}
\]

\[
\begin{align*}
\text{fun } f \ x &\ = \ \text{case } x \ \text{of} \\
&\quad \pat_1 \Rightarrow \exp_1 \\
| &\quad \pat_2 \Rightarrow \exp_2 \\
\vdots \\
| &\quad \pat_n \Rightarrow \exp_n
\end{align*}
\]

\[
\begin{align*}
\text{fn } &\text{pat}_1 \Rightarrow \exp_1 \\
| &\text{pat}_2 \Rightarrow \exp_2 \\
\vdots \\
| &\text{pat}_n \Rightarrow \exp_n
\end{align*}
\]

\[
\begin{align*}
\text{fn } x &\Rightarrow \text{case } x \ \text{of} \\
&\quad \pat_1 \Rightarrow \exp_1 \\
| &\quad \pat_2 \Rightarrow \exp_2 \\
\vdots \\
| &\quad \pat_n \Rightarrow \exp_n
\end{align*}
\]
Some examples:

```haskell
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:

\[
\text{fun sumList nil = 0}
\]
\[
| \text{sumList (h::t) = h + sumList t}
\]

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

1. If the give list is \texttt{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 \texttt{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 $(h,R) \Rightarrow exp$ 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 $R^k$.

   $$R^1 = R$$
   $$R^k = R \times R^{k-1} \quad (k \geq 2)$$
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.

```haskell
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 $R^+$. 

\[ R^+ = R^1 \cup \cdots \cup R^N \text{ is } \bigwedge_{k=1}^{N} (h, f, z) = h(f(N), \cdots, h(f(1), z) \cdots) \]

with $h$ to be the set union and $f(k) = R^k$, $z = \emptyset$.

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

```plaintext
- 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 \( \textit{a} \rightarrow \textit{bool} \) and a list of type \( \textit{a} \text{ list} \) 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 \texttt{map, flatten, member, unique, prefixList, permutations}

(2) \texttt{forall} which takes a list and a predicate \( P \) (a function that return \texttt{bool}) and checks whether the list contains an element that is true of \( P \), and

\texttt{exists} which takes a list and a predicate \( P \) (a function that return \texttt{bool}) and check whether all the elements of the list is true of \( P \) By definition, for any \( P \), \texttt{exists \ P \ nil} is false and \texttt{forall \ P \ nil} is true.

2.1. \texttt{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. \texttt{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 \texttt{foldl} that performs the following computation.

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

(1) Give a definition of \texttt{foldl}.
(2) Define \texttt{rev} using \texttt{foldl}.

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

(1) Give an example of \( R \) in normal form such that \( \texttt{tc } R \) is not in normal form.
(2) Rewrite \( \texttt{tc} \) so that it always return a normal relation for any normal input.
Define the following functions on relations:

1. **isRelated** which tests whether a given pair \((a, b)\) is related in a given relation \(R\).

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 relation \(R\) 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 \( \tau \) is defined inductively as:

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

This is defined as:

```ocaml
- 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))

The pre-order representation is: $a(b()())(c(d()())())$. (φ は空の木を表す)
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:

```haskell
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*

datatype typeSpec =
    Con₁ ⟨of type₁⟩
    | Con₂ ⟨of type₂⟩
    ...
    | Conₙ ⟨of typeₙ⟩

• *typeSpec* specify the type to be defined
• the right hand side of = specify the type of each component
Using Data Structures with Pattern Matching

In case expression of the form

\[
\text{case } \mathtt{exp} \text{ of } \mathtt{pat}_1 \Rightarrow \mathtt{exp}_1 \mid \mathtt{pat}_2 \Rightarrow \mathtt{exp}_2 \mid \cdots \mid \mathtt{pat}_n \Rightarrow \mathtt{exp}_n
\]

\(\mathtt{pat}_i\) can contain:

1. variables,
2. constants,
3. any data constructors defined in \texttt{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

```haskell
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

\texttt{infix 5 ::}

\texttt{datatype 'a list = nil | :: of 'a * 'a list}

Booleans

\texttt{datatype bool = true | false}

Special forms for bool are defined as:

\[ exp_1 \ \text{andalso} \ exp_2 \] \[ \Rightarrow \] \texttt{case exp}_1\texttt{ of true => exp}_2\ \texttt{| false => false}

\[ exp_1 \ \text{orelse} \ exp_2 \] \[ \Rightarrow \] \texttt{case exp}_1\texttt{ of false => exp}_2\ \texttt{| true => true}

\[ \text{if exp then } exp_1 \ \text{else } exp_2 \] \[ \Rightarrow \] \texttt{case exp of true => exp}_1\ \texttt{| 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

```haskell
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:

```haskell
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:

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

Examples:

```haskell
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:

   \[
   \begin{align*}
   \text{hd} & \Rightarrow \text{HD} \\
   \text{tl} & \Rightarrow \text{TL} \\
   \text{null} & \Rightarrow \text{NULL} \\
   h::t & \Rightarrow \text{CONS}(h, \text{fn}() \Rightarrow t)
   \end{align*}
   \]

Example

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

- NTH 100000000 naturalNumbers;

val it = 100000000 : int```

A more complicated example:
For finite lists:

```plaintext
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:

```plaintext
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 infilist
- TAKE 20 PRIMES;
- 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 \ldots val x_n = exp_n in \text{exp} end`, ML evaluates $exp_1, \ldots, exp_n$ in this order and then evaluate $\text{exp}$.

- Tuples $(exp_1, \ldots, exp_n)$ and records \{\(l_1 = exp_1, \ldots, l_n = exp_n\}\} are evaluated left to right.

- For function applications $exp_1 \ exp_2$, ML first evaluates $exp_1$ to obtain a function \(\text{fn } x \Rightarrow exp_0\), 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", "c", ..., "z", "aa", "ab", ..., "az", "ba", ...

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:

```ml
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 $[\text{ord}(s_1), \text{ord}(s_2), \cdots, \text{ord}(s_n)]$ of character codes.

next “increments” this list as:

```ocaml
define state as 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:

\[(\text{fn } x \Rightarrow x = x) \; exp\]

and

\[exp = exp\]

have the same meaning.

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

\[- (\text{fn } x \Rightarrow x = x) \; \text{(ref 1)};\]
\[val \; it = \text{true} : \text{bool}\]
\[- \; \text{ref 1} = \text{ref 1};\]
\[val \; it = \text{false} : \text{bool}\]
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 \textit{exnId}

exception \textit{exnId} of \(\tau\)

raise generates exception.

raise \textit{exnId}

raise \textit{exnId} \textit{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

<table>
<thead>
<tr>
<th>exception name</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bind</td>
<td>bind failure</td>
</tr>
<tr>
<td>Chr</td>
<td>illegal character code</td>
</tr>
<tr>
<td>Div</td>
<td>divide by 0</td>
</tr>
<tr>
<td>Empty</td>
<td>illegal usage of hd and th</td>
</tr>
<tr>
<td>Match</td>
<td>pattern matching failure</td>
</tr>
<tr>
<td>Option</td>
<td>empty option data</td>
</tr>
<tr>
<td>Overflow</td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>array etc too big</td>
</tr>
<tr>
<td>Subscript</td>
<td>index out of range</td>
</tr>
</tbody>
</table>
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

fun strictPower n m = if n = 0 andalso m = 0 then raise Undefined else power n m;

val strictPower = fn : int -> int -> int

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,dict)
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 NotFoundException)
    handle Found v => v
  end
Exercise Set (5)

1. Complete the function `decompose`:

   - `decompose "a(b()())(c(d()())())"`;
   `val it = ("a", "b()()", "c(d()())()") : 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 $\text{Node}(x,L,R)$.

and should have the following structure and type:

\[
\begin{align*}
- \text{fun } \text{treeFold } f \ z \ \text{Empty} &= \ z \\
| \text{treeFold } f \ z \ (\text{Node } (x,L,R)) &= \ldots \\
\end{align*}
\]

\[
\begin{align*}
\text{val } \text{treeFold} &= \ fn : \ ('a \times 'b \times 'b) \rightarrow 'b \rightarrow 'a \ \text{tree} \rightarrow 'b \\
\end{align*}
\]

(1) Complete the definition of $\text{treeFold}$.

(2) Re-define $\text{nodes}$, $\text{sumTree}$, and $\text{mapTree}$ using $\text{treeFold}$.

5. Define the following function of type $'a \ \text{list} \rightarrow 'a \ \text{option}$:

(1) $\text{car}$ that returns the head of a given list.

(2) $\text{cdr}$ that returns the tail of a given list.

(3) $\text{last}$ that returns the last element of a given list.
6. Write a function

```ocaml
define 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

```ocaml
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 \( n \)th the element in a given infinite list.

11. Complete the definition of **genSym**.

12. Define a function

   \[
   \text{makeGensym} : \text{char list} \to \text{unit} \to \text{string}
   \]

   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", ….
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 $id_1.id_2\ldots.id_{n-1}.id_n.id$.

   ```
   - val q = IntQueue.newQueue();
   val q = ref [] : queue
   - map (fn x => IntQueue.enqueue(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 \textit{id} is to import all the component defined in the structure \textit{id} to the current environment.

- open IntQueue;

  opening IntQueue
  
  \texttt{type queue = int list ref}

  \texttt{exception EmptyQueue}

  \texttt{val newQueue : unit -> queue}

  \texttt{val enqueue : 'a * 'a list ref -> unit}

  \texttt{val dequeue : 'a list ref -> 'a}

  \texttt{val removeLast : 'a list -> 'a list * 'a}

- dequeue();

  \texttt{val it = 3 : int}
Example of module programming: **functional queue** that replace IntQueue.

A functional queue = \((\text{newItems list}, \text{oldItems list})\)

- **newItems**: the elements recently added to the queue
- **oldItems**: the old elements that are removed soon.

and the system maintain the invariant

\[
\text{entire queue} = \text{newItems} \circlearrowleft \text{the reversal of oldItems}
\]

so that if

\[
\text{newItems} = [a_1, \ldots, a_n] \\
\text{oldItems} = [b_1, \ldots, b_m]
\]

then

\[
L = [a_1, \ldots, a_n, b_m, \ldots, b_1]
\]
• enqueue adds 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:

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

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

```plaintext
    structure Q = IntQueue
```

to

```plaintext
    structure Q = FastIntQueue.
```
Specifying Module Signature

As we have learned, 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

\[ \text{signature } \textit{sigId} = \text{sig} \text{ various specs end} \]
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.

```plaintext
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

```plaintext
structure structId : sigSpec = module
```

Example:

```plaintext
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;
- val q = AbsIntQueue.newQueue();
- val q = - : queue
- AbsIntQueue.enqueue (1,q);
- val it = () : unit
- AbsIntQueue.dequeue q;
- val it = 1 : int
But you cannot do the following:

- \( q := [1] ; \)

\textit{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:

```ml
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 define a queue for integers as:

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

and also a queue for strings as:

```ml
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.

```ocaml
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:

```ocaml
- 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 programmed 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

```haskell
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

\textit{functorId(\textit{various declarations})}

Example:

\texttt{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

\[ [E_1, \ldots, E_n] \text{ one of } E_1 \text{ through } E_n \]
\[ E^* \text{ 0 or more } E \]
\[ E^+ \text{ 1 or more } E \]
Constant and Identifiers

\[ scon \ := \ int \ | \ word \ | \ real \ | \ string \ | \ char \quad \text{(constant)} \]
\[ int \ := \ \langle \sim \rangle [0-9]+ \quad \text{(decimals)} \]
\[ \quad | \ \langle \sim \rangle \text{0x}[0-9,a-f,A-F]+ \quad \text{hexa decimal notation} \]
\[ word \ := \ 0w[0-9]+ \quad \text{(unsigned decimal)} \]
\[ \quad | \ 0wx[0-9,a-f,A-F]+ \quad \text{(unsigned hexadecimal)} \]
\[ real \ := \ integers \cdot [0-9]+ \ [E,e] \langle \sim \rangle [0-9]+ \quad \text{(reals)} \]
\[ \quad | \ integers \cdot [0-9]+ \]
\[ \quad | \ integers \ [E,e] \langle \sim \rangle [0-9]+ \]
\[ char \ := \ \#"[printable,escape]" \quad \text{(characters)} \]
\[ string \ := \ " [printable,escape]* " \quad \text{(strings)} \]

\text{printable} \text{ is the set of printable characters except for } \setminus \text{ and } "\text{.}
\texttt{escape} ::= \texttt{\a} \quad \text{warning (ASCII 7)} \\
| \texttt{\b} \quad \text{backspace (ASCII 8)} \\
| \texttt{\t} \quad \text{tab (ASCII 9)} \\
| \texttt{\n} \quad \text{new line (ASCII 10)} \\
| \texttt{\v} \quad \text{vertical tab (ASCII 11)} \\
| \texttt{\f} \quad \text{home feed (ASCII 12)} \\
| \texttt{\r} \quad \text{return (ASCII 13)} \\
| \texttt{\^C} \quad \text{control character } C \\
| \texttt{\\} \quad \text{itself} \\
| \texttt{\"} \quad " \text{itself} \\
| \texttt{\\"} \quad " \text{itself} \\
| \texttt{\dd} \quad \text{character having the code } ddd \text{ in decimal} \\
| \texttt{\f \ldots f} \quad \text{ignore } f \ldots f \text{ where } f \text{ is some format character} \\
| \texttt{\ux} \quad \text{unicode}
# Classes of Identifiers Used in the Syntax Definition

<table>
<thead>
<tr>
<th>class</th>
<th>contents</th>
<th>note</th>
</tr>
</thead>
<tbody>
<tr>
<td>vid</td>
<td>variables</td>
<td>long</td>
</tr>
<tr>
<td>tyvar</td>
<td>type variables starting with '</td>
<td></td>
</tr>
<tr>
<td>tycon</td>
<td>type constructors</td>
<td>long</td>
</tr>
<tr>
<td>lab</td>
<td>record labels</td>
<td></td>
</tr>
<tr>
<td>strid</td>
<td>structure names</td>
<td>long</td>
</tr>
<tr>
<td>sigid</td>
<td>signature names</td>
<td>alphanumeric</td>
</tr>
<tr>
<td>funid</td>
<td>functor names</td>
<td>alphanumeric</td>
</tr>
</tbody>
</table>

## Long Identifiers

\[
\text{long}X ::= X \mid \text{strid}_1 \cdots \text{strid}_n . X
\]
Syntax of the Core ML

Some auxiliary definitions:

\[ xSeq ::= x \quad \text{one element} \]
\[ \quad | \quad \text{empty sequence} \]
\[ \quad | \quad (x_1, \ldots, x_n) \quad \text{finite sequence (} n \geq 2 \text{)} \]
Syntax for Expressions

\[
exp ::= \text{infix} \\
| exp : ty \\
| \text{exp andalso exp} \\
| \text{exp orelse exp} \\
| \text{exp handle match} \\
| \text{raise exp} \\
| \text{if exp then exp else exp} \\
| \text{while exp do exp} \\
| \text{case exp of match} \\
| \text{fn match}
\]

\[
match ::= mrule \langle \mid \text{match} \rangle
\]

\[
mrule ::= \text{pat => exp}
\]
Function Applications and Infix Expressions

\[
\begin{align*}
\text{appexp} &::= \text{atexp} \\
\text{appexp} \text{ atexp} &\quad \text{(left associative)} \\
\text{infix} &::= \text{appexp} \\
\text{infix vid infix} &
\end{align*}
\]
Atomic Expressions

\[
\text{atexp} ::= \text{scon} \\
\quad | \langle \text{op} \rangle \text{longvid} \\
\quad | \{\langle \text{exprow} \rangle \} \\
\quad | () \\
\quad | (\text{exp}_1, \cdots, \text{exp}_n) \\
\quad | [\text{exp}_1, \cdots, \text{exp}_n] \\
\quad | (\text{exp}_1; \cdots; \text{exp}_n) \\
\quad | \text{let} \ \text{dec} \ \text{in} \ \text{exp}_1; \cdots; \text{exp}_n \ \text{end} \\
\quad | (\text{exp})
\]

\[
\text{exprow} ::= \text{lab} = \text{exp} \langle , \exprow \rangle
\]
Patterns

\[ atpat \ ::= \ scon \]
\[ \quad | \langle \text{op} \rangle \text{longvid} \]
\[ \quad | \{ \langle \text{patrow} \rangle \} \]
\[ \quad | () \]
\[ \quad | (\text{pat}_1, \ldots, \text{pat}_n) \]
\[ \quad | [\text{pat}_1, \ldots, \text{pat}_n] \]
\[ \quad | (\text{pat}) \]

\[ patrow \ ::= \ldots \]
\[ \quad | \text{lab} = \text{pat} \langle , \text{patrow} \rangle \]
\[ \quad | \text{vid}\langle : \text{ty}\rangle \langle \text{as pat} \rangle \langle , \text{patrow} \rangle \]

\[ pat \ ::= \ atpat \]
\[ \quad | \langle \text{op} \rangle \text{longvid atpat} \]
\[ \quad | \text{pat vid pat} \]
\[ \quad | \text{pat} : \text{ty} \]
\[ \quad | \langle \text{op} \rangle \text{pat} \langle : \text{ty} \rangle \text{as pat} \]
Types

\[ ty ::= tyvar \]
\| \{\langle tyrow\rangle\} \]
\| tySeq longtycon \]
\| \(ty_1 \ldots * ty_n\) \]
\| ty -> ty \]
\| (ty) \]

\[ tyraw ::= lab : \langle , tyraw \rangle \]
Declarations (1)

\[
\begin{align*}
\text{dec} &= \text{val\ tyvarSeq\ valbind} \\
          &\quad | \text{fun\ tyvarSeq\ funbind} \\
          &\quad | \text{type\ tybind} \\
          &\quad | \text{datatype\ datbind\ ⟨withtyp\ tybind⟩} \\
          &\quad | \text{datatype\ tycon = longtycon} \\
          &\quad | \text{exception\ exbind} \\
          &\quad | \text{local\ dec\ in\ dec\ end} \\
          &\quad | \text{opne\ longstrid_{1} \cdots longstrid_{n}} \\
          &\quad | \text{dec ; dec} \\
          &\quad | \text{infix\ ⟨d⟩\ vid_{1} \cdots vid_{n}} \\
          &\quad | \text{infixr\ ⟨d⟩\ vid_{1} \cdots vid_{n}} \\
          &\quad | \text{nonfix\ vid_{1} \cdots vid_{n}} \\
\text{valbind} &= \text{pat = exp\ ⟨and\ valbind⟩} \\
               &\quad | \text{rec\ valbind}
\end{align*}
\]
Declarations (2)

funbind ::=  \( \langle \text{op} \rangle \text{vid atpat}_{11} \cdots \text{atpat}_{1n} \langle : \text{ty} \rangle = \text{exp}_1 \)  \((m, n \geq 1)\)
| \( \langle \text{op} \rangle \text{vid atpat}_{21} \cdots \text{atpat}_{2n} \langle : \text{ty} \rangle = \text{exp}_2 \)
| \( \cdots \)
| \( \langle \text{op} \rangle \text{vid atpat}_{m1} \cdots \text{atpat}_{mn} \langle : \text{ty} \rangle = \text{exp}_m \)

tybind ::=  \text{tyvarSeq tycon} = \text{ty} \langle \text{and tybind} \rangle

datbind ::=  \text{tyvarSeq tycon} = \text{conbind} \langle \text{and datbind} \rangle

conbind ::=  \langle \text{op} \rangle \text{vid} \langle \text{of ty} \rangle \langle | \text{conbind} \rangle

exbind ::=  \langle \text{op} \rangle \text{vid} \langle \text{of ty} \rangle \langle \text{and exbind} \rangle
| \langle \text{op} \rangle \text{vid} = \langle \text{op} \rangle \text{lognvid} \langle \text{and exbind} \rangle
Declarations (3)

\[
todec ::= \text{strdec} \langle \text{topdec} \rangle \\
| \text{sigdec} \langle \text{topdec} \rangle \\
| \text{fundec} \langle \text{topdec} \rangle
\]
Structures

\[
\begin{align*}
\text{strdec} & \ ::= \ \text{dec} \\
& \quad \mid \text{structure strbind} \\
& \quad \mid \text{local strdec in strdec end} \\
& \quad \mid \text{strdec \langle ; \rangle strdec} \\
\text{strbind} & \ ::= \ \text{strid} = \text{strexp \langle and strbind \rangle} \\
& \quad \mid \text{strid : sigexp = strexp \langle and strbind \rangle} \\
& \quad \mid \text{strid :> sigexp = strexp \langle and strbind \rangle} \\
\text{strexp} & \ ::= \ \text{struct strdec end} \\
& \quad \mid \text{longstrid} \\
& \quad \mid \text{strexp : sigexp} \\
& \quad \mid \text{strexp :> sigexp} \\
& \quad \mid \text{funid (strid : sigexp)} \\
& \quad \mid \text{funid (strdec)}
\end{align*}
\]
Signature

\[
\begin{align*}
\text{sigdec} & ::= \text{signature sigbind} \\
\text{sigexp} & ::= \text{sig spec end} \\
& \quad | \text{sigid} \\
& \quad | \text{sigid where type} \\
& \quad | \text{tyvarSeq longtycon= ty} \\
\text{sigbind} & ::= \text{sigid = sigexp (and sigbind)}
\end{align*}
\]
Specifications (1)

\[ spec ::= \text{val } val\text{desc} \]
\[ \quad \text{type } typ\text{desc} \]
\[ \quad \text{eqtype } typ\text{desc} \]
\[ \quad \text{datatype } dat\text{desc} \]
\[ \quad \text{datatype } tycon = \text{datatype } longtycon \]
\[ \quad \text{exception } ex\text{desc} \]
\[ \quad \text{structure } str\text{desc} \]
\[ \quad \text{include } sigexp \]
\[ \quad spec \langle ; \rangle spec \]
\[ \quad spec \text{ sharing type} \]
\[ \quad longtycon_1 = \cdots = longtycon_n \]
Specifications (2)

\[
\begin{align*}
\text{valdesc} &::= \text{vid : ty} \langle \text{and valdesc} \rangle \\
\text{typdesc} &::= \text{tyvarSeq tycon} \langle \text{and typdesc} \rangle \\
\text{datdesc} &::= \text{tyvarSeq tycon = condesc} \langle \text{and datdesc} \rangle \\
\text{condesc} &::= \text{vid} \langle \text{of ty} \rangle \langle | \text{condesc} \rangle \\
\text{exdesc} &::= \text{vid} \langle \text{of ty} \rangle \langle \text{and exdesc} \rangle \\
\text{strdesc} &::= \text{strid : sigexp} \langle \text{and strdesc} \rangle
\end{align*}
\]
Functors

\[\text{fundec} ::= \text{functor} \; \text{funbind}\\
\text{funbind} ::= \text{funid} \; (\; \text{strid} : \text{sigexp}) \; \langle : \text{sigexp} \rangle = \text{strexp} \; \langle \text{and funbind} \rangle\\
\quad | \; \text{funid} \; (\; \text{strid} : \text{sigexp}) \; :> \text{sigexp} = \text{strexp} \; \langle \text{and funbind} \rangle\\
\quad | \; \text{funid} \; (\; \text{spec} \;) \; \langle : \text{sigexp} \rangle = \text{strexp} \; \langle \text{and funbind} \rangle\\
\quad | \; \text{funid} \; (\; \text{spec} \;) \; \langle :> \text{sigexp} \rangle = \text{strexp} \; \langle \text{and funbind} \rangle\]
STANDARD ML BASIS LIBRARY
# The Contents of the Library

## Mandatory Library

<table>
<thead>
<tr>
<th>structure 名</th>
<th>signature 名</th>
<th>remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>ARRAY</td>
<td>polymorphic array</td>
</tr>
<tr>
<td>BinIO</td>
<td>BIN_IO</td>
<td>binary IO</td>
</tr>
<tr>
<td>BinPrimIO</td>
<td>PRIM_IO</td>
<td>low level IO</td>
</tr>
<tr>
<td>Bool</td>
<td>BOOL</td>
<td></td>
</tr>
<tr>
<td>Byte</td>
<td>BYTE</td>
<td></td>
</tr>
<tr>
<td>Char</td>
<td>CHAR</td>
<td></td>
</tr>
<tr>
<td>CharArray</td>
<td>MONO_ARRAY</td>
<td></td>
</tr>
<tr>
<td>CharVector</td>
<td>MONO_VECTOR</td>
<td></td>
</tr>
<tr>
<td>CommandLine</td>
<td>COMMAND_LINE</td>
<td>command line parameters</td>
</tr>
<tr>
<td>Date</td>
<td>DATE</td>
<td></td>
</tr>
<tr>
<td>General</td>
<td>GENERAL</td>
<td></td>
</tr>
<tr>
<td>structure name</td>
<td>signature mane</td>
<td>remark</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>IEEEReal</td>
<td>IEEE_REAL</td>
<td></td>
</tr>
<tr>
<td>Int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>IO</td>
<td>IO</td>
<td></td>
</tr>
<tr>
<td>LargeInt</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>LargeReal</td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td>LargeWord</td>
<td>WORD</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td>LIST</td>
<td></td>
</tr>
<tr>
<td>ListPair</td>
<td>LIST_PAIR</td>
<td></td>
</tr>
<tr>
<td>Math</td>
<td>MATH</td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>OPTION</td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>OS</td>
<td>system interface</td>
</tr>
<tr>
<td>OS.FileSys</td>
<td>OS_FILE_SYS</td>
<td></td>
</tr>
<tr>
<td>OS.IO</td>
<td>OS_IO</td>
<td></td>
</tr>
<tr>
<td>structure name</td>
<td>signature name</td>
<td>remark</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>OS.Path</td>
<td>OS_PATH</td>
<td></td>
</tr>
<tr>
<td>OS.Process</td>
<td>OS_PROCESS</td>
<td></td>
</tr>
<tr>
<td>Position</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>Real</td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td>SML90</td>
<td>SML90</td>
<td>for compatibility</td>
</tr>
<tr>
<td>String</td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td>StringCvt</td>
<td>STRING_CVT</td>
<td>string utilities</td>
</tr>
<tr>
<td>Substring</td>
<td>SUBSTRING</td>
<td>substrings</td>
</tr>
<tr>
<td>TextIO</td>
<td>TEXT_IO</td>
<td></td>
</tr>
<tr>
<td>TextPrimIO</td>
<td>PRIM_IO</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>TIME</td>
<td></td>
</tr>
<tr>
<td>Timer</td>
<td>TIMER</td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td>VECTOR</td>
<td></td>
</tr>
<tr>
<td>structure name</td>
<td>signature name</td>
<td>remark</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>-------</td>
</tr>
<tr>
<td>Vector</td>
<td>VECTOR</td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>WORD</td>
<td></td>
</tr>
<tr>
<td>Word8</td>
<td>WORD</td>
<td></td>
</tr>
<tr>
<td>Word8Array</td>
<td>MONO_ARRAY</td>
<td></td>
</tr>
<tr>
<td>Word8Vector</td>
<td>MONO_VECTOR</td>
<td></td>
</tr>
</tbody>
</table>
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) StringCvt.reader
  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) StringCvt.reader
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 -> order
val <= : string * string -> bool
val < : string * string -> bool
val >= : string * string -> bool
val > : string * string -> bool
end
signature INTEGER =
  sig
    eqtype int
    val precision : 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) StringCvt.reader
val fmt : StringCvt.radix -> int -> string
end
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 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 \rightarrow int
val realFloor : real \rightarrow real
val realCeil : real \rightarrow real
val realTrunc : real \rightarrow real
val toInt : IEEEReal.rounding_mode
    \rightarrow real \rightarrow int
val toLargeInt : IEEEReal.rounding_mode
    \rightarrow real \rightarrow Int32.int
val fromInt : int \rightarrow real
val fromLargeInt : Int32.int \rightarrow real
val toLarge : real \rightarrow Real64.real
val fromLarge : IEEEReal.rounding_mode
    \rightarrow Real64.real \rightarrow real
val toDecimal : real \rightarrow IEEEReal.decimal_approx
val fromDecimal : IEEEReal.decimal_approx \rightarrow 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 list
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

<table>
<thead>
<tr>
<th>type</th>
<th>structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>eqtype unit</td>
<td>General</td>
</tr>
<tr>
<td>eqtype int</td>
<td>Int</td>
</tr>
<tr>
<td>eqtype word</td>
<td>Word</td>
</tr>
<tr>
<td>type real</td>
<td>Real</td>
</tr>
<tr>
<td>eqtype char</td>
<td>Char</td>
</tr>
<tr>
<td>eqtype string</td>
<td>String</td>
</tr>
<tr>
<td>type substring</td>
<td>Substring</td>
</tr>
<tr>
<td>type exn</td>
<td>General</td>
</tr>
<tr>
<td>eqtype ’a array</td>
<td>Array</td>
</tr>
<tr>
<td>eqtype ’a vector</td>
<td>Vector</td>
</tr>
<tr>
<td>eqtype ’a ref</td>
<td></td>
</tr>
</tbody>
</table>
## Pre-defined datatypes

<table>
<thead>
<tr>
<th>type</th>
<th>structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>datatype bool = false</td>
<td>Bool</td>
</tr>
<tr>
<td>datatype 'a option = NONE</td>
<td>Option</td>
</tr>
<tr>
<td>datatype order = LESS</td>
<td>General</td>
</tr>
<tr>
<td>datatype 'a list = nil</td>
<td>List</td>
</tr>
</tbody>
</table>
### Predefined constants

<table>
<thead>
<tr>
<th>name</th>
<th>longvid</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref : 'a -&gt; 'a ref</td>
<td>General.</td>
</tr>
<tr>
<td>! : 'a ref -&gt; 'a</td>
<td>(built-in primitive)</td>
</tr>
<tr>
<td>:= : 'a ref * 'a -&gt; unit</td>
<td>General.:=</td>
</tr>
<tr>
<td>before : 'a * unit -&gt; 'a</td>
<td>General.before</td>
</tr>
<tr>
<td>ignore : 'a -&gt; unit</td>
<td>General.ignore</td>
</tr>
<tr>
<td>exnName : exn -&gt; string</td>
<td>General.exnName</td>
</tr>
<tr>
<td>exnMessage : exn -&gt; string</td>
<td>General.exnMessage</td>
</tr>
<tr>
<td>o : ('a -&gt; 'b) * ('c -&gt; 'a) -&gt; 'c -&gt; 'b</td>
<td>General.o</td>
</tr>
<tr>
<td>getOpt : ('a option * 'a) -&gt; 'a</td>
<td>Option.getOpt</td>
</tr>
<tr>
<td>isSome : 'a option -&gt; bool</td>
<td>Option.isSome</td>
</tr>
<tr>
<td>valOf : 'a option -&gt; 'a</td>
<td>Option.valOf</td>
</tr>
<tr>
<td>not : bool -&gt; bool</td>
<td>Bool.not</td>
</tr>
<tr>
<td>name</td>
<td>defined in</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
</tr>
<tr>
<td>real : int -&gt; real</td>
<td>Real</td>
</tr>
<tr>
<td>trunc : real -&gt; int</td>
<td>Real</td>
</tr>
<tr>
<td>floor : real -&gt; int</td>
<td>Real</td>
</tr>
<tr>
<td>ceil : real -&gt; int</td>
<td>Real</td>
</tr>
<tr>
<td>round : real -&gt; int</td>
<td>Real</td>
</tr>
<tr>
<td>ord : char -&gt; int</td>
<td>Char</td>
</tr>
<tr>
<td>chr : int -&gt; char</td>
<td>Char</td>
</tr>
<tr>
<td>size : string -&gt; int</td>
<td>String</td>
</tr>
<tr>
<td>str : char -&gt; string</td>
<td>String</td>
</tr>
<tr>
<td>concat : string list -&gt; string</td>
<td>String</td>
</tr>
<tr>
<td>implode : char list -&gt; string</td>
<td>String</td>
</tr>
<tr>
<td>explode : string -&gt; char list</td>
<td>String</td>
</tr>
<tr>
<td>substring : string * int * int -&gt; string</td>
<td>String</td>
</tr>
<tr>
<td>name</td>
<td>defined in</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td><code>^</code> : string * string -&gt; string</td>
<td>String</td>
</tr>
<tr>
<td>null : 'a list -&gt; bool</td>
<td>List</td>
</tr>
<tr>
<td>hd : 'a list -&gt; 'a</td>
<td>List</td>
</tr>
<tr>
<td>tl : 'a list -&gt; 'a list</td>
<td>List</td>
</tr>
<tr>
<td>length : 'a list -&gt; int</td>
<td>List</td>
</tr>
<tr>
<td>rev : 'a list -&gt; 'a list</td>
<td>List</td>
</tr>
<tr>
<td>@ : ('a list * 'a list) -&gt; 'a list</td>
<td>List</td>
</tr>
<tr>
<td>app : ('a -&gt; unit) -&gt; 'a list -&gt; unit</td>
<td>List</td>
</tr>
<tr>
<td>map : ('a -&gt; 'b) -&gt; 'a list -&gt; 'b list</td>
<td>List</td>
</tr>
<tr>
<td>foldr : ('a * 'b -&gt; 'b) -&gt; 'b -&gt; 'a list -&gt; 'b</td>
<td>List</td>
</tr>
<tr>
<td>foldl : ('a * 'b -&gt; 'b) -&gt; 'b -&gt; 'a list -&gt; 'b</td>
<td>List</td>
</tr>
<tr>
<td>print : string -&gt; unit</td>
<td>TextIO</td>
</tr>
<tr>
<td>vector : 'a list -&gt; 'a vector</td>
<td>Vector</td>
</tr>
<tr>
<td>use : string -&gt; unit</td>
<td>(primitive)</td>
</tr>
</tbody>
</table>
### Overloaded identifiers

<table>
<thead>
<tr>
<th>name</th>
<th>default type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>int * int -&gt; int</td>
</tr>
<tr>
<td>-</td>
<td>int * int -&gt; int</td>
</tr>
<tr>
<td>*</td>
<td>int * int -&gt; int</td>
</tr>
<tr>
<td>div</td>
<td>int * int -&gt; int</td>
</tr>
<tr>
<td>mod</td>
<td>int * int -&gt; int</td>
</tr>
<tr>
<td>/</td>
<td>real * real -&gt; real</td>
</tr>
<tr>
<td>~</td>
<td>int -&gt; int</td>
</tr>
<tr>
<td>abs</td>
<td>int -&gt; int</td>
</tr>
<tr>
<td>&lt;</td>
<td>int * int -&gt; bool</td>
</tr>
<tr>
<td>&gt;</td>
<td>int * int -&gt; bool</td>
</tr>
<tr>
<td>&lt;=</td>
<td>int * int -&gt; bool</td>
</tr>
<tr>
<td>&gt;=</td>
<td>int * int -&gt; bool</td>
</tr>
</tbody>
</table>
\[
text := \{\text{string, char}\}
\]
\[
\text{wordint} := \{\text{word, int}\}
\]
\[
\text{realint} := \{\text{real, int}\}
\]
\[
\text{num} := \{\text{word, int, real}\}
\]
\[
\text{numtext} := \{\text{string, char, word, int, real}\}
\]
Binary Operators Declared at the Top-Level

\begin{itemize}
  \item \texttt{infix 7 * / div mod}
  \item \texttt{infix 6 + - ^}
  \item \texttt{infixr 5 :: @}
  \item \texttt{infix 4 = <> > >= < <=}
  \item \texttt{infix 3 := o}
  \item \texttt{infix 0 before}
\end{itemize}
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 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 option -> unit
val foldli : (int * 'a * 'b -> 'b)
  -> 'b -> 'a array * int * int option option -> 'b
val foldri : (int * 'a * 'b -> 'b)
  -> 'b -> 'a array * int * int option option -> 'b
val modifyi : (int * 'a -> 'a)
  -> 'a array * int * int option 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)
val mapi : (int * 'a -> 'b) -> 'a vector * int * int option -> unit
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
     in
     (sort a)@[p]@(sort b)
   end;
   
   This is not what you should write for an industrial strength sorting program.
Let’s run the sort function

Making a test data:

```haskell
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.

\[- \text{val data} = \text{makeList} 1000000;\]
\[\text{val data} = [...] : \text{int list}\]
\[- \text{sort data;}\]
\[\text{val data} = [...] : \text{int list}\]

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 $i$th 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 $A(k + 1, j)$. 
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 $i_3$.
4. Partition $A(i+1, j)$ using $A[i]$ as the pivot. Let $k$ be the largest index such that $A[k] \leq \text{pivot}$.

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

1. Scan from $a$ to the right and find the first $k$ such that $A[k] > pivot$.
2. Scan from $b$ to the left and find the first $l$ such that $A[l] \leq pivot$.
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:

```ml
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

```ml
- val a = randomArray 1000000;
  val a = [| .... |] : int array
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**

```ocaml
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.

```ocaml
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 \( n \log(n) \).

Check the behavior of the sort function:

```ocaml
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 elements 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:

```plaintext
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 and shows the following information:

```haskell
-> evalSort [100000,500000,1000000];
```

<table>
<thead>
<tr>
<th>Comparison</th>
<th>time (micro s)</th>
<th>micro s./n</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>0</td>
<td>0.000000000</td>
</tr>
<tr>
<td>500000</td>
<td>20</td>
<td>0.040000000</td>
</tr>
<tr>
<td>1000000</td>
<td>30</td>
<td>0.030000000</td>
</tr>
</tbody>
</table>

-----------------------------

| avarage     | 0.023333333     |

<table>
<thead>
<tr>
<th>Sorting</th>
<th>time (mil. s)</th>
<th># of comp / nlogn</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>Count</td>
<td>Result</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>-----------------</td>
</tr>
<tr>
<td>100000</td>
<td>190</td>
<td>4.90248850</td>
</tr>
<tr>
<td>500000</td>
<td>1140</td>
<td>5.16144689</td>
</tr>
<tr>
<td>1000000</td>
<td>2460</td>
<td>5.28952707</td>
</tr>
</tbody>
</table>

---

**Average**: 5.11782082
INPUT AND OUTPUT
Stream datatypes: `instream` and `outstream`

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

Two models 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 -> unit

...
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

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:

```markdown
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 | (* / *) | 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 ()
    | _ => ()
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 "") 
    in ID(getRest "") 
end
The lex function:

```haskell
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
**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.

```ocaml
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 * substring
val span : substring * substring -> substring
val translate : (char -> string) -> substring -> string
val fields : (char -> bool) -> substring -> substring list
val tokens : (char -> bool) -> substring -> substring list
... end
**StringCvt structure**

A collection of utility functions for formatting.

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

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 $\tau$, you can simply do the following.

1. Use `Substring.getc` as a character reader from `substring`,  
2. Define a function 

   \[
   \text{scan} : (\text{char, substring}) \rightarrow (\tau, \text{substring}) \text{ reader}.
   \]

3. Apply `scan` to `getc`.  

For each atomic type, a `scan` function is already given.

\[
\text{Int.scan : StringCvt.radix} \\
\rightarrow (\text{char, 'a}) \text{StringCvt.reader} \\
\rightarrow (\text{int, 'a}) \text{StringCvt.reader}
\]

- fun decScan x = Int.scan StringCvt.DEC x;

val decScan = fn : (char,'a) StringCvt.reader 
\rightarrow (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

\[
\begin{align*}
url & ::= \text{http://domain} \langle \text{path} \rangle \langle \text{anchor} \rangle \\
& \quad | \text{file://} \text{path} \\
& \quad | \text{relativePath} \\
\text{domain} & ::= \text{id} \mid \text{id} . \text{domain} \\
\text{path} & ::= \text{/relativePath} \\
\text{relativePath} & ::= \varepsilon \mid \text{id} / \text{relativePath} \\
\text{id} & ::= \text{a string of alpha numeric characters} \\
\text{anchor} & ::= \# \text{id}
\end{align*}
\]
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)
352
&

%


| "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
              end
      end
  in (host,anchor)
  end
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.

```ml
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:

```haskell
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:

\[%[−][dd]type\]

• — is for left align. The default is right align.
• \(dd\) 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 formatting strings, we define the following datatype.

```plaintext
datatype format = SPEC of formatSpec
    | LITERAL of string
```

LITERAL is a data string to be printed.
A function to parse a format string.

```ml
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 formatting string into \texttt{formatSpec}.

\begin{verbatim}
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
    end

end
\end{verbatim}
of "d" => INT StringCvt.DEC
| "s" => STRING
| "f" => REAL (StringCvt.FIX NONE)
| "e" => REAL (StringCvt.SCI NONE)
| "g" => REAL (StringCvt.GEN NONE)
| _ => raise formatError},

end

end
The formatting function.

```haskell
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 formatting 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
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

- 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。