nML 프로그래밍 걸음마
nML Primer

Kwangkeun Yi
cs.kaist.ac.kr/~kwang
Research On Program Analysis System
National Creative Research Initiative Center
Division of Computer Science
KAIST

Spring 2001

목 차

1 장

Why nML?

Implementing a software in nML has numerous advantages over current practice.

- complete *type safety*: if a program passes the compiler's type checker, the program will never "dump core" at run time.
- no need to declare types of expressions: nML's type-checking algorithm not only checks but also can infer types from program text. This techniques is called *type reconstruction* or *type inference*. Type annotation can optionally be provided by the programmer, in which case they are treated as assertions that are checked at compile-time.
- *polymorphic* functions: programmer can define a function that can be applied to infinitely many types of values without violating the type safety.
- *module* system: a support for programming-in-large. (C's analogy is a pair of `.h` and `.c` files.) An nML module can be parameterized. For example, a generic quick-sort module may be defined with, as parameters, the input data set and some operations. By applying this generic module to appropriate parameters, various quick-sort programs are acquired: a quick-sort program for integer arrays, string linked-list, and etc.
- automatic memory management: run-time support for garbage collection. Programmers don't have to `malloc` and `free`. No memory leak.

- small, simple, and has expressive power (as a general-purpose language) with minimized conceptual complexity.

정확하고 빠뜨림없는 nML의 정의를 보고 싶은 사람은 다음의 자료를 읽기바랍니다:

- **프로그래밍 언어 nML**, 프로그램 분석 시스템 연구단, KAIST, 2001 (`http://ropas.kaist.ac.kr/n/doc/n.ps`)

2 장

A Gentle Introduction to nML

1 Interaction with nML Compiler System

The "read-eval-print" dialogue in which an expression is entered, nML analyzes, compiles, and executes it, and the result is printed.

```
# 1+2 ;;
- :  int = 3
```

Note that

- the end of an expression is marked by `;;`. We can type-in an expression across multiple lines by entering the newline without `;;`.

You can also use the batch compiler `nmlc`. Its use is almost identitical to the c batch compiler `cc` or `gcc`. Please see other materials in `http://ropas.kaist.ac.kr/n` for its use.

2 Values of Base Types

2.1 Unit

The type `unit` consists of a single value, written `()`.

```
# ();;
- :  unit = ()
```

This value is used when an expression has no interesting value (e.g. an assignment expression), or when a function has no arguments.

2.2 Booleans

The type `bool` consists of `true` and `false`.

```
# true ;;
- :  bool = true
# false ;;
- :  bool = false
```

Primitive functions on boolean values are `orelse`, `andalso`, and `not`.

```
# true orelse false ;;
- :  bool = true
# true andalso false ;;
- :  bool = false
# not true ;;
- :  bool = false
```

Conditional expression `if` e_1 `then` e_2 `else` e_3 requires that e_1 has a boolean value and e_2 and e_3 have the same type.

2.3 Integer

The type `int` has infinite elements. Negative integer is prefixed with `-`.

```
# 0001 ;;
- :  int = 1
# 2 ;;
- :  int = 2
# -2 ;;
- :  int = -2
# +2 ;;
- :  int = 2
```

Primitive functions on integers are `+`, `-`, `*`, `/`, `div`, `mod`, `=`, `<`, `>`, `<=`, `>=`.

2.4 Real

The type `real` also has infinite elements.

```
# 0.1 ;;
- :  float = 0.1
# 1E2 ;;
- :  float = 100
# 1.0E01 ;;
- :  float = 10
```

Primitive functions on reals are `+,-,*,/,=,<,>,<=,>=`.

2.5 String

```
# "KAIST" ;;
- :  string = "KAIST"
```

String concatenation operator (infix) is `^`.

```
# "KAIST" ^"/" ^"CS" ;;
- :  string = "KAIST/CS"
```

See the manual for other operations on strings.

3 Values of Compound Types

3.1 Tuple: Product Type

A pair of integers is of type `int * int`. In nML, a value of a tuple type is written inside parentheses, where component values are separated by comma.

```
# (true , () ) ;;
- :  bool * unit = (true, ())
# ("blue" , 1 , true ) ;;
- :  string * int * bool = ("blue", 1, true)
```

Equality between tuples is component-wise.

```
# (1 , true ) = ( 0 + 1 , false orelse true ) ;;
- :  bool = true
# (1 , true ) = ( 1 , false ) ;;
- :  bool = false
# (1 , true ) = ( 1 , "a" ) ;;
This expression has type (int * bool) * (int * string)
but is here used with type (int * bool) * (int * bool)
```

3.2 List

A list consists of finite sequence of values of the same type.

```
# [1,2] ;;
- :  int list = [1, 2]
# [1, true ] ;;
This expression has type bool list but is here used with type int lis
```

Empty list is `nil`

```
# nil ;;
- :  'a list = []
```

The `'a list` means "any list." `'a` is called *type variable*. There are two primitive operations on lists, `::` for "cons" and `@` for "append":

```
# 1::nil ;;
- :  int list = [1]
# [1]@[2] ;;
- :  int list = [1, 2]
```

3.3 Record: Labeled Tuple

To use record, we should define the type of record first.
The last compound type is the record type.

```
# {name = "kaist" , age = 25 } ;;
Unbound label name
# type info = { name :  string , age :  int } ;;
```

```
type info = { name:  string; age:  int }
# {name = "kaist" , age = 25 };;
- :  info = {name="kaist", age=25}
# val x = { name = "kaist" , age = 25 } ;;
val x :  info = {name="kaist", age=25}
```

A field value is selected as

```
# x.age ;;
- :  int = 25
```

3.4 Functions

A value in an arrow type (from argument type to result type) is a function. In nML, a function is treated exactly like other values. A function can be a component of a tuple, of a record, can be passed to another function, can be returned as a function application, and so on. No special treatment is required.

```
# fn x => x + 1 ;;
- :  int -> int = <fun>
# fn (x,y) => if x then y else y-1 ;;
- :  bool * int -> int = <fun>
# fn (x,y,z) => if x then y else z ;;
- :  bool * 'a * 'a -> 'a = <fun>
# fn (x,y) => x@y ;;
- :  'a list * 'a list -> 'a list = <fun>
# fn x => fn y => x + y + 1 ;;
- :  int -> int -> int = <fun>
```

How do we define recursive functions? We need a name for a function. See below.

4 Values of User-defined Types

The type system of nML is extensible.

4.1 Type Abbreviation

New name for the same type is defined.

```
# type intpair = int * int ;;
type intpair = int * int
```

New type name is used to abbreviate complex type expressions, primarily for the sake of readability, rather than to introduce a new type.[1] This is sometimes called *transparent type binding*. Why "transparent?" Because new type name does not hide anything; type abbreviation and its defining type expression is interchangeable. In the above example, `intpair` and `int * int` are same indeed.

4.2 Datatype Binding

A data type is specified by declaring a new type name (called *type constructor*) and providing a set of value constructors for that type.

The first charactor of type constructors must be in lower-case and that of value constructors in upper-case. But conventionally, type constructors are in lower-case and value constructors in upper-case.

```
# type color = Red | Blue | Green ;;
type color = Red | Blue | Green
# type human = man | woman ;;
Characters 17-18:
Syntax error:
```

The above defines a new type `color` whose elements are `Red`, `Blue` and `Green`. This example is similar to the enumeration type in C and PASCAL.

```
# type human = MAN of string * int | WOMAN of string * int ;;
type human = MAN of string * int | WOMAN of string * int
```

The above defines a new type `human` whose elements are constructed as

[1] This is in some sense the same as `typedef` in C.

```
# MAN("dugbo", 12 ) ;;
- :  human = (MAN ("dugbo", 12))
# WOMAN("suni", 10 ) ;;
- :  human = WOMAN (("suni", 10))
```

Infinite number of `human` values can be constructed using the constructors `MAN` and `WOMAN`.

Recursively-defined datatype is very useful.

```
# type link = BOT | NODE of int * link ;;
type link = BOT | NODE of int * link
```

Some values are

```
# NODE(1 , BOT) ;;
- :  link = (NODE (1, BOT))
# NODE(1 , NODE(2,BOT)) ;;
- :  link = (NODE (1, NODE (2, BOT)))
```

As another example,

```
# type tree = LEAF of int | NODE of tree * tree ;;
type tree = LEAF of int | NODE of tree * tree
```

Or, more generally using a type variable

```
# type 'a tree = LEAF of 'a | NODE of 'a tree * 'a tree ;;
type 'a tree = LEAF of 'a | NODE of 'a tree * 'a tree
```

In this case we can construct various `tree`'s depending on a type bound to the type variable `'a`. For example,

```
# LEAF 3 ;;
- :  int tree = (LEAF 3)
# LEAF "a" ;;
- :  string tree = (LEAF "a")
```

When we define a function whose argument has a user-defined datatype, we can easily express the function body using *patterns*.

Before we talk about patterns in nML, let us first present how to name a value. In PASCAL and C, a name for a value is done by declaring a variable and by assigning a value to the variable. In nML, this is done a bit simpler by using the keyword `val`:

`val` *id* `=` *e*

5 Lexical Rule for Names

In nML, identifiers are for values, types constructors, data contructors, record fields, and pattern variables (see next section for patterns).

The first character of both variable id(*varid*) and type id (*tyid*)must be in lower-case. The first character of data constrctor id (*conid*), structure id(*strid*), signature id(*sigid*), functor id($fctid$) must be in upper-case. Structure, singature functor would be explained later section.

Type variable must start with a single print `'`: e.g., `'a`.

Here are the definitions :

alphanum	::=	a − z | A − Z | *hangul*|0 − 9|_|'
upper	::=	A − Z | _
lower	::=	a − z | *hangul*
hangul	::=	syllables of KSX1001 (a.k.a. KSC5601 or eur-kr)
	|	syllables of KSX1005-1 (a.k.a. KSC5700, unicode, or ISO/IEC10646-1)
sym	::=	! | % | & | $ | # | + | - | / | : | < | = | > | ? | @ | \ | ~ | ‘ | ^ | | | *
lid	::=	*lower*(*alphanum*)*
uid	::=	*upper*(*alphanum*)*
varid	::=	*lid*
prefixid	::=	(! | ? | ~)*sym**
infixid	::=	(% | & | $ | # | + | - | / | : | < | = | > | @ | \ | ‘ | ^ | | | *)*sym**
opid	::=	*prefixid* | *infixid*
tyid	::=	*lid*
conid	::=	*uid*
strid	::=	*uid*
sigid	::=	*uid*
fctid	::=	*uid*
tyvar	::=	' *lid*
lab	::=	(0 − 9)(0 − 9)* | *lid*
varlongid	::=	*varid* | *strid*. *varlongid*
oplongid	::=	*opid* | *strid*. *oplongid*
tylongid	::=	*tyid* | *strid*. *tylongid*
conlongid	::=	*conid* | *strid*. *conlongid*
strlongid	::=	*strid* | *strid*. *strlongid*

Value id and type constructor occupies different name space. This means, a common name can be used for a value and a type constructor. nML can determine whether a name is for a value or a type constructor. For example,

```
# val x = 1 ;;
val x :  int = 1
# type x = Int of int | Str of string ;;
type x = Int of int | Str of string
# x ;;
- :  int = 1
# val y = Int 2 ;;
val y :  x = (Int 2)
```

```
# val z = Str "nML" ;;
val z :  x = (Str "nML")
```

An alphanumeric identifier for a value consists of alphabets, numbers, underscore, and prime ', starting with an alphabet. An identifier is case-sensitive: e.g., `x` and `X` are different name.

Symbols that can't appear in the alphanumeric identifiers may be used to form an identifier. For example, identifiers composed only of the following symbols are also legitimate identifiers.

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

For example,

```
# val !!!  = 1;;
val !!!  :  int = 1
# val x = !!!  + 1;;
val x :  int = 2
```

Type and data constructors have the same lexical rule. Type variable must start with a single prime ': e.g., `'a`.

Following names are reserved. It's not allowed to redefine any of them:

```
and andalso array bool case char div do else end exception exn false
fn for fun functor handle if in int include land let list local lor
lsl lsr lxor mod nil not of open orelse raise real rec ref sig
signature string struct structure then true type unit val where while
Nil True False Match Zero Overflow Bound Equality
+ - * / ** ++ -- += -= *= /= << >> && || = <> < > <= >= :: @ ^
:= ! -> <- => | : ; { } [ ] [| |] , ( ) .
```

6 Naming Values: Binding, or Value Declaration

A value name is introduced by binding the name to a value as follows:

```
# val x = 1 + 2 ;;
val x :  int = 3
# val inc = fn x => x + 1 ;;
val inc :  int -> int = <fun>
```

It is important to note that nML binding is *not* assignment. In nML, when a value is bound to an identifier, there is no way to change the value that the identifier is bound. The same identifier may be rebound to a different value, though. Because multiple bindings for the same identifier is allowed, some convention about which binding to use must be provided. Here comes the notion of *scoping*.

nML binding is statically scoped. This means that whenever an identifier is used in an expression, it refers to the closest textually enclosing value binding for that identifier. For example,

```
# val x = 10 ;;
val x :  int = 10
# val y = x ;;
val y :  int = 10
# val x = true ;;
val x :  bool = true
# y ;;
- :  int = 10
```

After new binding for `x` happened, the value of `y` is not affected.

As another example,

```
# val x = 10 ;;
val x :  int = 10
# val f = fn y => x + y ;;
val f :  int -> int = <fun>
# val x = "ab" ;;
val x :  string = "ab"
# f 3 ;;
- :  int = 13
```

The value of `x` when a function value `f` is bound is that of `x`'s first binding. The later overshadowing binding does not affect the value inside the body of `f`.

Multiple identifiers may be bound simultaneously, using the keyword `and` as a separator:

```
# val x = 1 ;;
val x :  int = 1
# val x = true and y = x ;;
```

```
val x :  bool = true
val y :  int = 1
```

Multiple value bindings joined by `and` are evaluated in parallel. This is also used when we define mutually recursive functions.

Recursive function is declared as

```
# val rec f = fn x => if x = 0 then 1 else x * f x-1 ;;
val f :  int -> int = <fun>
```

A function binding is also done using the keyword `fun`.

`fun f(x) =` e

is same as

`val rec f = fn x =>` e

It is also useful to have local declarations whose role is to assist in the construction of some other declarations. This is accomplished using the keyword `local`, `in`, and `end`, as follows:

```
# local
    val x = 10
  in
    val k = ( x , x )
    val t = x + x
  end ;;
val k :  int * int = (10, 10)
val t :  int = 20
```

The binding for `x` is local to the bindings for `k` and `t`, that is, `x` is available during the evaluation of the bindings for `k` and `t`, but not thereafter.

We can also localize a declaration to an expression using `let`:

```
# let
    val x = 10
  in
    x+1
  end ;;
- :  int = 11
```

6.1 Use of and

정의할 때 and는 네 군데에서 쓰일 수 있습니다: 타입 정의 (type bind), 값 정의 (value bind), 모듈 정의 (structure bind), 그리고 예외상황 정의 (exception bind). (모듈타입에서 접속방안을 정의할 때 쓰이는 and는 이 네가지 경우를 본뜨게 된다.) 그 역할은 기본적으로 and로 묶인 복수의 정의들("정의 뭉치"라고 하자)은 서로의 이름을 모르는 상태에서 동시에 정의되는 것을 표현할 때 쓰인다.

단, 타입 정의에서는 서로 물고 무는 선언들로 여겨진다. 또하나, 기억할 것은, 값 정의에서 재귀함수의 정의들이 뭉치로 오면, "재귀"의 정의에 의해, 서로 물고 무는 선언들로 여겨진다.

예를들어,

```
type t = s and s = A | B
```

는 A와 B를 원소로하는 같은 타입 t와 s를 선언해 준다.

```
type t = A | Even of s and s = B | Odd of t
```

이면, Even(Odd(A))는 t타입이고, Odd(Even(B))는 s타입이다.

```
type t = A of s and s = B of t
```

이면, t나 s 타입의 값을 가지고 프로그램하면 그 프로그램은 영원히 돕니다. (왜일까요?)

```
val x = 1 val y = x and x = 10
```

이면 선언뭉치는 마지막 두개이고, y는 1, x는 10이 된다.

```
rec val even = fn x => not (odd x) and odd = fn x => not (even x)
```

혹은, 달콤하게,

```
fun even x = not (odd x) and odd x = not (even x)
```

은 서로 물고무는 정의가 된다.

7 Patterns

One cute and powerful notion in nML programming are patterns. When we bind a name to a value using

$$\texttt{val}\ lhs\ \texttt{=}\ e$$

the left-hand side lhs is actually a pattern that matches the result of e.

```
# val x = ( 10 , "kk" );;
val x :  int * string = (10, "kk")
# val (a,b) = x ;;
val a :  int = 10
val b :  string = "kk"
```

The left-hand side of the second value binding is a *pattern*, which is built up from pattern variables and constants using value constructors. The pattern variables will be bound to values by pattern matching.

```
# val x = ( 10 , true ) ;;
val x :  int * bool = (10, true)
# val (a,true) = x ;;
Warning:  this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(_, false)
val a :  int = 10
```

The left-hand side of the second value binding is a pattern involving constant `true`.

Notice that the simplest case of a pattern is a single pattern variable. This is the form of value binding that we introduced in the previous section.

Pattern matching may be performed against values of any of the types that we have introduced so far.

```
# val l = [1,2] ;;
val l :  int list = [1, 2]
# val [x,y] = l ;;
Warning:  this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
```

```
[]
val x :  int = 1
val y :  int = 2
```

Pattern match of datatype values are performed using the data constructors as follows:

```
# type tree = LEAF of int | NODE of tree * tree ;;
type tree = LEAF of int | NODE of tree * tree
# val x = NODE( LEAF(2) , LEAF(1) ) ;;
val x :  tree = (NODE (LEAF 2, LEAF 1))
# val NODE(LEAF a , b ) = x ;;
Warning:  this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
LEAF _
val a :  int = 2
val b :  tree = (LEAF 1)
```

Note that a pattern can be nested as above.
Note that `[1,2]` is an abbreviation of

```
1 ::  2 ::  nil
```

where the list data constructors are

```
::  :  'a * 'a list -> 'a list
nil :  'a list
```

Therefore,

```
# val l = [1,2] ;;
val l :  int list = [1, 2]
# val x::tail = l ;;
Warning:  this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val x :  int = 1
val tail :  int list = [2]
```

Patterns matching against record value is done on the basis of labeled fields.

```
# type info = {name :  string , age :  int} ;;
type info = {name:  string; age:  int}
# val r = {name ="Suni" , age = 5} ;;
val r :  info = {name="Suni", age=5}
# val {age = x , name = y} = r ;;
val x :  int = 5
val y :  string = "Suni"
```

It's sometimes convenient to use partial record pattern. This is done by record wild-card as follows:

```
# val {age = x , ...  } = r ;;
val x :  int = 5
```

Somtimes it is useful to bind "intermediate" pattern variables. For example, when a pattern `(x,y)` matches to a pair `(1,2)` we may want to match a single pattern variable for the pair as well as match `x` and `y` to each component. This is done by using the keyword `as`.

```
# val l as (x,y) = (1,2) ;;
val x :  int = 1
val y :  int = 2
val l :  int * int = (1, 2)
# val m = ( l , x ) ;;
val m :  (int * int) * int = ((1, 2), 1)
```

Wild-card pattern is _, that matches with any thing. We use this wild-card when some part of a value is not of our interest.

```
# fun f(x) = (x*3 , x+x+1) ;;
val f :  int -> int * int = <fun>
# val (x,_) = f(10) ;;
val x :  int = 30
```

The `(x,_)` pattern matches with any pair whose first component is of our interest.

8 Defining Functions

In Section 6where we showed a function declaration, a unnamed function was simply of the following form

$$\texttt{fn}\ lhs\ \texttt{=>}\ e$$

Function definition is accomplished either by using `val` or using `fun`. For example, the factorial function:

```
# val rec fac = fn x => if x = 0 then 1 else x * fac(x-1) ;;
val fac :  int -> int = <fun>
```

Or simply, by using `fun`:

```
# fun fac(x) = if x = 0 then 1 else x * fac(x-1) ;;
val fac :  int -> int = <fun>
```

In this section, more details follow.

A function is defined using patterns for the argument part.[2] Actually, in the above definitions of `fac`, the `x` in `fn x =>` and `fac(x)` is a pattern (called *pattern variable*).

See how we define a frontier function over the `tree` datatype:

```
# type tree = LEAF of int | NODE of tree * tree ;;
type tree = LEAF of int | NODE of tree * tree
# fun front(LEAF x) = [x]
  | front(NODE(left,right)) = (front left) @ (front right) ;;
val front :  tree -> int list = <fun>
```

Same function may be defined using `val` and `fn`:

```
# val rec front = fn LEAF(x) => [x]
  | NODE(left,right) => (front left) @ (front right) ;;
val front :  tree -> int list = <fun>
```

And see how the list append operation `@` is defined:

[2]This practice is more convenient than the conventional practice in C and PASCAL where `case` or (nested) `if` expression on the arguments are used.

```
# fun append( nil , l ) = l
  | append( hd::tl , l ) = hd ::  append(tl,l) ;;
val append :  'a list * 'a list -> 'a list = <fun>
```

List reversal is defined as follows:

```
# fun reverse l =
    let fun rev(nil,y) = y
      | rev(hd::tl,y) = rev(tl,hd::y)
    in
      rev(l,nil)
    end ;;
val reverse :  'a list -> 'a list = <fun>
```

9 Exceptions

A function may be undefined for some inputs. For example, consider the division function, which is undefined when the divisor is zero.

```
# 1 div 0 ;;
Uncaught exception:  Division_by_zero
```

The `div` is defined such that when the divisor is zero, it raises an exception named `Div`. In order to handle the raised exception gracefully, we use `handle` expression.

```
# (1 div 0 ) handle Division_by_zero => 99 ;;
- :  int = 99
```

nML's exception mechanism provides the means for a function to "give up" in a graceful and type-safe way whenever it is unable or unwilling to return a value in a certain situation.

In nML, exception values are treated just like any other value (until they are raised). They can be passed as function arguments, returned as the results of function applications, bound to identifiers, stored in locations and etc.

Exception is declared using the keyword `exception`.

```
# exception Impossible ;;
exception Impossible
```

An exception declaration is similar to `datatype` binding except that the type is always of type `exn`.

```
# Impossible ;;
- :  exn = (Impossible)
```

We may regard the exception `Impossible` as the constant constructor for `exn` datatype. Exception can have an argument of any type, as a data constructor does:

```
# exception Error of string ;;
exception Error of string
```

The exception `Error` is a data constructor of type `string -> exn`.

```
# Error("line 10") ;;
- :  exn = (Error("line 10"))
```

Exceptions can be declared locally inside `local` or `let` binding.
The function `raise :  exn -> 'a` raises an excpetion.

```
# exception Error of string ;;
exception Error of string
# fun f(x,k) = if x < 0 then raise k("negative") else x*x ;;
val f :  int * (string -> exn) -> int = <fun>
```

Once an exception is raised, a handler is located by the dynamic means: by going up the current evaluation chain to find potential handlers. During this process, one or more levels of the currently active call chain are aborted, up to the function containing the handler.

In nML, the syntax for an exception handler is:

$$e \texttt{ handle } p_1 \texttt{ => } e_1 \mid \cdots \mid p_n \texttt{ => } e_2$$

where the types of e_i's should be equal to the type of e. Patterns p_i's are compared with raised exceptions from the subcomputation of e. When the exception matches with pattern p_k, the corresponding expression e_k is evaluated. If the match fails, the raised exception continues to propagate back along the evaluation chain until it meets another handler, and so on.

Use of the exception facilities is not necessarily limited to deal with errors. The programmer can use exceptions as a "control diverter" to escape any

control structure to a point where the corresponding exception is handled. Also, using the exceptions, the programmer can tailor an operation's results or effects to particular purposes in a wider variety of contexts than would otherwise be the case.

10 Other Expression Constructs

- Conditional expression `if` e_1 `then` e_2 `else` e_3. The `else` is not optional.
- Case expression

```
case e1 of p1 => e1
   | p2 => e2
   | p3 => e3
...
```

  The `if` expression is actually an abbreviation of

```
case e1 of true => e2
   | false => e3
```

- Expression sequence

```
(e1; e2; ···; en)
```

- While expression

```
while e1 do e2
```

  This expression is equivalent to

```
let val rec var = fn () =>
  if e1 then (e2; var()) else ()
in var() end
```

11 Examples

Consider the following quick sort function

```
fun quick [] = []
  | quick (a::rest) =
      let fun split(left,right,[]) = quick left @ (a :: quick right)
            | split(left,right,x::l) =
                if x <= (a:int) then split(x::left, right, l)
                          else split(left, x::right, l)
      in
        split([],[],rest)
      end
```

The `quick` sorts the list of integers.

```
# quick [2,1,4] ;;
- :  int list = [1, 2, 4]
```

We can generalize the function so that it gets a comparison function and returns a quick sort function.

```
fun quick(le) =
   let fun sort [] = []
          | sort (a::rest) =
              let fun split(left,right,[]) = sort left @ (a::sort right)
                    | split(left, right, x::l) =
                        if le(x,a) then split(x::left, right, l)
                                   else split(left, x::right, l)
              in split([],[],rest)
             end
    in sort
    end ;;
```

For example,

```
# val intquick = quick( fn(x,y:int) => x <= y ) ;;
val intquick :  int list -> int list = <fun>
# intquick [4,1,2] ;;
- :  int list = [1, 2, 4]
```

```
# val pairquick = quick (fn ((x:int,y),(a,b)) => x <= a);;
val pairquick :  (int * '_a) list -> (int * '_a) list = <fun>
# pairquick [(2,"a") , (0,"b")] ;;
- :  (int * string) list = [(0, "b"), (2, "a")]
```

Exercise 1 Implement an interpreter of the following program. The input to the interpreter is a value of the datatype `stm`. See example value `prog`.

```
type id = string

type binop = PLUS | MINUS | TIMES | DIV

type stm = SEQ of stm list
             | ASSIGN of id * exp
             | PRINT of exp list

     and exp = VAR of id
             | CONST of int
             | BINOP of exp * binop * exp
             | IF of exp * exp * exp

val prog =
SEQ[ASSIGN("a",BINOP(CONST 5, PLUS, CONST 3)),
    ASSIGN("b",IF(BINOP(VAR"a",MINUS,CONST 1),
                  BINOP(CONST 10, TIMES, VAR"a"),
                  BINOP(CONST 20, DIV, VAR"a"))),
    PRINT[VAR "b"]]
```

The semantics is as follows. `SEQ` has a list of statement each of which is evaluated from left to right. `ASSIGN("x",e)` assigns a value of `e` to identifier `x`. `PRINT` prints out the values of expressions in the list from left to right separated by comma ending with newline. Expression `VAR "x"` is the value of `x`. `BINOP` applies the binary operator to the operands. `IF` is usual conditional expression, where depending on whether the first expression's value is zero or not it evaluates the second expression (in case of non-zero) or the third expression.

Exercise 2 Write a program that performs symbolic differentiation of algebraic expressions. For example, if the arguments to the program are

$ax^2 + bx + c$ and x, the program should return $2ax + b$. Differentiation of any such expression can be performed by applying the following reduction rule:

$$
\begin{aligned}
&dc/dx = 0 \quad \text{for } c \text{ a constant or a variable different from } x \\
&dx/dx = 1 \\
&d(u+v)/dx = du/dx + dv/dx \\
&d(uv)/dx = u(dv/dx) + v(du/dx)
\end{aligned}
$$

The input algebraic expression is a value of datatype `ae`.

```
type ae = CONST of int
        | VAR of string
        | POWER of string * int
        | TIMES of ae list
        | SUM of ae list
```

For example, $ax^2 + bx + c$ is

```
SUM [TIMES [VAR "a", POWER("x",2)],
     TIMES [VAR "b", VAR "x"],
     VAR"c"]
```

Your program `diff` should be of type `ae * string -> ae`.

Exercise 3 Queue can be implemented for almost-constant time complexity for both the add and delete operations.

Types of each operations are:

```
empty:  queue
add:  queue * elmt -> queue
delete:  queue -> elmt * queue
```

A queue $[a_1, \cdots, a_m, b_1, \cdots, b_n]$ (b_n is the head) is represented as a pair of two lists L and R:

$$L = [a_1, \cdots, a_m], \quad R = [b_n, \cdots, b_1].$$

After adding an element `x`, new queue becomes

$$[x, a_1, \cdots, a_m], [b_n, \cdots, b_1].$$

After deleting an element, new queue becomes

$$[a_1, \cdots, a_m], [b_{n-1}, \cdots, b_1].$$

When deleting, we sometimes have to reverse the L list and let it be the R list. Empty queue is $([], [])$.

Program such queue.

Exercise 4 We are developing an nML programming environment as follows. Inside an editor when the user moves the cursor to a position in a program source then the system response with the type of the expression that "most tightly" encloses the cursor position.

The environment has a table that maps a pair of integers $\langle m, n \rangle$ to a string s. Each integer pair is the range of character positions that each expression of the program occupies. The output string is the type for that expression.

One important property: there exist no two ranges that partially overlap. Every two ranges are either disjoint or one being completely enclosed by the other.

We need to implement such table and write a function whose input is a current cursor position and the ouput is the type of the expression whose range is the smallest among those that enclose the input position.

You must assume that the table is big, having more than 100K entries. The table's space/time complexity should be carefully optimized. Your function's average-case space/time complexity must be analyzed.

Exercise 5 Maze-generation. Suppose we have $n \times n$ identical square cells tiling a paper. A maze is defined to have one-entry, one-exit, and only one path from the entry to the exit. The goal of maze generation is to make it as hard to find its path as possible. Followings are reasonable rules for making a "difficult" maze:

- The larger the number of open walls the harder. (Well, not to the bird's eye view, but to the poor mice).
- The longer the wrong paths the harder.
- The longer the solution path the harder.

Devise a good maze generation algorithm and implement it in nML.

12 Imperative Feature

nML supports references and assignments without violating type-safety. A reference value has the type τ `type`, which is the type of references to values of type τ.

The function `ref: 'a -> 'a ref` allocates a space in the memory for the value passed as argument, and returns a reference to that location.

```
# val x = ref 1 ;;
val x :  int ref = ref 1
```

Note that it is impossible to have a reference without its initial value. The function `!: 'a ref -> 'a` returns the content of a location.

```
# val a = !x ;;
val a :  int = 1
```

The function `:= : 'a ref * 'a -> unit` is the assignment function.

```
# x := 2 ;;
- :  unit = ()
# !x ;;
- :  int = 2
```

As another example,

```
# val x = ref [1] ;;
val x :  int list ref = ref [1]
# x := [2] ;;
- :  unit = ()
# x := !x @ [3,4] ;;
- :  unit = ()
# !x ;;
- :  int list = [2, 3, 4]
```

See what nML says when

```
# val x = ref [] ;; val x :  '_a list ref = ref []
```

This is because nML doesn't know the type of list ref. So the result type is `'a list ref`. If you want to make int list ref, you have to annotate non-polymorphic type to the empty list:

```
# val x = ref ( []:int list ) ;;
val x :  int list ref = ref []
```

13 Environment

In a program, same name can be used for different things. The concept of environment is introduced to explain how to resolve this problem.

Recall that in nML we can name values (using `val` and `fun`), types (using `type`), data constructors (using `type`), record fields, and patterns. (For the module system, we can also name environments called structures, structure types called signatures, and parameterized environment called functors.[3] We will present these things shortly.)

An environment is a sequence of pairs where a pair has name and its binding object (value, type, data constructor, and etc). When a name occurs in an expression its binding object is referred to by looking up the current environment. In the class I will draw diagrams to show how the environment changes according to declarations.

In nML, the environment of a name is determined statically. This means that a name's binding is the closest textually enclosing binding for that name.

```
let val x = 10
in
 (let val x = (1,2)
  in x
  end,
 x)
end
```

is `((1,2),10)`.

```
let val x = 10
   fun f y = x + y + 1
in
  let val x = -10
  in (f 1 , f x )
  end
end ;;
```

is `(12, 1)`.

As in the value bindings, `type` bindings changes the environment.

[3] Don't be scared; structures, signatures, and functors are conceptually very simple, maybe simpler than `.h` and `.c` files in C.

```
let
  type t = BLUE | YELLOW
  fun blue() = BLUE
in
  let
   type t = BLUE | RED | PURPLE | GREEN | YELLOW
   fun mix(BLUE, RED) = PURPLE
     | mix(BLUE, YELLOW) = GREEN
     | mix(_,_) = YELLOW
  in
   mix(blue(),RED)
  end
end
```

In the above example, `mix(blue(),RED)` has type error. Because the `blue()` is of type `t` of the outer declaration, while `mix` requires a pair of values that have the inner `t` type.

14 The Modules System

In order for a language to be suitable for large-scale, serious programming (i.e., implementing software that will last generations, not a weekend, and will therefore have to be robust and maintainable), it must provide security and facilities for organizing relatively independent modules with well-defined interfaces.[4]

In nML, modular partitioning of programs is accomplished with `structure`s. A structure has a collection of localized declarations, thus defines an environment. Recall that an environment is the repository of the meanings of the names that have been declared in a program. In nML, program modularization is to partition the environment into chunks (structures) that can be manipulated relatively independently of one another.

Just as the type of a name hints the name's use in a program, so structures have a form of type called `signature` that describes the structures to the rest of the world.

[4]In C programming, a module is roughly a pair of `.h` and `.c` files. The header (`.h`) files are `#include`d when we need to access its procedures and variables inside another `.c` file. This mechanism is extremely flexible, but, unfortunately, is not secure.

A structure can also be parameterized. This is accomplished by defining a `functor` that maps structures to structures. Applying a functor to argument structures generates a new structure, as dictated in the body of the functor.

15 Structure and Signatures

A structure is an environment turned into a manipulatable object.

A structure (enclosed by `struct` and `end`) is named using the keyword `structure`.

```
# structure S =
 struct
   type t = int
   val x = 10
   fun f x = if x = 0 then 1 else x * f(x-1)
 end ;;
module S : sig type t = int val x :  int val f :  int -> int end
```

After this structure binding, nML records that structure `S` has type `t` that is `int`, value `x` that is integer 10, and function `f` that computes factorial.

Structure declarations can be nested, as environments are nested by nested `let` bindings.

If we want to use `x` and `f` outside the structure `S`, they must be prefixed with the structure name:

```
# val x = "ab" ;;
val x :  string = "ab"
# S.x ;;
- :  int = 10
# x ;;
- :  string = "ab"
```

We may `open` structures in order to use the identifiers without the structure prefix:

```
# open S ;;
# f 10 ;;
- :  int = 6
```

Just as every value has a type, so structures have types as well, namely signatures.

A signature (enclosed by `sig` and `end`) can have names using the keyword `signature`:

```
# signature S = sig val x :  int end ;;
```

This signature binding can only be done at the top-level, not, for example, within a structure declaration. Signature binding can't be nested either.

Using signatures you can trim the view of a structure.

```
structure FOO : S =
struct
  val x = 10
  val y = 1
end ;;
```

Because of the signature `S`, only `FOO.x` can be accessed from outside. If a structure declaration has no signature specification, then nML derives the "maximal" signature (type) for that structure.

It is important to note that signature specification for a structure is valid only when the structure has "more meats" than the signature specifies. For example, following is wrong

```
structure FOO : sig type t val x :  int end =
struct
  val x = 10
end
```

because the structure body has nothing about the type `t`. As another example,

```
structure FOO : sig val f :  'a list -> 'a end =
struct
  fun f [] = 0
    | f (x::rest) = x
end
```

fails because the `f` has type `int list -> int`, not as general as the polymorphic specification.

Therefore, note that many structures match with a signature. For example, with the following signature

```
signature SIG = sig val select :  int list -> int end
```

any structure that have a function named `select` of type `int list -> int` can match.

16 Functor: Function on Structures

Functor is a function that takes structures to structures. There are two uses of functors in nML programming.

- Functor facilitates the code re-use. Therefore a common code is implemented by a functor, and its particular instantiations are generated by applying the functor with appropriate arguments.
- Functor is used for managing the dynamics of software development in nML. Functors are glues that assembles a program from its component parts (structures). In this sense, functors plays the role of the linker in conventional programming system.

A functor definition is as follows:

`functor` f `(`s_1 `:` g_1`, ⋯, `s_n `:` g_n`) :` h `= struct` *body* `end`

It indicates that functor f takes structures $s_1, \cdots, s_n$ that respectively match with signatures $g_1, \cdots, g_n$ and returns a new structure as specified in the structure *body*. The type h of the result structure is optionally specified.

For example,

```
signature SIG =
  sig
    type t
    val eq : t * t -> bool
  end

functor F(P: SIG) =
  struct
    type t = P.t * P.t
    fun eq((x,y),(x',y')) = P.eq(x,x') andalso P.eq(y,y')
  end
```

We can generate various structures applying the functor `F` to different structures that can match with the argument signature `SIG`.

```
structure IntEq = F( struct type t = int
                     fun eq(x,y:int) = x=y end )

structure LstEq = F( struct type t = int list
                     fun eq(x::r,x'::r') = (x:int)=x' andalso eq(r,r') end )
```

16.1 Code Re-use by Functors

We implement a common code as a functor and apply the functor to implement other parts that uses the common part.

Suppose we must implement a code for managing base sets (e.g., integer sets) and product sets of other sets.

We define a functor that generates a set structure:

```
functor SetFun (S: sig type t val equal: t * t -> bool end) : SETSIG =
  struct
   type t = S.t
   type set = t list

   val equal = S.equal
   fun member(x, nil) = false
     | member(x, a::rest) = if equal(x,a) then true else member(x,rest)
   fun add(x,s) = if member(x,s) then s else x::s
  end
```

where the signature `SETSIG` is:

```
 signature SETSIG =
   sig
    type t
    type set
```

```
    val equal : t * t -> bool
    val member : t * set -> bool
    val add : t * set -> set
  end
```

We can define a functor for generating a structure for the product set:

```
functor ProductSetFun (A : SETSIG ,
                       B : SETSIG) : SETSIG =
  struct
    type t = A.t * B.t
    type set = t list
    fun equal((a,b),(a',b')) = A.equal(a,a') andalso B.equal(b,b')
    fun member(x,nil) = false
      | member(x, a::rest) = if equal(x,a) then true
                             else member(x,rest)
    fun add(x,s) = if member(x,s) then s else x::s
  end
```

Now, we use these functors to generates various sets.

```
structure IntSet = SetFun(struct
                          type t = int
                          fun equal(x,y:t) = x=y
                         end)

structure IntStringSet = SetFun(struct
                                type t = int * string
                                fun equal((x,y),(x',y'):t) = x=x'
                               end)

structure Pset = ProductSetFun( IntSet ,
                                IntStringSet)

structure Pset' = ProductSetFun( Pset ,
                                 IntSet )
```

Exercise 6 Implement an interpreter of the following language. The input to the interpreter is a value of the datatype `exp`. See example value `prog`.

```
type id = string
type binop = PLUS | MINUS | TIMES | DIV
type exp = INT of int
             | VAR of id
             | READ of exp
             | WRITE of exp * exp
             | IF of exp * exp * exp
             | ARRAY of id * exp list
             | WHILE of exp * exp
             | BINOP of binop * exp * exp
             | SEQ of exp list
             | PRINT of exp list

val prog =
SEQ[WRITE(VAR("a"), BINOP(PLUS, INT 5, INT 3)),
    WHILE(READ(VAR("a")),
          SEQ[WRITE(ARRAY("x", [READ(VAR("a"))]), INT 0),
              WRITE(VAR("a"),
                    BINOP(MINUS, READ(VAR("a")), INT 1))]),
    READ(ARRAY("x", [INT 3]))]
```

The semantics is as follows. An expression evaluation can lead to either an integer, a location, or undefined. `INT(`x`)` is integer x. `VAR(`x`)` is a location bound to x. `READ(`e`)` is the value stored at location e. `WRITE(`e`,` e'`)` writes a value e' to location e, returning undefined value. `IF(`e`,` e'`,` e''`)` evaluates e. If it is zero, evaluate e'', otherwise evaluate e'. `ARRAY(`x`,` e`)` is location of array $x[e]$. `WHILE(`e`,` e'`)` keeps evaluating e' while e's value is non zero. `WHILE` expression's value is undefined. `BINOP(`op`,` e`,` e'`)` applies the binary operation op to the values of e and e'. `SEQ(`e`,` e'`)` sequentially evaluates the expressions from left to right and the last expression's value is the value of the `SEQ` expression. `PRINT(`e`,` e'`)` prints out the values of expressions in the list from left to right separated by comma, ending with newline.

Exercise 7 Solve the following marriage problem in a fully functorized fashion.

There are two sets of males and females. The sizes of the two sets need not be the same. Our marriage problem is to find the set of married pairs (polygamy possible) whose total "happiness" is maximal. The input to this problem is two sets of preference lists: each male ranks his preference for females, and vice versa. Depending on how the happiness is defined, the problem becomes NP-Complete or can be deterministically solved in a polynomial time. Our definition of the happiness is defined as follows:

- total-happiness(pairs) = sum of each pair's happiness
- pair-happiness(`m`,`f`) = `m`'s rank in `f`'s preference list + `f`'s rank in `m`'s preference list

This happiness measure allows a polynomial time solution. However, don't try to use the optimal solution. Rather, use the following simple algorithm.

Suppose there are n males and m females. We compute the happiness of $n \times m$ pairs. At each iteration, we choose the happiest pair from the set of possible couples. Initially, the possible couples are all the $n \times m$ pairs. After each iteration, the number of possible couples is reduced by not considering already-married persons.

This algorithm has one parameter. That is, what if there exist more than one happiest pair at an iteration? Your functor must be parameterized about this tie-break policy.

Define functors such that the following top-level functor applications work.

```
structure D = Data(...)
structure T = TieBreak( D )
structure Main = Marriage( D , T )

(* val Main.marry: string -> unit *)
```

Exercise 8 A *lattice* L is a set with a partial order such that any two elements a and b have their least upper bound ($a \sqcup b$) and the greatest lower bound ($a \sqcap b$) in L. A complete lattice is a lattice whose least (bottom) and greatest (top) elements exist.

Given a set S, its powerset with set-inclusion as its partial order and $\sqcup = \cup$ and $\sqcap = \cap$ is a complete lattice. (Bottom = $\{\}$, top = S.) For two

complete lattices L and L', their Cartesian product $L \times L'$ with componentwise partial order

$$\langle a, b\rangle \sqsubseteq \langle a', b'\rangle \iff a \sqsubseteq a' \text{ and } b \sqsubseteq b'$$

is also a complete lattice.

Define functors `PowerLatt` and `ProductLatt` that returns a powerset lattice from a set and a product lattice from two lattices, respectively. The two functors must satisfy the following signature specification.

When you define the functors, you may want to use some library structures for aggregate data (e.g., dictionary, integer sets, and etc.) See the OCaml Library Manual.

```
signature LATTICE =
 sig
   type elmt
   type 'a t = BOT | TOP | ELMT of 'a
   val join : elmt t * elmt t -> elmt t       (* least upper bound *)
   val meet : elmt t * elmt t -> elmt t       (* greatest lower bound *)
   val le : elmt t * elmt t -> bool           (* partial order LE *)
   val pp : elmt t * out_channel * int -> unit  (* pretty printer *)
 end

signature SET =
 sig
   type elmt
   type t = ELMT of elmt
   val size : unit -> int               (* set size *)
   val enlist : unit -> t list          (* list the set elements *)
   val ord : t -> int                   (* ord number of an element *)
   val pp : t * out_channel * int -> unit (* pretty printer *)
 end

functor PowerLatt(S: SET): LATTICE = struct ... end
functor ProductLatt(L: LATTICE ,
                    L': LATTICE): LATTICE = struct ... end
```