ML 演習
The Meta Language

児玉靖司

平成 13 年 8 月 22 日

もくじ

1

はじめに

1970 年代よりエジンバラ大学教授 R. Milner 博士を中心として開発された言語が，ML(Meta-Language) である．その後，いくつか改良され，いくつかのバージョンができた．

- エジンバラ大学による EDML
- Standard ML of New Jersey
- CAML(Caml, Caml-light, Caml-super-light など)

これらは，基本的な考え方は同じだが，開発者はまったく別である．この演習では，Standard ML(以下 SML という) を使う．SML は，

- 関数プログラミング言語
- 型推論
- コンパイル方式

という特徴を持つ．また，プログラムの意味を厳密に定義することができる (意味論的にクリア)．特に，型推論 (type inferences)

項 (プログラミング言語では式と同様) の型を宣言すること無しに，処理系が自動的に型を推論する

というものである．この機構は，最近ではいくつかの言語に備わっているが ML は，その中で最も有名なものといえるだろう．多くの ML の処理系は，初期の段階では LISP の上で構築されたが，EDML，SML は C 言語を用いて効率よいものとなっている．また，コンパイル方式でありながら，そのインターフェースはインタプリタ方式と同様に手軽にプログラミングすることができる．

さて，まず SML を起動しよう．

1.1 電卓のように

操作 1: 電卓のように

```
prompt% sml↩
Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit
- 10 + 3;↩
val it = 13 : int
- 5 div 2;↩
val it = 2 : int
- 11.2 + 13.1;↩
val it = 24.3 : real
- 24.3;↩
val it = 24.3 : real
- ~10.1 + 21.3;↩
val it = 11.2 : real
- ~ 10.3;↩
val it =  10.3 : real
- val test = "hello";↩
val test = "hello" : string
- val sample = test ^ " " ^ "guys!";↩
val sample = "hello guys!" : string
- size sample;↩
val it = 11 : int
- sample;↩
val it = "hello guys!" : string
```

上のように，プログラマブルの電卓のように数字を入力することにより計算を行なうことができる．SML の終了は，
CTRL+D をタイプする．ここで，

操作 2: 型推論

```
- 10 + 3;↩
val it = 13 : int
```

のように，入力は数字と演算子のみに対し，処理系が int 型 (整数型) であると推論している．この機構を **型推論** (type inference) と呼ぶ．マイナス記号は，チルド~(tilde) を使う．しかし，

操作 3: ~記号

```
- ~10.3;↩
val it = ~10.3 : real
- ~ 10.3;↩
val it = ~10.3 : real
```

では意味が違う．前者は，通常の -10.3 の意味だが，後者は，10.3 を引数として ~ という関数を呼び出すことを示す．つまり，~ という関数が存在する (後述).

演算子 ^ は，文字列を連結する．また，関数 size は文字列の長さを返す関数である．このように，関数呼び出しは，

関数名 引数

とするだけで実行することができる．

1.2 変数

値を識別するための名前を **変数** (variables) と呼ぶ．例えば，

操作 4: 変数

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

など x，y は変数である．しかし，更新する (代入する) ことはできない．この = という演算子は，‘代入’ ではなく ‘割当て’(binding) をする．そのため，この次に

操作 5: 割当て

```
- val x = 11;↩
val x = 11 : int
- x;↩
val it = 11 : int
```

などとすると新たな値となる[1]．またこの名前は，変数として再割当てされるとは限らない[2]．この名前は，以下定義した関数などで使うことができる．

1.3 関数定義とパターンマッチ

SML は，型を持つ言語である．つまり各**式** (expression) (項 (term) ともいう) には型という属性がついている．例えば，

型	意味
bool	論理型
int	整数型
real	実数型
string	文字列型

などがある．では，関数はどう表現するのであろう．関数も型の一種で，通常入力と出力があるので，

入力値の型 -> 出力値の型

[1] 正確な意味での変数ではないので，var ではなく val になっていることに注意しよう．
[2] 関数として定義することも可能．つまり，型が全く変わってしまうかもしれない．

のように，-> を使って表現する．例えば，上の関数 size は，

操作 6: size 関数

```
- size;↩
val it = fn : string -> int
```

より，文字列型 string を受渡され，整数型 int を返す関数であることがわかる．では，関数を定義してみよう．

操作 7: 関数

```
- fun twice x = x * 2;↩
val twice = fn : int -> int
- twice( 10 );↩
val it = 20 : int
- twice 10;↩
val it = 20 : int
- twice;↩
val it = fn : int -> int
```

関数定義には，fun を使う．関数 twice は，引数 x を受け渡されて，その 2 倍を計算し返す関数である．ここで，変数 x が，整数型であると推論されていることに注意しよう[3]．

引数は，通常 1 つである．複数の場合は，括弧 () をつけて書く．しかし，括弧をつけると別の 組 という型になる (後述)．ので正確には，引数は，常に 1 つである．次の例をみてみよう．

操作 8: 引数

```
- fun swap( x, y ) = ( y, x );↩
val swap = fn : 'a * 'b -> 'b * 'a
- swap( "test", 10.0 );↩
val it = (10.0,"test") : real * string
- fun decrement x = x - 1;↩
val decrement = fn : int -> int
- fun square x = x * x;↩
std_in:27.18 Error: overloaded variable cannot be resolved: *
- fun square ( x: int ) = x * x;↩
val square = fn : int -> int
- fun all_equal( x, y, z ) = ( x = y ) andalso ( y = z );↩
val all_equal = fn : ''a * ''a * ''a -> bool
- fun ordered( x, y, z ) = ( x < y ) andalso ( y < z );↩
std_in:29.48 Error: overloaded variable cannot be resolved: <
std_in:29.30 Error: overloaded variable cannot be resolved: <
```

まず，関数 swap の引数の型が，'a また，'b となっているが，この型は特定の型を示すのではなく，SML で表現できる型全て ($\forall\tau$ と表現する) を示す．そのような型を**多相型** (polymorphic type) と呼ぶ．関数 swap は，2 つの引数を渡され，その型は SML で表現できる型のどんな型でも実行可能であることを示す．

[3]つまり，整数 2 が整数型で，演算子 * は，

同じ型の値を 2 つ受け渡されて，その型を返す．

という関数なので，整数型と推論される．

関数 decrement は，簡単に定義できるが，関数 square の定義では，失敗している．これは，x の型が特定できないので，多相型としたいところであるが，演算子 * が，全ての型には対応できないため，エラーとなっている．このような場合には，明示的に型 (int) などを指定する必要がある[4]．

多相型の 1 種として，''a，''b のようにアポストロフィ (') が 2 つついた型がある．これも多相型であるが，型の中でも = (等しい) という評価ができる型を示す．しかし，他の > や，< を評価することができる型を示した多相型は表現できないので関数 ordered の定義は失敗する．

関数 swap の定義で引数の型が，'a * 'b となっている．このような時，'a と 'b の組 (tuple) と呼ぶ．組は，括弧 (()) を使って表現する．この他に，角括弧 [] でくくるリスト (list) がある．

論理 (logic) に関する演算子は，以下のものがある．

not	否定
andalso	論理積
orelse	論理和
if ··· then ··· else ···	if 式

関数定義の中で，条件判断をする場合 if ··· then ··· else ··· を使う場合と **パターンマッチ** (pattern match) を使う場合がある．

操作 9: パターンマッチ

```
- fun not x = if x = true then false↩
=                          else true;↩
val not = fn : bool -> bool
- fun p_not true = false↩
=   | p_not false = true;↩
val p_not = fn : bool -> bool
- not true↩
val it = false : bool
- p_not false;↩
val it = true : bool
```

共に，'否定' を表す関数を定義する．if ··· then ··· else ··· の then の後と else の後は同じ型でなければならないことに注意しよう．

1.4 リスト

リスト (list) は，角括弧 ([]) を使って書く．

[4]SML では，このように関数の引数の型としてのみ型指定が必要な場合がある．他の ML の処理系では，この指定すら必要としないものもある．しかし，その場合は，演算子 * の引数が，整数型 (int 型) と限定している．

操作 10: リスト

```
- [1,2];↩
val it = [1,2] : int list
- [[1],[2,3,4]];↩
val it = [[1],[2,3,4]] : int list list
- val alist = "a" :: nil;↩
val alist = ["a"] : string list
- val alist = ["a"];↩
val alist = ["a"] : string list
- val alist = ["c","b","a"];↩
val alist = ["c","b","a"] : string list
- val blist = "c" :: "b" :: alist;↩
val blist = ["c","b","c","b","a"] : string list
- val front :: rest = [3,2,1];↩
std_in:51.1-51.27 Warning: binding not exhaustive
           front :: rest = ...
val front = 3 : int
val rest = [2,1] : int list
```

リストのリストも書くことができる．しかし，一つのリストの要素は同じ型でなければならない．演算子 :: (コンス, cons) により，要素とリストを連結する．前項が要素であることに注意しよう．他に，リストとリストを連結する演算子として @(アペンド, append) がある (後述)．また，このコンス演算子 (::) は，演算子 = の左に書くこともできる．

では，リストの要素を逆順にする関数 reverse を考える．

プログラム 1: reverse 関数

```
- reverse [23,45,32,43];↩
val it = [43,32,45,23] : int list
- reverse ["aa","bb","cc","dd"];↩
val it = ["dd","cc","bb","aa"] : string list
```

このように，多相型'a のリストを受け付けることができるようにする．プログラムは，

操作 11: プログラム入力

```
- fun reverse nil = nil↩
= | reverse ( front :: rest ) = ( reverse rest ) @[front];↩
val reverse = fn : 'a list -> 'a list
```

このプログラムの実行の様子を表現すると，

```
reverse [23,45,32,43];
reverse [45,32,43] @ [23];
( reverse [32,43] @ [45] ) @ [23];
( ( reverse [43] @ [32] ) @ [45] ) @ [23];
( ( ( nil @ [43] ) @ [32] ) @ [45] ) @ [23];
```

となり，結果として [43,32,45,23] を返す．

このように関数を定義していくが，SML を終了すると，それまでの定義は，消えてしまう．そのため，このような関数定義は，ファイルに保存しておき演算子 use を使う．たとえば，このプログラムをエディタ (mule など) を使って，reverse.sml と保存したとすると，

操作 12: プログラムの実行

```
- use "reverse.sml";↩
[opening reverse.sml]
val reverse = fn : 'a list -> 'a list
val it = () : unit
- reverse [1,2,3];↩
val it = [3,2,1] : int list
```

となる．最後に，リストに関するいくつかの関数を紹介する．

操作 13: リストに関する関数

```
- nil;↩
val it = [] : 'a list
- null;↩
val it = fn : 'a list -> bool
- null [1,2,3];↩
val it = false : bool
- null [];↩
val it = true : bool
- null nil;↩
val it = true : bool
- hd;↩
val it = fn : 'a list -> 'a
- hd [3,4,5];↩
val it = 3 : int
- tl;↩
val it = fn : 'a list -> 'a list
- tl [5,6,7];↩
val it = [6,7] : int list
- (op ::);↩
val it = fn : 'a * 'a list -> 'a list
- (op @);↩
val it = fn : 'a list * 'a list -> 'a list
```

nil はあらゆる型のリストの空リストを示す．関数 null は，第 1 引数が nil の場合 true を返し，それ以外の場合 false を返す．関数 hd，tl は，それぞれ，リストの最初，最後の要素を返す (演算子 ::，@ については先述)．op は，中置記法で用いる演算子を関数呼び出しの形 ff(x, y) で用いる場合に使う．

1.5 中置演算子

中置演算子を定義するためには infix を使う．例えば，リストを配列のように整数で取り出すための演算子!を定義する．

操作 14: 中置演算子!

```
- exception infix_operation;↩
- infix !;↩
infix !
- fun nil ! 0 = raise infix_operation↩
- |   (x::xs) ! 0 = x↩
= |   (x::xs) ! n = xs ! (n-1);↩
std_in:60.1-61.28 Warning: match nonexhaustive
(nil,0) => ...
(x :: xs,0) => ...
(x :: xs,n) => ...
val ! = fn : 'a list * int -> 'a
- ["a","b","c"]!2;↩
val it = "c" : string
```

1.6 カリー化と高階関数

一般に，ff(x, y) で表す関数呼び出しを ff x y のように変換することを **カリー化** (currying) と呼ぶ．通常カリー化した形で書いたほうが都合がよいのでこのように書く．ML ではそのような呼び出しもできる．

操作 15: カリー化

```
- val inc = 1 +;↩
std_in:77.14 Error: nonfix identifier required
std_in:77.1-77.13 Error: operator is not a function
  operator: int
  in expression:
    1 + : overloaded
std_in:77.13 Error: overloaded variable cannot be resolved: +
- (op+)(1,2);↩
val it = 3 : int
- fun plus (x:int)(y:int) = x + y:int;↩
val plus = fn : int -> int -> int
- val inc = plus 1;↩
val inc = fn : int -> int
- inc 2;↩
val it = 3 : int
- inc 10;↩
val it = 11 : int
```

ここで，関数 plus の型は，int -> int -> int，関数 inc の型は，int -> int であることに注意しよう．カリー化された関数は，引数の個数だけ -> となる関数となる．

1.6.1 関数 curry

では，より複雑な関数を表現するためカリー化を行なう関数を定義してみよう．

操作 16: curry 関数

```
- fun curry ff x y = ff ( x, y );↩
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

この関数により,

操作 17: curry 関数を使う

```
- max;↩
val it = fn : int * int -> int
- max( 2, 3 );↩
val it = 3 : int
- curry max 2 3;↩
val it = 3 : int
```

関数 max のように，通常 ff(x, y); と呼び出す関数をカリー化した形で書くことができる．これにより,

操作 18: 関数のカリー化

```
- val inc = curry op+ 1;↩
val inc = fn : int -> int
- val twice = curry op* 2;↩
val twice = fn : int -> int
- val plus = curry (op +): int -> int -> int;↩
val plus = fn : int -> int -> int
- val real_plus = curry (op +): real -> real -> real;↩
val real_plus = fn : real -> real -> real
- val times = curry (op * ): int -> int -> int;↩
val times = fn : int -> int -> int
- val real_times = curry (op * ): real -> real -> real;↩
val real_times = fn : real -> real -> real
- val isitgreaterthan = curry (op <): int -> int -> bool;↩
val isitgreaterthan = fn : int -> int -> bool
- val greaterthan = curry (op >): int -> int -> bool;↩
val greaterthan = fn : int -> int -> bool
- val str_lessthan = curry (op <): string -> string -> bool;↩
val str_lessthan = fn : string -> string -> bool
- val concat = curry op^;↩
val concat = fn : string -> string -> string
- val cons = curry op::;↩
val cons = fn : 'a -> 'a list -> 'a list
- val equal = curry op=;↩
val equal = fn : ''a -> ''a -> bool
- val notequal = curry op<>;↩
val notequal = fn : ''a -> ''a -> bool
```

ここで，*) は，注釈の終わりを示す記号なので，*) のように空白を空けなければならない.

1.6.2 部分的に適用した関数

次に，以下の関数を考える．

プログラム 2: 関数 Get_nth

```
exception Get_nth
fun get_nth _ nil = raise Get_nth
|   get_nth 1 (front :: _) = front
|   get_nth n (_ :: rest ) = get_nth (n-1) rest;
```

この関数 get_nth は，

操作 19: 関数 get_nth

```
promptStandard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit
- use "get_nth.sml";↩
[opening get_nth.sml]
exception Get_nth
val get_nth = fn : int -> 'a list -> 'a
val it = () : unit
- get_nth 3 ["a","b","c","d","e"];↩
val it = "c" : string
```

のように使う．第 2 引数であるリストの中から，第 1 引数で示す要素を返す．関数定義中のアンダスコア (_) は，パターンマッチの際，‘すべてにマッチする’ を意味する．関数 get_nth の型は，int -> 'a list -> 'a となっていることに注意しよう．そのため，

操作 20: get_nth 関数を使う

```
- val get_third = get_nth 3;↩
val get_third = fn : 'a list -> 'a
- get_third [1,2,3,4,5];↩
val it = 3 : int
```

のように，get_nth 3 を別の名前 (関数) として定義することができる．

1.6.3 合成関数

次のように定義することもできる．

操作 21: 合成関数

```
- fun twice x = x * 2;↩
val twice = fn : int -> int
- fun quad x = twice(twice x);↩
val quad = fn : int -> int
- fun many x = twice(twice(twice(twice x)));↩
val many = fn : int -> int
- quad 3;↩
val it = 12 : int
- many 4;↩
val it = 64 : int
```

このような場合は，合成関数として定義することもできる．

操作 22: 合成関数を使う

```
- val many = twice o twice o twice o twice;↩
val many = fn : int -> int
- many 4;↩
val it = 64 : int
```

合成関数は，演算子 o を使う．

1.6.4 関数の値

関数は，

操作 23: 関数の例

```
- fun twise x = x * 2;
val twise = fn : int -> int
- twise 2;
val it = 4 : int
```

のように fun を使って定義したが，値としても定義することができる．

操作 24: 関数の値

```
- val twise = fn x => x * 2;
val twise = fn : int -> int
- twise 3;
val it = 6 : int
```

この場合，右辺の fn x => x * 2; が **関数の値** (関数名がないことに注意) である．実際，引数として関数を指定する場合などには，値として関数を指定するのでこちらを使う．

1.6.5 繰り返し

高階関数 (higher order functions) を使って，繰り返しを行なうことを考える．

`fun repeat n ff state`

で，`n` は繰り返す回数，`ff` は呼び出す (以下，**適用** (application) という) 関数，`state` は，初期状態を表すとする．この関数により，N 個のドット `.` を表示する関数は，

操作 25: 繰り返し

```
- fun printdots n = repeat n (concat ".") "";↩
val printdots = fn : int -> string
- printdots 20;↩
val it = "...................." : string
```

ここで，関数 `concat` は，上で定義した関数である．最後に関数 `repeat` を定義してみよう．

操作 26: 関数 `repeat`

```
- fun repeat 0 f state = state↩
= |   repeat n f state = repeat (n-1) f (f state);↩
val repeat = fn : int -> ('a -> 'a) -> 'a -> 'a
```

関数 `repeat` を呼び出すたびに，関数 `f` が 1 度づつ適用されていることに注意しよう．このように，関数の引数として関数を渡すような関数を，高階関数と呼ぶ．

1.6.6 関数 `map`

典型的な高階関数として関数 `map` がある．

操作 27: `map` 関数

```
- map;↩
val it = fn : ('a -> 'b) -> 'a list -> 'b list
```

以下のように使います．

操作 28: 関数 `map` を使う

```
- val plus = curry (op +): int -> int -> int;↩
val plus = fn : int -> int -> int
- val inc = plus 1;↩
val inc = fn : int -> int
- inc 1;↩
val it = 2 : int
- map inc [73,82,34,54];↩
val it = [74,83,35,55] : int list
```

この操作を関数 `map` を使わないで定義すると，

操作 29: 関数 map と同等の関数

```
- fun map_inc nil = nil↩
= |   map_inc (front :: rest) = (inc front) :: (map_inc rest);↩
val map_inc = fn : int list -> int list
- map_inc [73,82,35,55];↩
val it = [74,83,36,56] : int list
```

となり，関数 map 上の定義をするための**構文糖** (syntax suger) ともいうことができる．

2

1 章のドリル

次の仕様に従った関数を定義せよ．すべて fun で始まる 1 つの関数として定義し，次章で扱う let ... in ... end 構文は用いずに定義せよ．レポートは，jLaTeX を使って清書し，プログラムの説明，考察をせよ．

2.1 関数 funpow

```
- funpow;↩
val it = fn : int -> ('a -> 'a) -> 'a -> 'a
- funpow 3 twice 2;↩
val it = 16 : int
```

第 2 引数以降の関数適用 ('a -> 'a) -> 'a を，第 1 引数で示す回数行なう．

2.2 関数 append

```
- append;↩
val it = fn : 'a list -> 'a list -> 'a list
- append [2,3,4] [3,4,5];↩
val it = [2,3,4,3,4,5] : int list
- append ["ae","ef","fh"] ["ef","fh","ll"];↩
val it = ["ae","ef","fh","ef","fh","ll"] : string list
```

リストの追加を行なう．

```
- (op @);↩
val it = fn : 'a list * 'a list -> 'a list
```

と似ているが型が違うことに注意しよう．@ を使わずに定義せよ．

2.3 関数 el

```
- el;↩
val it = fn : int -> 'a list -> 'a
- el 2 ["ef","gh","ef"];↩
val it = "gh" : string
- el 3 nil;↩

uncaught exception EL
```

```
- el ~3 [3,4,5];↩

uncaught exception EL
- el 0 [4,5,6];↩

uncaught exception EL
- el 5 [2,3,4];↩

uncaught exception EL
```

第 2 引数で示すリストの，第 1 引数番目の要素を取り出す.

$$\texttt{el}\ i\ [x_1, \cdots, x_n] = x_i$$

第 1 引数が第 2 引数で示すリストの適切な位置を示さない場合や，第 2 引数が nil の場合は，例外処理 expection, raise を行なう.

2.4 関数 last

```
- last;↩
val it = fn : 'a list -> 'a
- last [2,3,4];↩
val it = 4 : int
- last ["aa","ef","fg","lm"];↩
val it = "lm" : string
```

リストの最後の要素を返す．第 1 引数が nil の場合は，例外処理を行なう.

2.5 関数 butlast

```
- butlast;↩
val it = fn : 'a list -> 'a list
- butlast [3,4,5,6];↩
val it = [3,4,5] : int list
- butlast nil;↩

uncaught exception ButLast
- butlast ["ef","fg","gh"];↩
val it = ["ef","fg"] : string list
```

リストの最後の要素以外を含んだリストを返す．第 1 引数が nil の場合は，例外処理を行なう (関数 tl を使わずに定義せよ).

2.6 関数 replicate

```
- replicate;↩
val it = fn : 'a -> int -> 'a list
```

```
- replace "a" 10;↩
val it = ["a","a","a","a","a","a","a","a","a","a"]
: string list
- replace 1 5;↩
val it = [1,1,1,1,1] : int list
- replace true 0;↩
val it = [] : bool list
- replace false ~1;↩

uncaught exception Replicate
```

第 1 引数で示す要素を，第 2 引数個含んだ要素を返す．第 2 引数が負の数の場合例外処理を行なう．

2.7 関数 itlist

```
- itlist;↩
val it = fn : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
- curry max;↩
val it = fn : int -> int -> int
- itlist (curry max) [3,4,5] 2;↩
val it = 5 : int
- itlist (curry op+) [3,4,5] 2;↩
val it = 14 : int
- itlist (curry op::) ["a","b","c"] ["a"];↩
val it = ["a","b","c","a"] : string list
```

以下の仕様を満たす関数.

$$\begin{aligned} \texttt{itlist}\ f\ [x_1, x_2, \cdots, x_n]\ x &= f\ x_1\ (f\ x_2\ (\cdots (f\ x_n\ x)\cdots)) \\ &= ((f\ x_1)\ \texttt{o}\ (f\ x_2)\ \texttt{o}\cdots\texttt{o}\ (f\ x_n))\ x \end{aligned}$$

2.8 関数 rev_itlist

```
- rev_itlist;↩
val it = fn : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
- rev_itlist (curry op::) ["a","b","c"] ["a"];↩
val it = ["c","b","a","a"] : string list
- rev_itlist (curry op+) [3,4,5] 2;↩
val it = 14 : int
- rev_itlist (curry op-) [3,4,5] 2;↩
val it = 2 : int
```

以下の仕様を満たす関数.

$$\begin{aligned} \texttt{rev\_itlist}\ f\ [x_1, x_2, \cdots, x_n]\ x &= f\ x_n\ (f\ x_{n-1}\ (\cdots (f\ x_1\ x)\cdots)) \\ &= ((f\ x_n)\ \texttt{o}\ (f\ x_{n-1})\ \texttt{o}\cdots\texttt{o}\ (f\ x_1))\ x \end{aligned}$$

2.9 関数 end_itlist

```
- end_itlist;↩
val it = fn : ('a -> 'a -> 'a) -> 'a list -> 'a
- end_itlist (curry op-) [2,3,4];↩
val it = 3 : int
- end_itlist (curry op@) [["aa"],["bb"],["cc"]];↩
val it = ["aa","bb","cc"] : string list
- end_itlist (curry op@) nil;↩

uncaught exception END_ITLIST
```

以下の仕様を満たす関数.

$$\begin{aligned} \texttt{end\_itlist}\ f\ [x_1, x_2, \cdots, x_{n-1}, x_n] &= f\ x_1\ (f\ x_2\ (\cdots (f\ x_{n-1}\ x_n)\cdots)) \\ &= ((f\ x_1)\ \texttt{o}\ (f\ x_2)\ \texttt{o} \cdots \texttt{o}\ (f\ x_{n-1}))\ x_n \end{aligned}$$

リストの要素は，同じ型でなければならないので (curry op::) は，使えないことに注意しよう.

2.10 関数 first_itlist

以下の仕様を満たす関数.

$$\texttt{first\_itlist}\ f\ [x_1, x_2, \cdots, x_{n-1}, x_n] = f\ \cdots\ f((f\ x_1\ x_2)\ x_3)\ \cdots\ x_n$$

2.11 関数 map を再定義せよ.

2.12 関数 find

```
- find;↩
val it = fn : ('a -> bool) -> 'a list -> 'a
- fun pp x = if length x = 3 then true else false;↩
val pp = fn : 'a list -> bool
- find pp [["test","aa","bb","a"],["a"],["234"],["test","test1","test2"]];↩
val it = ["test","test1","test2"] : string list
- find pp nil;↩

uncaught exception Find
```

述語と，その述語に渡すことができる要素からなるリストを受け渡し，最初にその述語を満たす要素を返す関数.

2.13 関数 assoc

```
- assoc;↩
val it = fn : ''a -> (''a * 'b) list -> ''a * 'b
- assoc 3 [(3,4),(33,4),(11,2)];↩
val it = (3,4) : int * int
- assoc 3 [(22,33),(33,4),(3,4),(44,55)];↩
val it = (3,4) : int * int
```

2 つの要素の組のリストをから第一要素をキーとして検索する関数.

2.14 関数 rev_assoc

```
- rev_assoc;↩
val it = fn : ''a -> ('b * ''a) list -> 'b * ''a
- rev_assoc "aa" [("aa","bb"),("cc","dd"),("ee","aa"),("test","aho")];↩
val it = ("ee","aa") : string * string
```

2 つの要素の組のリストから第二要素をキーとして検索する関数.

2.15 関数 filter

```
- filter;↩
val it = fn : ('a -> bool) -> 'a list -> 'a list
- fun px x = if x > 3 then true else false;↩
val px = fn : int -> bool
- filter px [45,1,2,3,4,5,6,67];↩
val it = [45,4,5,6,67] : int list
- filter px nil;↩
val it = [] : int list
```

述語と，その述語に渡すことができる要素からなるリストを受け渡し，述語を満たす要素からなるリストを返す.

2.16 関数 flat

```
- flat;↩
val it = fn : 'a list list -> 'a list
- flat [["aa","bb","dd"],["dd","ff"],["hh","ll","qq"]];↩
val it = ["aa","bb","dd","dd","ff","hh","ll","qq"] : string list
- flat nil;↩
val it = [] : 'a list
```

2.17 関数 combine

```
- combine;↩
val it = fn : 'a list * 'b list -> ('a * 'b) list
- combine ([1,2,3,4],["a","b","c","d"]);↩
val it = [(1,"a"),(2,"b"),(3,"c"),(4,"d")] : (int * string) list
- combine ([1,2,3],["a","b"]);↩

uncaught exception Combine
```

以下の仕様を満たす関数.

$$\texttt{combine}([x_1,\cdots,x_n],[y_1,\cdots,y_n]) = [(x_1,y_1),\cdots,(x_n,y_n)]$$

2.18 関数 split

(let···in···end を使ってもよい)

```
- split;↩
val it = fn : ('a * 'b) list -> 'a list * 'b list
- split nil;↩
val it = ([],[]) : 'a list * 'b list
- split [("a",1),("b",2),("c",3),("d",4)];↩
val it = (["a","b","c","d"],[1,2,3,4]) : string list * int list
```

以下の仕様を満たす関数.

$$\texttt{split}[(x_1, y_1), \cdots, (x_n, y_n)] = ([x_1, \cdots, x_n], [y_1, \cdots, y_n])$$

2.19 関数 forall

以下の仕様をみたす関数.

```
- forall;↩
val it = fn : ('a -> bool) -> 'a list -> bool
- fun pl x = if x < 10 then true else false;↩
val pl = fn : int -> bool
- forall pl [32,43,54,56];↩
val it = false : bool
- forall pl [1,2,3,4,5,6];↩
val it = true : bool
```

exists は既定義関数である.

```
- exists;↩
val it = fn : ('a -> bool) -> 'a list -> bool
- fun pp x = if x = "aa" then true else false;↩
val pp = fn : string -> bool
- exists pp ["test","bb","cc"];↩
val it = false : bool
- exists pp ["test","aa","bb","cc"];↩
val it = true : bool
```

この関数を使って，関数 forall を定義せよ．関数の型は exists と同じである.

2.20 関数 mem

```
- mem;↩
val it = fn : ''a -> ''a list -> bool
- mem 3 [1,2,3,4,5];↩
val it = true : bool
- mem "a" ["a","b","c"];↩
val it = true : bool
- mem "aa" ["a","b","c","d"];↩
val it = false : bool
```

第一引数の要素が，第二引数のリストにあれば true，なければ false を返す関数．関数 exists を使って定義せよ．

2.21 関数 intersect

```
- intersect;↩
val it = fn : ''a list -> ''a list -> ''a list
- intersect [23,34,45,56,67] [34,45,999];↩
val it = [34,45] : int list
```

以下の仕様を満たす関数．

$$\texttt{intersect}\ l_1\ l_2 = l_1 \cap l_2$$

2.22 関数 subtract

```
- subtract;↩
val it = fn : ''a list -> ''a list -> ''a list
- subtract ["a","b","c","d"] ["d","c","ef"];↩
val it = ["a","b"] : string list
```

以下の仕様を満たす関数．$-$ は差集合を表す演算子．

$$\texttt{subtract}\ l_1\ l_2 = l_1 - l_2$$

2.23 関数 union

```
- union;↩
val it = fn : ''a list -> ''a list -> ''a list
- union ["at","then","where"] ["then","so","what","where"];↩
val it = ["at","then","where","so","what"] : string list
```

以下の仕様を満たす関数．

$$\texttt{union}\ l_1\ l_2 = l_1 \cup l_2$$

2.24 関数 distinct

```
- distinct;↩
val it = fn : ''a list -> bool
- distinct nil;↩
val it = true : bool
- distinct [11,2,2,3,3,4];↩
val it = false : bool
- distinct [1,2,3,4];↩
val it = true : bool
- distinct ["aa","aaa","bbb","ccc"];↩
val it = true : bool
```

引数で受け渡すリストの要素が互いに独立な場合に true を返し，他の場合 false を返す．

2.25 関数 setify

```
- setify;↩
val it = fn : ''a list -> ''a list
- setify [2,2,23,4,3,1,2];↩
val it = [23,4,3,1,2] : int list
- setify ["aa","bb","test","aa","dd","cc"];↩
val it = ["bb","test","aa","dd","cc"] : string list
```

引数で示すリストの要素で同じものは削除し，最後に出現した要素のみ残す.

2.26 関数 set_equal

```
- set_equal;↩
val it = fn : ''a list -> ''a list -> bool
- set_equal [1,2,3,4] [4,3,2,1];↩
val it = true : bool
- set_equal ["aa","bb","cc"] ["aa","bb"];↩
val it = false : bool
- set_equal ["aa","bb","cc"] ["bb","aa","cc"];↩
val it = true : bool
```

第 1 引数と第 2 引数にリストを渡し，要素が等しい時 true，それ以外の時 false を返す (順不同).

2.27 関数 take

```
- take;↩
val it = fn : int -> 'a list -> 'a list
- take 3 [3,4,5,6];↩
val it = [3,4,5] : int list
- take 0 [3,4,5,6];↩
val it = [] : int list
- take 5 [3,4,5,6];↩
val it = [3,4,5,6] : int list
```

第 1 引数で示す整数値に位置するリスト (第 2 引数) の要素までのリストを返す.

2.28 関数 alltrue

```
- alltrue;↩
val it = fn : ('a -> bool) -> 'a list -> bool
- alltrue (curry op= 1) [2,3,1];↩
val it = false : bool
- alltrue (curry op= "a") ["a","a"];↩
val it = true : bool
- alltrue (curry op= 1) [];↩
val it = true : bool
```

bool 型リストを渡し，全てが true の場合は true，そうでない場合は false を返す関数.

2.29 演算子@

演算子@を再定義せよ．新しい名前~として定義せよ.

```
- infix ~↩
infix ~
- fun op ~ (nil,l) =   ↩
= |   op ~ (a::r,l) =          ;↩
val ~ = fn : 'a list * 'a list -> 'a list
- ["a","b"] ~ ["c"];↩
val it = ["a","b","c"] : string list
- op~;↩
val it = fn : 'a list * 'a list -> 'a list
```

上の空白を埋め，プログラムを完成せよ．@ を使わずに再定義せよ.

2.30 演算子 o

演算子 o を再定義せよ．新しい名前~として定義せよ.

```
- infix ~;↩
infix ~
- fun f ~ g =       ↩
val ~ = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
- (twice ~ twice ~ twice) 4;↩
val it = 32 : int
```

上の空白を埋めプログラムを完成せよ．o を使わず再定義せよ.

3

さらなる構文と例題

この章では，1 章に続きより実践的なプログラムを書くための構文を紹介する．

3.1 unit 型

関数は，その性質上必ず「値を返さなければならない」．しかし，値を表示する関数 print など，値を返す必要がない場合または，いろいろな値を返す可能性がある (例えば，引数で渡した値を返そうとした場合など) 場合，値 () を返すことにより「関数」としての性質を保つ．値 () の型は unit 型で，逆に unit 型はこの値 () しかない．

操作 30: unit 型と関数 print

```
- ();↩
val it = () : unit
- print 10;↩
10val it = () : unit
- print "str";↩
strval it = () : unit↩
- print "\n";↩

val it = () : unit
- print true;↩
trueval it = () : unit
- print 3.1;↩
3.1val it = () : unit
```

関数 print は，**デバッグ** (debugging) の時に有用である．ただし "\n" を表示しないと改行しないことに注意しよう．

3.2 let ··· in ··· end

let ··· in ··· end を関数定義の中で定義することによって，その定義を入れ子にすることができる．また，let ··· in には，変数または関数を局所的に定義することができ，その識別子 (identifier) の**スコープ** (scope) を作ることができる．

ではニュートン法を使ってルート (2 乗根) と，3 乗根を求める関数を定義する．ニュートン法とは (ルートを求める場合),

$$f(x) = x^2 - a$$

とおき，$y = f(x)$ のグラフが x 軸 と交わる点を 1 次曲線 (直線) で近似する方法である．その繰り返しは，

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - a}{2x_n}$$

よりプログラムは以下のようになる．

プログラム 3: 関数 newton，ファイル名 (newton.sml)

```
fun deriv f x dx = (f(x+dx) - f(x)) / dx;
fun abs x = if x > 0.0 then x else ~x;
fun newton f epsilon =
  let fun until p change x = if p(x) then x
    else until p change (change(x)) in
  let fun satisfied y = abs(f y) < epsilon in
  let fun improve y = y - (f(y) / (deriv f y epsilon)) in
    until satisfied improve
  end end end;
fun square_root x epsilon = newton (fn y => y*y-x) epsilon x;
fun cubic_root x epsilon = newton (fn y => y*y*y-x) epsilon x;
```

関数 deriv は，第 1 引数で示す関数の微分を求める関数である．関数 abs は，引数で示す値の絶対値を求める関数，関数 newton がニュートン法により値を求める関数である．第 1 引数には，実際に近似する関数，第 2 引数には,「近似する度合」を指定する．

操作 31: 関数 newton の実行

```
- use "newton.sml";↩
[opening newton.sml]
val deriv = fn : (real -> real) -> real -> real -> real
val abs = fn: real -> real
val newton = fn : (real -> real) -> real -> real -> real
val square_root = fn : real -> real -> real
val cubic_root = fn : real -> real -> real
val it = () : unit
- square_root 2.0 1E~5;↩
val it = 1.41421569552701: real
- cubic_root 8.0 1E~5;↩
val it = 2.00000000130563 : real
- 2.0 * (newton cos 1E~5 1.5);↩
val it = 3.14159265358854 : real
```

let ··· in ··· end 構文を使うことにより in と end の間に式の**続き** (sequence) を書くことができるがその他の場所に書く場合は 括弧 (( )) で囲む必要がある．階乗を求める関数 factorial を考える．

プログラム 4: 関数 factorial

```
fun factorial 1 = 1
|  factorial n = n * (factorial (n-1));
```

また，その実行は，

操作 32: 関数 factorial の実行 (1)

```
- factorial 10;↩
val it = 3628800 : int
- factorial 12;↩
val it = 479001600 : int
```

しかし，このプログラムを実行するとして，途中経過を表示しようとすると，

プログラム 5: 途中経過を表示した関数 factorial

```
fun factorial 1 = 1
|  factorial n = ( print (factorial (n-1)); print "\n"; n * (factorial (n-1)) );
```

のようになる．またこの関数を実行すると以下のようになる．

操作 33: 関数 factorial の実行 (2)

```
- factorial 1;↩
val it = 1 : int
- factorial 2;↩
1
val it = 2 : int
- factorial 3;↩
1
2
1
val it = 6 : int
- factorial 4;↩
1
2
1
6
1
2
1
val it = 24 : int
```

(( ) で囲まれた「式の続きの値」は最後に評価された式の値である (この場合は n*(factorial (n-1)))．

3.3 ストリーム

ファイルからの入出力は，**ストリーム** (stream) を使う．そのための型として instream，outstream が用意されている．

サンプル 1: ファイル test

```
This is a test.
```

というファイル test があるとすると，

操作 34: ファイルからの入力

```
- val IN = open_in("test");↩
val IN = - : instream
- input(IN,1);↩
val it = "T" : string
- input(IN,1);↩
val it = "h" : string
```

のように，関数 open_in でファイルをオープンし，関数 input でその中の文字を取り出すことができる．ではファイルの中身をリストに変換する関数を定義する．

操作 35: 関数 readList

```
- fun readList(file) =↩
=     if end_of_stream(file) then nil↩
=     else input(file,1) :: readList(file);↩
val readList = fn : instream -> string list
- readList(IN);↩
val it = ["T","h","i","s"," ","i","s"," ","a"," ","t","e",...] : string list
```

関数 end_of_stream によりファイルの最後かをチェックしている．outstream に対しても同様に処理することができる．ファイルのオープンには関数 open_out を使い，文字列を出力する関数 output を使う．

操作 36: ファイルへの出力

```
- val OUT = open_out("test");↩
val OUT = - : outstream
- output(OUT,"That is a sample.\n");↩
val it = () : unit
- flush_out(OUT);↩
val it = () : unit
```

関数 open_out を実行することによりファイルの中身が 0 になるので注意しよう．関数 output を実行しても即座にファイルに出力されない．関数 flush_out を実行することにより実際に出力される．標準入出力のための変数は instream 型の std_in と，outstream 型の std_out が用意されている．通常使わなくなったファイルは，関数 close_in，関数 close_out によりクローズ処理をする．

3.4 実行可能プログラムの生成

SML で作成したプログラムを実行可能なコマンドとして生成することができる．

1. 作成したプログラム (関数) を含んだ新たな SML として生成する
2. 実行可能なコマンドとして生成する

の 2 通りを考えることができる．1. は，sml として起動した場合と同様であるが，起動時に新たに定義した関数を含んでいるシステムを生成する．2. は，全く新たなコマンドとして生成する．この場合，起動時の引数をどのように扱うかが問題となる．

3.4.1 関数 exportML

作成したプログラムを含んだ新たな SML として生成する場合は，関数 exportML を使う．

プログラム 6: ファイル export.sml

```
fun comb(n,m) =
  let exception BadN; exception BadM; in
    if n <= 0 then raise BadN
    else if m < 0 orelse m > n then raise BadM
    else if m=0 orelse m=n then 1
    else comb(n-1,m) + comb(n-1,m-1)
  end;
exportML("newsml");
```

関数を定義した後に，関数 exportML を呼び出すことにより上で定義した関数を含んだ新たな SML を生成する．関数 comb は，場合の数 (combination) を求める関数で，以下の公式により導くことができる．

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

$$\binom{n}{n} = \binom{n}{0} = 1$$

3.4.2 新たなコマンドとして生成する

新たなコマンドとして SML のプログラムを生成する．

プログラム 7: ファイル combfile.sml

```
fun comb1(n,m) =
  let exception BadN; exception BadM; in
    if n <= 0 then raise BadN
    else if m < 0 orelse m > n then raise BadM
    else if m=0 orelse m=n then 1
    else comb1(n-1,m) + comb1(n-1,m-1)
  end;
fun scanInt(nil) = 0
|   scanInt(x::xs) = ord(x) - ord("0") + 10*scanInt(xs);
fun stringToInt(s) = scanInt(rev(explode(s)));
fun comb([_,n,m],_) = ( print(comb1(stringToInt(n), stringToInt(m)));
        print("\n"));
exportFn("combfile", comb);
```

関数 comb1 は，上の関数と同じである．comb([_,n,m],_) と定義することにより 2 引数 n,m を取り出すことができる．しかし，各引数は文字列型の値として取り出すので，整数型に変換する必要がある．

3.5 関数 explode と implode

文字列を一文字ごとに，文字列リストに展開したり，逆に 文字列に戻す関数が，各々関数 explode，関数 implode である．

操作 37: 関数 explode と，関数 implode

```
val it = () : unit
- explode;↩
val it = fn : string -> string list
- implode;↩
val it = fn : string list -> string
- explode "the test";↩
val it = ["t","h","e"," ","t","e","s","t"] : string list
- explode "this is a test";↩
val it = ["t","h","i","s"," ","i","s"," ","a"," ","t","e",...] : string list
- implode ["t","h","i","s","\n"];↩
val it = "this\n" : string
- implode (explode "the test");↩
val it = "the test" : string
```

3.6 ソート

ここでは，アルゴリズムの典型的な例であるソートを紹介する．簡単のために，整数型の値のリストの要素をソートする場合を説明する．

3.6.1 バブルソート

文字どおり「泡」のようにソートを行うのがバブルソートである．例えば，

```
bubblesort [⑦, ①, ~5, 9, 3, 6, 4, ~2, 8];
bubblesort [1, ⑦, ~⑤, 9, 3, 6, 4, ~2, 8];
bubblesort [1, ~5, ⑦, ⑨, 3, 6, 4, ~2, 8];
bubblesort [1, ~5, 7, ⑨, ③, 6, 4, ~2, 8];
bubblesort [1, ~5, 7, 3, ⑨, ⑥, 4, ~2, 8];
bubblesort [1, ~5, 7, 3, 6, ⑨, ④, ~2, 8];
bubblesort [1, ~5, 7, 3, 6, 4, ⑨, ~②, 8];
bubblesort [1, ~5, 7, 3, 6, 4, ~2, ⑨, ⑧];
bubblesort [1, ~5, 7, 3, 6, 4, ~2, 8, 9];
```

上の図では，左から 2 つづつの値を比較し，大きい方が必ず右になるように交換を行っている．全ての要素に対してこの処理を行うことにより結果として一番右の要素が最大数となる (この場合は 9)．しかし，これだけでは全ての数がソートされたわけではないので，この処理を 1 パスとし，(最大) その要素数だけこの処理が必要となる．この例では，以下のように処理する．

```
2 パスの結果      [~5, 1, 3, 6, 4, ~2, 7, 8, 9]
3 パスの結果      [~5, 1, 3, 4, ~2, 6, 7, 8, 9]
4 パスの結果      [~5, 1, 3, ~2, 4, 6, 7, 8, 9]
```

```
5 パスの結果      [~5, 1, ~2, 3, 4, 6, 7, 8, 9]
6 パスの結果      [~5, ~2, 1, 3, 4, 6, 7, 8, 9]
```

となり 6 パスが必要なことがわかる．では，この処理をする関数 bubblesort を定義してみよう．

プログラム 8: 関数 bubblesort

```
fun bubblesort xs = until is_sorted swop xs;
```

として定義する．関数 swop は 2 要素を取り出し，比較し，左辺 > 右辺の関係が成立すれば，2 要素を交換し，以下繰り返しとなる．関数 until は，第 1 引数で示す述語が true になるまで (f x) を繰り返す演算子である．各関数は以下のようになる．

プログラム 9: 関数 swop と 関数 bubblesort

```
fun swop [] = []
|  swop [x] = [x]
|  swop ((x:int)::y::xs) = if x <= y then x::swop (y::xs)
     else y::swop (x::xs);
fun until p f x = if p x then x else until p f (f x);
```

関数 is_sorted は，ソートされたかどうかをチェックする関数である．バブルソートの繰り返しは最大「要素数」であることは明らかであるがその繰り返しをできるだけ少なくするための措置である．

プログラム 10: 関数 is_sorted

```
fun is_sorted xs = let fun isless ((x:int),y) = x < y
  in alltrue isless (zip xs (tl xs)) end;
```

この中で使用している関数 zip は，整数型リストの隣り合った要素の組からなるリストを返すことを目的とした関数である．

プログラム 11: 関数 zip と関数 alltrue

```
fun zip [] ys = []
|  zip (x::xs) [] = []
|  zip (x::xs) (y::ys) = (x,y)::zip xs ys;
```

操作 38: 関数 zip

```
- val xs = [1,2,3,4];↩
val xs = [1,2,3,4] : int list
- zip xs (tl xs);↩
val it = [(1,2),(2,3),(3,4)] : (int * int) list
```

関数 alltrue は 2 章で定義した関数である．

3.6.2 クイックソート

もっとも有名で，効率がよいといわれているソートがクイックソートである．たとえば以下のように処理する．

```
qsort [⑦, 1, ~5, 9, 3, 6, 4, ~2, 8] -> qsort [1, ~5, 3, 6, 4, ~2] @ [7] @ qsort [9, 8]
qsort [①, ~5, 3, 6, 4, ~2] -> qsort [~5, ~2] @ [1] @ qsort [3, 6, 4]
qsort [⑤, ~2] -> qsort [] @ [~5] @ qsort [~2]
qsort [~2] -> [~2]
qsort [③, 6, 4] -> qsort [] @ [3] @ qsort [6, 4] -> [3, 4, 6]
qsort [1, ~5, 3, 6, 4, ~2] -> [~5, ~2, 1, 3, 4, 6]
qsort [9, 8] -> [8, 9]
```

結果として，

```
qsort [7, 1, ~5, 9, 3, 6, 4, ~2, 8] -> [~5, ~2, 1, 3, 4, 6, 7, 8, 9]
```

となり，ソートが完了する．関数 qsort に受渡される第 1 要素を元にして，その数より小さい値と大きい値に分割し (中間値を求めて分けるという方法もあるが，中間値を求めるだけで処理が必要となる)，その分けられたリスト各々でまたソートを行う．最終的に，2 要素か，単一要素まで分割し (2 要素の場合はその大小を判断し交換を行う)，各リストを結合 (アペンド演算子@を用いる) しソートを完了する．クイックソートは，平均して N 個の要素に対して $Nlog_2N$ 回の比較で処理できることが知られている (並列でない計算では最も少ない回数)．では，この関数 qsort を定義してみよう．

プログラム 12: qsort 関数

```
fun qsort [] = []
|  qsort ((x:int)::xs) = qsort (filter (curry op>= x) xs) @ [x] @
     qsort (filter (curry op< x) xs);
```

ここで関数 filter は 2 章で定義された関数である．クイックソートは，このように分割してソートを行い，**分割統治アルゴリズム** (divide and conquer algorithm) の一つであるということができる．

3.6.3 インサーションソート (挿入法)

挿入法というソートは，コンピュータでなく一般に人間が (資料などを) ソートする方法である．ただし，効率はあまり良いわけではない．例えば以下のようにソートする．

```
isort [7, 1, ~5, 9] -> insert 7 (isort [1, ~5, 9])
                    -> insert 7 (insert 1 (isort [~5, 9]))
                    -> insert 7 (insert 1 (insert ~5 (isort [9])))
                    -> insert 7 (insert 1 (insert ~5 (insert 9 (isort []))))
                    -> insert 7 (insert 1 (insert [~5, 9]))
                    -> insert 7 [~5, 1, 9]
                    -> [~5, 1, 7, 9]
```

リストの要素を左から一つづつ取り出し，別のリスト (最初は []) の適切な部分に挿入していく方法である．プログラムは以下のようになる．

プログラム 13: 関数 insert と 関数 isort

```
fun insert (x:int) [] = [x]
|  insert x (y::ys) = if x < y then x::y::ys else y::insert x ys;
fun isort [] = []
|  isort (x::xs) = insert x (isort xs);
```

3.7 例題

では，ここで比較的大きな例題を紹介しよう．テキスト (アルファベットと区切り文字) からその語数を数える関数を考える．テキストには，アルファベットの他に，区切り文字や空白 (空白に相当する文字) などいくつかの文字が含まれる．

操作 39: 関数 punct と blanks

```
- val punct = [",",":","(",")",";","'","\""];↩
val punct = [",",":","(",")",";","'","\""] : string list
- val blanks = ["\n","\t"];↩
val blanks = ["\n","\t"] : string list
```

を定義する．そこで，問題となる関数を以下のように定義することを考える．

操作 40: total 関数

```
- fun total xs = (sum o count) xs;↩
val it = fn : string -> int
- fun sum xs = foldl (curry op+) 0 xs;↩
val sum = fn : int list -> int
```

ここで xs は，引数として受渡す文字列である．その文字列 xs を文 (. まで) ごとに区切り，語数を数える関数が count である．そして，関数 sum にそのリスト (整数型リスト) として適用し，総和を求めている．関数 foldl は，左から関数を適用する関数で，同様に右から関数を適用する関数 foldr も定義することができる． 仕様は以下の通りである．

$$\texttt{foldl}\ f\ a\ [x_1, x_2, \cdots, x_n] \ = \ f\ (f\ (\cdots (f\ a\ x_1) \cdots) x_{n-1})\ x$$

$$\texttt{foldr}\ f\ a\ [x_1, x_2, \cdots, x_n] \ = \ f\ x_1\ (f\ x_2\ (\cdots (f\ x_n\ a) \cdots))$$

この関数を定義すると以下のようになる．

プログラム 14: 関数 foldl と関数 foldr

```
fun foldl f a [] = a
|   foldl f a (x::xs) = foldl f (f a x) xs;
fun foldr f a [] = a
|   foldr f a (x::xs) = f x (foldr f a xs);
```

関数 count は，以下の通りである．

プログラム 15: 関数 count

```
fun count xs = (map findtotal o map split_up o split o reformat o replace) xs;
```

ここで新たに，関数 replace，reformat，split，split_up，findtotal を定義する必要がある．関数 replace は，

操作 41: 関数 replace

```
- replace "AA BB. C:DD.";↩
val it = ["A","A"," ","B","B",".","C"," ","D","D","."] : string
list
```

ピリオド (.) を除く区切り文字を空白に置き換える関数である．そして，関数 reformat により複数の空白を一つの空白としてまとめ，関数 split により，残ったピリオドをもとに文として切り分ける．そして文字列リストにしてまとめる．上の例は，以下のようになる．

操作 42: 関数 reformat，split を適用

```
- (split o reformat o replace) "AA BB. C:DD.";↩
val it = ["AA BB","C DD"] : string list
```

関数 split_up により，各文字列を単語に分割し，文字列リストとする．結果として，文字列リストのリストとなる．更に，各要素に，関数 findtotal を適用することにより，語数を数えて整数型リストとなる．この 2 つの関数は，各要素に対して適用を行うので 関数 map を用いる．

操作 43: 関数 split_up と，関数 findtotal

```
- (map split_up o split o reformat o replace) "AA BB. C:DD.";↩
val it = [["AA","BB"],["C","DD"]] : string list list
- (map findtotal o map split_up o split o reformat o replace) "AA BB. C:DD.";↩
val it = [2,2] : int list
```

では，各関数を定義していこう．まず，関数 replace を定義するが，その中で用いる関数 member を定義する．関数 member は，第 1 引数で指定する要素が，第 2 引数で指定するリスト中に存在するかどうかをチェックする関数である．既定義の関数 exists を使い定義することができる．また，簡単のために infix 指定をし中置の演算子とする．

プログラム 16: 関数 member

```
infix member;
fun x member xs = exists (curry op= x) xs;
```

この関数 member を使うと関数 replace は以下のようになる．

プログラム 17: 関数 replace

```
fun replace xs =
  let fun valid c =
    let val punct = [",",":","(",")",";","'","\""];
      val blanks = ["\n","\t"]
    in if c member punct orelse c member blanks then " " else c end
  in map valid (explode xs)
end;
```

また，関数 reformat，関数 split は以下のようになる．

プログラム 18: 関数 reformat

```
fun reformat (c::d::cs) =
  if c = " " andalso d = "" then reformat (d::cs) else (c::reformat (d::cs))
| reformat [c] = if c = " " then [] else [c]
| reformat [] = [];
```

プログラム 19: 関数 split

```
fun dropwhile p [] = []
|  dropwhile p (x::xs) = if p x then dropwhile p xs else x::xs;
fun takewhile p [] = []
|  takewhile p (x::xs) = if p x then x::takewhile p xs else [];
fun split [] = []
|  split ("."::cs) = split cs
|  split (" "::cs) = split cs
|  split cs = let val sent = takewhile (curry op<> ".") cs;
     val rest = dropwhile (curry op<> ".") cs in
       ((implode sent)::(split rest))
     end;
```

ここで，関数 dropwhile，takewhile は，第 2 引数で示すリストの各要素に対し，第 1 引数で示す述語 (predicate, ある条件に合致するかどうかをチェックする関数で，true または false を返す関数) により各々 true を返すまで，リストを削除 (ドロップ)/連結する関数である．

以上の関数を適用した後，関数 map を使って各要素に適用する関数 split_up，findtotal は以下のようになる．

プログラム 20: 関数 split_up

```
fun split_up' [] = []
|  split_up' (" "::cs) = split_up' cs
|  split_up' cs = let val word = takewhile (curry op<>" ") cs;
     val rest = dropwhile (curry op<>" ") cs
       in (implode word)::(split_up' rest) end;
fun split_up cs = split_up' (explode cs);
```

プログラム 21: 関数 findtotal

```
fun findtotal xs = let fun inc w n = n+1 in foldr inc 0 xs end;
```

3.8 例題の検証

プログラムが，ある程度書きあがったらそのプログラムをテストまたは，検証するプログラムを書かなければならない．では，上の例題を検証するプログラムを書いてみる．

通常検証するプログラムは，ファイルからテストデータを読み，プログラム (関数) を実行し，またファイルに出力する．その後，出力された関数が期待どおりの内容かどうかをチェックする．Standard ML では関数 open_in,

open_out を使ってファイルをオープンし，関数 input，output によりデータの読み書きを行う．ただし，データを書き出す場合は close 処理 (close_in または close_out) を実行しなければならない．

プログラム 22: 関数 format

```
fun format f xs = let val OUT = open_out f in
  output( OUT, implode xs ); close_out(OUT) end;
```

この関数 format は，

操作 44: 関数 format の使い方

```
- format "f" ((replace (input ((open_in "data"), 1000))));↩
```

とし，ファイル data からテストデータを読み込み，ファイル f にその結果を出力する．

別のやり方を考える．関数一つ，一つをテストまたは検証するには，各関数の性質をよく考えそれにあったチェックを行う関数を書く．例えば，関数 replace には，以下のような関数 validchars を考える．

プログラム 23: 関数 validchars

```
fun not_in [] x = true
|  not_in xs x = not (x member xs);
fun validchars [] = true
|  validchars (x::xs) = let val punct = [",",":","(",")",";","'","\""];
     val blanks = ["\n","\t"] in
       (alltrue (not_in blanks) xs) andalso (alltrue (not_in punct) xs)
     end;
```

この関数 validchars は，

操作 45: 関数 validchars の使い方

```
- validchars (replace "a\nb,c.d:e(f)g;h");↩
val it = true : bool
```

となる．他の関数も以下のようにテストする．例えば，

サンプル 2: サンプルデータ

```
One of the main problems that is central to
the software production
process is to identify the nature of "progress"
and to find some way of
measuring it. There is no theory which enables us
to calculate limits
on the size, performance or complexity of software.
```

のような，ファイル data を用意する．

操作 46: 関数 split, split_up のテスト

```
- split (explode "The theory. Of types");↩
val it = ["The theory","Of types"] : string list
- split_up ("The theory of types");↩
val it = ["The","theory","of","types"] : string list
- map split_up (split (reformat (replace (input ((open_in "data"), 1000 ))))));↩
val it =
  [["One","of","the","main","problems","that","is","central","to","the",
  "software","production",...],
  ["There","is","no","theory","which","enables","us","to","calculate",
  "limits","on","the",...]] : string list list
```

となる.

4

3 章のドリル

各問に答えよ．レポートは jLaTeX を使って清書し，プログラムの説明，考察をせよ．

4.1 関数 is_sorted

関数 is_sorted をここで紹介した関数を使わずに定義せよ．

4.2 関数 qsort

関数 qsort を 関数 filter，curry を使わずに定義せよ．

4.3 新しいコマンドとして

バブルソートをする関数 bubblesort を 新たなコマンドとして定義するためのプログラムを実現せよ (ファイル compfile.sml 参照)．なお，引数は，整数値 5 個固定で，ソート結果を表示せよ．

4.4 インサーションソートの別解

関数 foldr，関数 takewhile，関数 dropwhile を用いてインサーションソートを定義することを考える．

プログラム 24: 関数 foldr，関数 takewhile，関数 dropwhile

```
fun foldr f a [] = a
|   foldr f a (x::xs) = f x (foldr f a xs);
fun dropwhile p [] = []
|  dropwhile p (x::xs) = if p x then dropwhile p xs else x::xs;
fun takewhile p [] = []
|  takewhile p (x::xs) = if p x then x::takewhile p xs else [];
```

プログラム 25: 関数 insert

```
fun isort [] = []
|  isort xs = foldr insert [] xs;
fun insert (x:int) xs = ____①____ @ [x] @ ____②____;
```

①，② に各々 関数 takewhile，関数 dropwhile を使うことによりプログラムを完成せよ．

4.5 マージソート

クイックソートと同様に，分割統治アルゴリズムの 1 種であるマージソートを紹介する．以下のように処理する．

```
msort [7, 1, ~5, 9] -> merge (msort [7,1]) (msort [~5, 9])
                    -> merge (merge (msort [7]) (msort [1])) (merge (msort [~5]) (msort [9]))
                    -> merge (merge [7] [1]) (merge [~5] [9])
                    -> merge [1,7] [~5, 9]
                    -> [~5, 1, 7, 9]
```

この処理をするプログラムは，以下のようになる．

プログラム 26: 関数 msort，関数 drop，関数 take

```
fun drop 0 xs = xs
|  drop n [] = []
|  drop n (x::xs) = drop (n-1) xs;
fun msort xs = let val n = length xs in
   if n <= 1 then xs else
     let val ys = take (n div 2) xs; val zs = drop (n div 2) xs in
     merge (msort ys) (msort zs) end end;
```

ここで呼び出される関数 merge を定義せよ (問 1)(なお，関数 take は 2 章で定義した関数である)．また，別の方法で関数 msort を定義せよ (問 2).

プログラム 27: 関数 merge

```
fun merge [] ys = ys
|  merge ((x:int)::xs) [] = (x::xs)
|  merge (x::xs) (y::ys) = ____________①____________;
```

4.6 ここで紹介している図を表示

さらに，時間のある人は，ここで紹介している図を表示するプログラムを実現せよ．なお，各ソートごとに別の関数でよい．例えば，関数 qsort の場合は，以下のようである (各自独自にアレンジしてよい).

```
qsort [⑦, 1, ~5, 9, 3, 6, 4, ~2, 8] -> qsort [1, ~5, 3, 6, 4, ~2] @ [7] @ qsort [9, 8]
qsort [①, ~5, 3, 6, 4, ~2] -> qsort [~5, ~2] @ [1] @ qsort [3, 6, 4]
qsort [⑤, ~2] -> qsort [] @ [~5] @ qsort [~2]
qsort [~2] -> [~2]
qsort [③, 6, 4] -> qsort [] @ [3] @ qsort [6, 4] -> [3, 4, 6]
qsort [1, ~5, 3, 6, 4, ~2] -> [~5, ~2, 1, 3, 4, 6]
qsort [9, 8] -> [8, 9]
```

4.7 セレクションソート (選択法)

以下のように処理する．

```
ssort [7, 1, ~5, 9] -> ~5 :: ssort [7, 1, 9]
                    -> ~5 :: (1 :: ssort [7, 9])
                    -> ~5 :: (1 :: (7 :: ssort [9]))
                    -> ~5 :: (1 :: (7 :: (9 :: [])))
                    -> [~5, 1, 7, 9]
```

引数で示されたリストの中で最小の要素を取り出し，前に結合していくやり方である．比較回数が $O(n^2)$ であることが知られている．以下の関数を用いて，関数 ssort を定義せよ．

プログラム 28: 中置演算子 -，関数 min，関数 fold1

```
infix --;
fun xs -- [] = xs
|  xs -- (y::ys) = let fun remove [] y = []
     |  remove (x::xs) y = if x = y then xs else x::(remove xs y)
     in ((remove xs y) -- ys) end
fun min2 a (b:int) = if a<=b then a else b;
fun foldl1 f (x::xs) = foldl f x xs; fun min xs = foldl1 min2 xs;
```

関数 foldl は 3 章で定義した関数である．

プログラム 29: 関数 ssort

```
fun ssort [] = []
|  ssort xs = ____①____;
```

① を埋めることによりプログラムを完成せよ．

4.8 バケットソート

以下のように処理する．

```
bucketsort [25,36,14,9,56,21,55,72,84,93,6,51,62,75,37,12,95]
-> isort [9,6] @ bucketsort [25,36,14,56,21,55,72,84,93,51,62,75,37,12,95]
-> [6,9] @ isort [14,12] @ bucketsort [25,36,56,21,55,72,84,93,51,62,75,37,95]
-> [6,9] @ [12,14] @ bucketsort [25,36,56,21,55,72,84,93,51,62,75,37,95]
-> [6,9] @ [12,14] @ isort [25,21] @ bucketsort [36,56,55,72,84,93,51,62,75,37,95]
-> [6,9] @ [12,14] @ [21,25] @ bucketsort [36,56,55,72,84,93,51,62,75,37,95]
-> [6,9] @ [12,14] @ [21,25] @ isort [36,37] @ bucketsort [56,55,72,84,93,51,62,75,95]
-> [6,9] @ [12,14] @ [21,25] @ [36,37] @ bucketsort [56,55,72,84,93,51,62,75,95]
-> [6,9] @ [12,14] @ [21,25] @ [36,37] @ isort [56,55,51] @ bucketsort [72,84,93,62,75,95]
-> [6,9] @ [12,14] @ [21,25] @ [36,37] @ [51,55,56] @ bucketsort [72,84,93,62,75,95]
-> [6,9] @ [12,14] @ [21,25] @ [36,37] @ [51,55,56] @ isort [62] @ bucketsort [72,84,93,75,95]
-> [6,9] @ [12,14] @ [21,25] @ [36,37] @ [51,55,56] @ [62] @ bucketsort [72,84,93,75,95]
-> [6,9] @ [12,14] @ [21,25] @ [36,37] @ [51,55,56] @ [62] @ isort [72,75] @ bucketsort [84,93,95]
-> [6,9] @ [12,14] @ [21,25] @ [36,37] @ [51,55,56] @ [62] @ [72,75] @ bucketsort [84,93,95]
-> [6,9] @ [12,14] @ [21,25] @ [36,37] @ [51,55,56] @ [62] @ [72,75] @ isort [84] @ bucketsort [93,95]
-> [6,9] @ [12,14] @ [21,25] @ [36,37] @ [51,55,56] @ [62] @ [72,75] @ [84] @ bucketsort [93,95]
```

```
-> [6,9] @ [12,14] @ [21,25] @ [36,37] @ [51,55,56] @ [62] @ [72,75] @ [84] @ isort [93,95]
-> [6,9] @ [12,14] @ [21,25] @ [36,37] @ [51,55,56] @ [62] @ [72,75] @ [84] @ [93,95]
-> [6,9,12,14,21,25,36,37,51,55,56,62,72,75,84,93,95]
```

ある範囲を決め (ここでは [0,10), [10,20), ... [90,100))，その範囲に該当する要素を適宜取り出し，その間は，インサーションソートでソートを行う．そして，求めるリストの先頭にアペンドしていく．上の例のように，範囲を 10 刻みで抽出し，インサーションソートでソートする関数 bucketsort を定義せよ．

操作 47: 関数 bucketsort

```
- bucketsort [25,36,14,9,56,21,55,72,84,93,6,51,62,75,37,12,95] 0 100;↩
val it = [6,9,12,14,21,25,36,37,51,55,56,62,...] : int list
```

4.9 例題の拡張

最初に 3 章で紹介した例題の拡張を考える．そして，そのプログラムを検証するプログラムも実装せよ．以下のように拡張する．

プログラム 30: 拡張した関数

```
fun count_each ws = ( map countwords o map split_up o split o
  reformat o replace ) ws;
fun sort_and_count xs = ( freq_count o wordsort o concat o count_each ) xs;
```

この 2 つの関数を実装することを考える．関数 countwords は，

操作 48: 関数 countwords

```
- countwords ["The","theory","of","types"];↩
val it = [("types",1),("of",1),("theory",1),("The",1)] : (string * int) list
```

のように，単語とその数の組のリストを返す関数である．関数 count_each は

操作 49: 関数 count_each

```
- count_each (input ((open_in "empty"), 1000));↩
val it = [] : (string * int) list list
- count_each (input ((open_in "data"), 1000));↩
val it =
  [[("it",1),("measuring",1),("of",3),("way",1),("some",1),("find",1),("to",3),
  ("and",1),("progress",1),("nature",1),("the",3),("identify",1),...],
  [("software",1),("of",1),("complexity",1),("or",1),("performance",1),
  ("size",1),("the",1),("on",1),("limits",1),("calculate",1),("to",1),
  ("us",1),...]] : (string * int) list list
```

となる関数で，ファイル empty は，中身 0 のファイルである．各文ごとに文字列とその数のリストとし，全体として (string * int) list list 型となるようにする．関数 concat は，以下で定義する関数である．

プログラム 31: 関数 concat

```
fun concat [] = []
|  concat (x::xs) = x @ (concat xs);
```

関数 wordsort は,

操作 50: 関数 wordsort

```
- wordsort [("type",2),("theory",1)];↩
val it = [("theory",1),("type",2)] : (string * int) list
```

となる関数である．組の第 1 要素である文字列をもとにソートする．関数 freq_count は,

操作 51: 関数 freq_count(1)

```
- freq_count (wordsort(concat(count_each(input ((open_in "data"), 1000)))));↩
val it =
  [("One",1),("There",1),("and",1),("calculate",1),("central",1),
  ("complexity",1),("enables",1),("find",1),("identify",1),("is",3), ("it",1),
  ("limits",1),...] : (string * int) list
```

となる関数である．また単独では以下のようになる．

操作 52: 関数 freq_count(2)

```
- freq_count;↩
val it = fn : (''a * int) list -> (''a * int) list
- freq_count [("Is",3),("Is",2),("test",4),("test",5),("test",1)];↩
val it = [("Is",5),("test",10)] : (string * int) list
- freq_count [("This",1),("This",2),("That",1)];↩
val it = [("This",3),("That",1)] : (string * int) list
```

新たに showpair を定義しよう．

プログラム 32: 関数 showpairs

```
fun showpairs ((w,(n:int))::wcs) = "("^w^","^(makestring n)^")"::(showpairs wcs)
|  showpairs [] = [];
```

この関数を使って，以下のように実行する．

操作 53: 関数 sort_and_count

```
- format "f1" (showpairs(freq_count (wordsort(concat(count_each(input ((open_in
"data"), 1000 )))))));↩
val it = () : unit
- format "f2" (showpairs(sort_and_count (input ((open_in "data"), 1000 ))));↩
val it = () : unit
```

ファイル f1 および f2 に以下の同じ結果が書き出される.

プログラム 33: ファイル f1 および f2

```
(One,1) (There,1) (and,1) (calculate,1) (central,1) (complexity,1)
(enables,1) (find,1) (identify,1) (is,3) (it,1) (limits,1) (main,1)
(measuring,1) (nature,1) (no,1) (of,4) (on,1) (or,1) (performance,1)
(problems,1) (process,1) (production,1) (progress,1) (size,1)
(software,2) (some,1) (that,1) (the,4) (theory,1) (to,4) (us,1)
(way,1) (which,1)
```

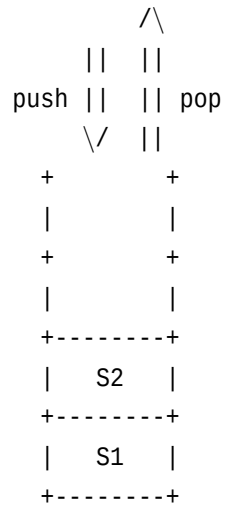
5

抽象データ型

より大きなデータ構造を表現する場合，そのデータ構造に付随した関数と共に一つの固まりとして表現したい．

5.1 スタック

最初に，簡単にスタックの例をあげる．**スタック** (stack) とは，



のような構造をしたデータで，S1，S2 の順番に push された要素は S2，S1 の順でのみしか pop できない．また通常，push できる要素は有限で，そのようなスタックを**有限スタック** (bounded stack) という．また，スタックの要素の取り出し方を LIFO(Last In First Out) という．これに対し，キューの要素の取り出し方を FIFO(First In First Out) という (後述)．さて，このスタックを SML のリストを使ってプログラムすると以下のようになる．

プログラム 34: スタックに関する関数

```
val stack = [];
fun isempty stack = stack = [];
fun push i stack = (i::stack);
fun top [] = hd []
|  top (t::l) = t;
fun pop [] = tl []
|  pop (t::l) = l;
fun showstack [] = ()
|  showstack ((t:string)::l) = ( print t; (if (not (l = [])) then
     ( print ","; showstack l ) else print "\n") );
```

またこのプログラムを実行すると以下のようになる．

操作 54: 関数の実行

```
- val stack = push "S1" stack;↩
val stack = ["S1"] : string list
- val stack = push "S2" stack;↩
val stack = ["S2","S1"] : string list
- showstack stack;↩
S2,S1
val it = () : unit
- top stack;↩
val it = "S2" : string
- val stack = pop stack;↩
val stack = ["S1"] : string list
- showstack stack;↩
S1
val it = () : unit
- isempty stack;↩
val it = false : bool
- val stack = pop stack;↩
val stack = [] : string list
- isempty stack;↩
val it = true : bool
```

このような定義では，各関数がバラバラに定義されており，型チェックも各関数独立に行いそれらの関係も独立になっている．これら関係した関数群を一つの固まりとして定義しかつ，一つの型として定義することにより型チェックの際にも重要な役割を果たすようにしたい．そのような型を**抽象データ型** (abstruct data type) という．その中の関数は，抽象データ型により**カプセル化** (encapsulation) されたという．**オブジェクト指向言語** (object oriented language) でいう「カプセル化」はここから派生した言葉である．SML では抽象データ型を定義するための構成子 `abstype` が用意されている．そして，抽象データ型 `stack` として定義することができる．

プログラム 35: 抽象データ型 stack

```
abstype stack = Stack of string list with
  val empty = Stack []
  fun isempty (Stack s) = s = []
  fun push i (Stack s) = Stack (i::s)
  fun top (Stack []) = hd []
  |  top (Stack (t::l)) = t
  fun pop (Stack []) = Stack (tl [])
  |  pop (Stack (t::l)) = Stack l
  fun show (Stack []) = ()
  |  show (Stack (t::l)) = ( print t; (if (not (l = [])) then
       ( print ","; show (Stack l) ) else print "\n") );
end
```

抽象データ型 `stack` の中身は `string list` となる．

一般に抽象データ型は以下のように定義する．

プログラム 36: 抽象データ型の定義

```
abstype 名前 = 構成子 of 型 with
  val 名前 = 変数宣言
    . . .
  fun 名前 = 関数宣言
    . . .
end
```

ここでは 構成子 of 型 のデータと with 以下 で定義する変数，関数の定義を一緒にして抽象データ型とすることを意味する．構成子 of 型 の値は，構成子 (型の値) で表現する．例えば Stack of string list は，Stack(["test","aa"]) と表現する．上のプログラムを実行すると以下のようになる．

操作 55: 抽象データ型 stack の関数の実行

```
- val s = empty;↩
val s = - : stack
- val s = push "S1" s;↩
val s = - : stack
- val s = push "S2" s;↩
val s = - : stack
- show s;↩
S2,S1
val it = () : unit
- top s;↩
val it = "S2" : string
- val s = pop s;↩
val s = - : stack
- show s;↩
S1
val it = () : unit
- isempty s;↩
val it = false : bool
- val s = pop s;↩
val s = - : stack
- isempty s;↩
val it = true : bool
```

抽象データ型 stack を定義した場合，上のように stack 型の値が - となり「カプセル化されている」．そのため，stack 型の値を変更または，取り出すなどする場合は用意された関数のみにより可能となる．表示するためには関数 show などを用意しなければならない．抽象データ型でない場合は，内部の値 (**状態** (state) という) がそのまま表示されている (文字列リスト型で実現されていることも分かってしまう)．このように抽象データ型を用いることにより,「関数の使用者 (ユーザ) が内部状態について知らなくても用意された関数を用いることにより変更できる」ことがわかる．このことは，次に説明するように 内部実現 が変わってもユーザは全く同じようにアクセスすることができる点へ結びつけることができる．また，型チェックも同様に実行される．

5.2 総称的抽象データ型

抽象データ型を使うことにより 内部実現をユーザに知られることなく データをカプセル化できることは前述のとおりである，逆に，抽象データ型の内部実現を変えても抽象データ型は同様の動作をしなければならない.「インターフェース (呼び出し操作) を別に定義する」ことにより，内部実現をいろいろ変えても同じプログラムですむ機構が用意されている．そのための構成子が signature，functor，structure である．抽象データ型 stack の例を改良してみよう．抽象データ型のインターフェースは signature を使って定義する．

プログラム 37: 抽象データ型 stack の signature

```
signature STACK = sig
  structure I : ITEM
  type stack
  val empty : stack
  val isempty : stack -> bool
  val push : I.item -> stack -> stack
  val top : stack -> I.item
  val pop : stack -> stack
  val show : stack -> unit
end;
```

ここで ITEM とはスタックの要素を示す structure である．structure とは signature で定義されたインターフェースを実現したものである．そのため ITEM にも signature を定義しなければならない.

一般に signature は，以下のように定義する．

プログラム 38: signature の定義

```
signature 名前 = sig
  内部で使う structure の宣言
  . . .
  この signature 自身の type または eqtype の宣言
  val 名前 : 型宣言
  . . .
end;
```

この signature 自身の type または eqtype の宣言，含まれる変数または関数 (val で宣言) の型宣言を行う．eqtype とは，この型に属する値が「等しい」かどうかを判断する演算子が定義されている型を示す (多相型になった時に ''a となる)．内部で structure を使う場合は，最初に宣言する．

signature ITEM を定義してみよう.

プログラム 39: ITEM の signature

```
signature ITEM = sig
  eqtype item
  val isequal : item -> item -> bool
  val show : item -> string
end;
```

この signature は 総称的抽象データ型 stack の中身を切替える際に変化するものである．実際に変化するのは，この signature を実現した structure である．

では signature で宣言したインターフェースを実現する structure を定義しよう．この具体的な実装である structure はインターフェースさえ同じであれば，複数定義することができる．ここでは signature ITEM に対して StringItem と，IntItem を定義しよう．

プログラム 40: StringItem の定義

```
structure StringItem : ITEM = struct
  type item = string
  fun isequal a b = a = b
  fun show (a:item) = a
end;
```

プログラム 41: IntItem の定義

```
structure IntItem : ITEM = struct
  type item = int
  fun isequal a b = a = b
  fun show (a:item) = makestring a
end;
```

各 structure を切替えることにより，文字列型の値を保持するスタックを作ったり，整数型の値を保持するスタックを作ったりすることができる．一般に structure は以下のように定義する．

プログラム 42: structure の定義

```
structure 名前 : 「対応する signature の名前」 = struct
  type 新しい型名 = 型
  val 変数名 = . . .
  . . .
  fun 関数名 = . . .
end;
```

structure の後に，名前，:，対応する signature を書く．実際にこの structure の内部の値の型を宣言する．例えば,「type item = int は，item 型を int 型とする」と読めばいいだろう．最後に signature で宣言された変数，関数を実現する．実際のスタックの動作は，**ファンクタ** (functor) を使って定義する．これは structure を引数として受取り structure(この場合は Stack) を返す関数である．

プログラム 43: ファンクタ MkStack の定義

```
functor MkStack (Itemstruct : ITEM) : STACK = struct
  structure I = Itemstruct
  abstype stack = Stack of I.item list with
    val empty = Stack []
    fun isempty (Stack s) = s = []
    fun push i (Stack s) = Stack (i::s)
    fun top (Stack []) = hd []
    |  top (Stack (t::l)) = t
    fun pop (Stack []) = Stack (tl [])
    |  pop (Stack (t::l)) = Stack l
    fun show (Stack []) = ()
    |  show (Stack (t::l)) = ( print (I.show t); (if (not (l = [])) then
         ( print ","; show (Stack l) ) else print "\n") )
  end
end;
```

最後に上のファンクタを使ってスタックを定義する.

プログラム 44: structure IntStack と structure StringStack

```
structure IntStack = MkStack (IntItem);
structure StringStack = MkStack (StringItem);
```

この IntStack と StringStack は以下のように実行する.

プログラム 45: IntStack の実行

```
- val s = IntStack.empty;↩
val s = - : IntStack.stack
- val s = IntStack.push 11 s;↩
val s = - : IntStack.stack
- val s = IntStack.push 22 s;↩
val s = - : IntStack.stack
- IntStack.show s;↩
22,11
val it = () : unit
- IntStack.top s;↩
val it = 22 : IntItem.item
- val s = IntStack.pop s;↩
val s = - : IntStack.stack
- IntStack.show s;↩
11
val it = () : unit
- IntStack.isempty s;↩
val it = false : bool
- val s = IntStack.pop s;↩
val s = - : IntStack.stack
- IntStack.isempty s;↩
val it = true : bool
```

ここでは IntStack の例を示したが，StringStack も同様に実行できる．上の例のように IntStack.empty のように IntStack を陽に指定しなくても実行できる．構成子 open を使えばよい．

操作 56: open を使う

```
- open IntStack;↩
open IntStack;
open IntStack
structure I : ITEM
val empty = - : stack
val isempty = fn : stack -> bool
val push = fn : IntItem.item -> stack -> stack
val top = fn : stack -> IntItem.item
val pop = fn : stack -> stack
val show = fn : stack -> unit
- val s = empty;↩
val s = - : stack
- val s = push 11 s;↩
val s = - : stack
- val s = push 22 s;↩
val s = - : stack
- show s;↩
22,11
val it = () : unit
- top s;↩
val it = 22 : IntItem.item
- val s = pop s;↩
val s = - : stack
- show s;↩
11
val it = () : unit
- isempty s;↩
val it = false : bool
- val s = pop s;↩
val s = - : stack
- isempty s;↩
val it = true : bool
```

このように open を使うことにより 抽象データ型 IntStack を同じく stack として扱うことができた．しかし，これらの抽象データ型によりカプセル化された値の型は IntItem.item となっていることに注意しよう．

最後にこのように定義した総称的 Stack を使って (演算子を継承して) 新たな抽象データ型を定義することができる．そのためには以下のようなファンクタを定義すればよい．

プログラム 46: ファンクタ MkApplications

```
functor MkApplications( structure S : STACK ) = struct
  open S
  fun size xs = if isempty xs then 0 else 1 + size (pop xs)
end;
```

このファンクタ MkApplications を以下のように使うことにより新たな structure を定義することができる．

プログラム 47: structure Applications

```
structure IntApplications = MkApplications (structure S = IntStack)
structure StringApplications = MkApplications (structure S = StringStack)
```

このプログラムを実行すると以下のようになる.

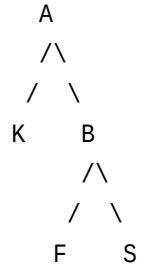
操作 57: ファンクタ MkApplications の実行

```
- open StringApplications;↩
open StringApplications
structure I : ITEM
val empty = - : stack
val isempty = fn : stack -> bool
val pop = fn : stack -> stack
val push = fn : StringItem.item -> stack -> stack
val show = fn : stack -> unit
val size = fn : stack -> int
val top = fn : stack -> StringItem.item
- val s = empty;↩
val s = - : stack
- val s = push "example1" s;↩
val s = - : stack
- val s = push "example2" s;↩
val s = - : stack
- show s;↩
example2,example1
val it = () : unit
- isempty s;↩
val it = false : bool
- size s;↩
val it = 2 : int
- StringApplications.size s;↩
val it = 2 : int
```

ここで StringApplications では，関数 size を使うことができることに注意しよう.

5.3 総称的 抽象データ型 2 分木

次に，より複雑なデータ構造である 2 分木について総称的データ構造として定義する．2 分木とは各**ノード** (node) に対して子が 2 つあるかまたは，子を持たない場合の木構造を指す．例えば,



である．この場合 A，B，F，K，S をノード (node) または**節** といい，ノード A を **ルート** (root) または **根** といい，ノード K，F，S のように子を持たないノードを**リーフ** (leaf) または，**葉**という．

通常このままでは計算機で扱うことはできないので，木構造は () を入れ子にして表す．上の例は，

((K), A, ((F), B, (S)))

となる．一つの () で一つのノードを表し，

(左の子ノード，現在のノード名，右の子ノード)

の順番で表現している．または，

(A, (K), (B, (F), (S)))

と書き，

(現在のノード名，左の子ノード，右の子ノード)

で表現する場合もある．本章では，前者の記法を使う．では，抽象データ型 tree を定義してみよう．

プログラム 48: 抽象データ型 tree

```
abstype tree = Empty | Tree of tree * I.titem * tree
```

ここで | は「または」を表し Empty か Tree であることを示す．Empty の場合は リーフであるので Empty という名前のみである．Tree の場合は tree と I.titem と tree の組を持つ．I.titem とは 以下で定義する signature の ITEM の型 (titem) を指す．ここで，ノードが持つ要素の型がどんなインターフェースを持たなければならないかを定義している．

プログラム 49: シグニチャ TITEM の定義

```
signature TITEM = sig
  eqtype titem
  val isequal : titem -> titem -> bool
  val isless : titem -> titem -> bool
  val isgreater : titem -> titem -> bool
  val show : titem -> string
end;
```

titem はこの signature 自身を示す型で，各要素を「等しい」かどうかを判定することができなければならない．isequal は左右のノードの要素を比較し「等しいか」を返す関数，isless は同じく「< か」を返す関数，isgreater は同じく「> か」を返す関数を示している．show はこのノードの要素を文字列に変換する関数，TITEM は以上の機能を持たなければならない．

この TITEM を組み合わせた木構造である signature TREE を定義しよう．

プログラム 50: シグニチャ TREE の定義

```
signature TREE = sig
  structure I : TITEM
  type tree
  val empty : tree
  val isempty : tree -> bool
  val cons : I.titem -> tree -> tree -> tree
  val left : tree -> tree
  val right : tree -> tree
  val root : tree -> I.titem
  val show : tree -> string list
end;
```

tree はこの signature TREE 自身の型，empty は TREE を空にする関数，isempty は木が空であるかを判断する関数，cons は 2 つの木を結合して一つの木にする関数である．その時親となるノードを I.titem で指定する．left，right は各々左右の子ノードを返す関数である．root はルートノードを返す関数，show は tree を string list に変換する関数である．

前例と同様に 2 分木を生成するファンクタを定義しよう．

プログラム 51: ファンクタ MkTree の定義

```
functor MkTree (Itemstruct : TITEM) : TREE = struct
  structure I = Itemstruct
  abstype tree = Empty | Tree of tree * I.titem * tree with
    val empty = Empty
    fun isempty (Tree t) = false
    |  isempty Empty = true
    fun cons i l r = Tree (l,i,r)
    fun left (Tree (l,i,r)) = l
    |  left Empty = hd []
    fun right (Tree (l,i,r)) = r
    |  right Empty = hd []
    fun root (Tree (l,i,r)) = i
    |  root Empty = hd []
    fun show (Tree (l,i,r)) = "("::(showleft l)@[I.show i]@(showright r)@[")"]
    |  show Empty = []
    and showleft (Tree (l,i,r)) = show (Tree (l,i,r))@[","]
    |  showleft Empty = []
    and showright (Tree (l,i,r)) = "," :: (show (Tree (l,i,r)))
    |  showright Empty = []
  end end;
```

最後に TITEM の実現部分である structure を定義しよう．

プログラム 52: ストラクチャ IntTItem

```
structure IntTItem : TITEM = struct
  type titem = int
  fun isequal a b = a = b
  fun isless (a:titem) b = a < b
  fun isgreater (a:titem) b = a > b
  fun show (a:titem) = makestring a
end;
```

プログラム 53: ストラクチャ StringTItem

```
structure StringTItem : TITEM = struct
  type titem = string
  fun isequal a b = a = b
  fun isless (a:titem) b = a < b
  fun isgreater (a:titem) b = a > b
  fun show (a:titem) = a
end;
```

このプログラム例を以下に示す.

操作 58: 抽象データ型 IntTree の実行

```
- open IntTree;↩
open IntTree
structure I : TITEM
val empty = - : tree
val isempty = fn : tree -> bool
val cons = fn : IntTItem.titem -> tree -> tree -> tree
val left = fn : tree -> tree
val right = fn : tree -> tree
val root = fn : tree -> IntTItem.titem
val show = fn : tree -> string list
- val t = empty;↩
val t = - : tree
- implode(show t);↩
val it = "" : string
- implode(show (cons 6 empty empty));↩
val it = "(6)" : string
- implode(show (left (cons 9 (cons 5 (cons 1 empty empty) (cons 6 empty empty))↩
=    (cons 15 (cons 12 empty empty) empty))));↩
val it = "((1),5,(6))" : string
- implode(show (cons 9 (cons 5 (cons 1 empty empty) (cons 6 empty empty))↩
=    (cons 15 (cons 12 empty empty) empty)));↩
val it = "(((1),5,(6)),9,((12),15))" : string
```

6

5 章のドリル

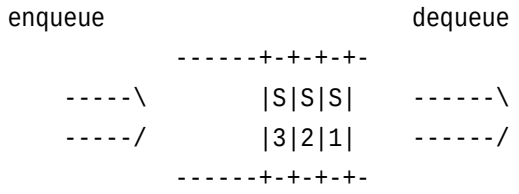
各問に答えよ。レポートは jLaTeX を使って清書し，プログラムの説明，考察をせよ．

6.1 整数を内部に持つ抽象データ型 stack

前章で紹介した抽象データ型 stack は，内部の値として string 型を扱った．これを修正して，int 型の値を扱うようにせよ．

6.2 抽象データ型キュー

キュー (queue) とは，



のような構造をしたデータで，S1，S2 の順に enqueue された要素は S1，S2 の順 (同じ順番) でのみ dequeue できない．また，キューのことを **バッファ**(buffer) ということもある．通常 enqueue できる要素は有限で，そのようなキューを**有限バッファ**という．スタックと違いこのような要素の取り出し方を FIFO(First In First Out) という．

抽象データ型キューは，以下の関数を持つ抽象データ型である．

- empty
  キューを空 (empty) にする．
- isempty
  キューが空かどうかチェックする．
- enqueue
  キューに要素を 1 つ enqueue する．
- dequeue
  キューから要素を 1 つ dequeue する．
- top
  次に dequeue するだろう要素を調べる (実際に dequeue しない)．

- length
  キューの長さ (要素の数) を返す.
- show
  キューの中身を表示する.

内部の値は，string 型とせよ.

6.3 抽象データ型ベクタ

ベクタ (vector) は，一般にはさまざまな意味で用いられているが，ここでは指定した位置に要素を出し入れすることができるデータ構造を示すことにする．抽象データ型ベクタは以下の関数を持つ抽象データ型である.

- empty
  ベクタを空にする.
- isempty
  ベクタが空かどうかチェックする.
- vector
  リストを引数として受渡し，ベクタに変換し返す.
- at
  引数として位置 (自然数) を受渡し，その位置にある要素を返す.
- length
  ベクタの長さを返す.
- show
  ベクタの中身を表示する.

内部の値は，int 型とせよ.

6.4 総称的抽象データ型キュー

総称的抽象データ型キューを定義し，実行例を示せ．実行例では，最低 (IntITEM と StringITEM に対して) 実行せよ.

6.5 総称的抽象データ型ベクタ

総称的抽象データ型ベクタを定義し，実行例を示せ．実行例では，最低 (IntITEM と StringITEM に対して) 実行せよ.

6.6 総称的抽象データ型 PhoneDB

総称的抽象データ型 PhoneDB を設計，定義し，実行例を示せ.

PhoneDB とは，名前と電話番号からなるデータベースを表現するデータ構造である．総称的抽象データ型 PhoneDB は以下の関数を持つ抽象データ型である.

- empty
  データベースを空にする.

- add
  データベースに新しい名前と電話番号を追加する (既に，同じ名前が存在した時は，番号を上書きし，異なる名前に同じ番号が登録されるようにしてもよい).
- find
  ある名前について，対応する番号をデータベースの中から探して返す (もし，その名前が存在しなかった場合は，例外 NotFound を発生させる).
- show
  名前と番号の一覧表を表示する.

6.7 ファンクタ MkApplications

総称的抽象データ型 stack の場合と同様に，tree の場合にもファンクタ MkApplications を定義しよう．このファンクタは，既存の抽象データ型 (tree) に新たな関数を付け加える.

- size
  木のノードの数を返す.
- height
  木の最大の高さを返す.

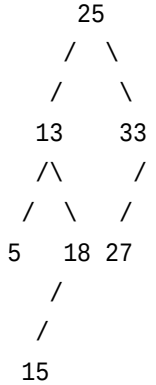
プログラム 54: ファンクタ MkApplications

```
functor MkApplications (structure T: TREE) = struct
open T
fun size t = ____①____
fun height t = ____②____
end
```

①，② の空白を埋めてプログラムを完成せよ．また，正しく実行するか実行例を示せ.

6.8 総称的抽象データ型順序付き木

順序付き木 (ordered tree) とは (2 分木とは限らないが本章では順序付き 2 分木に限る) 以下のような木構造である.



ここで，各ノードは,

「左」ノード <「自」ノード <「右」ノード

の関係を保っている．また，各ノードは順序 (この場合は，大小関係を定義できなければならないことに注意しよう).

総称的抽象データ型順序木の定義は，前章の総称的抽象データ型 2 分木をそのまま流用できる．もう既に signature TITEM は以下のように定義されていた.

プログラム 55: シグニチャ TITEM の定義

```
signature TITEM = sig
  eqtype titem
  val isequal : titem -> titem -> bool
  val isless : titem -> titem -> bool
  val isgreater : titem -> titem -> bool
  val show : titem -> string
end;
```

各要素は,「等しい」(isequal),「左が右より小さい」(isless),「左が右より大きい」(isgreater) をチェックする関数を持っていなければならなかった．整数 (IntTITEM の場合) では，このような順序が定義できることは明らかであるが，文字列 (StringTITEM の場合) でも順序が定義できる.

操作 59: 辞書式順序

```
- "a" < "b";↩
val it = true : bool
- "abc" < "abd";↩
val it = true : bool
- "abd" < "aba";↩
val it = false : bool
- "abc" < "abcde";↩
val it = true : bool
```

このように文字列に対する順序のことを**辞書式順序** (dictionary order) という.

順序付き木では，前章で紹介した関数に加えて以下の関数を定義する．関数 cons は，順序付き木では簡単ではないので削除することにする.

- insert : I.titem -> tree -> tree
  引数で指定した要素を，適当な位置に挿入する.
- remove : I.titem -> tree -> tree
  引数で指定した要素を木から削除する.
- max : tree -> I.titem
  引数で指定した木の中で最大 (順序として一番大きい要素) を取り出す．但し，削除はしない.

- `min : tree -> I.titem`
  引数で指定した木の中で最小 (順序として一番小さい要素) を取り出す．但し，削除はしない．

この定義を `signature OTREE` として定義しよう．

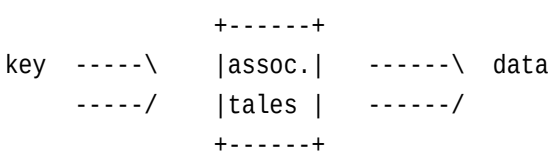
プログラム 56: シグチチャ `OTREE` の定義

```
signature OTREE = sig
  structure I : TITEM
  type otree
  val empty : otree
  val isempty : otree -> bool
  val insert : I.titem -> otree -> otree
  val left : otree -> otree
  val right : otree -> otree
  val root : otree -> I.titem
  val remove : I.titem -> otree -> otree
  val max : otree -> I.titem
  val min : otree -> I.titem
  val show : otree -> string list
end;
```

6.9 総称的抽象データ型結合表

総称的抽象データ型結合表を定義し，実行例を示せ．

結合表 (association tables)[1] とは，以下のように `key` を与えると，それに対応した `data` を返すようなデータ構造である．



プログラム 57: シグニチャ `OrderedType` の定義

```
signature OrderdType = sig
  eqtype t
  val compare: t -> t -> int
end;
```

`compare` は キーを定義域とする全域的関数で

$$\texttt{compare e1 e2} = \begin{cases} + & : \text{キー e1 は，キー e2 より大きい．} \\ 0 & : \text{キー e1 と，キー e2 は等しい．} \\ - & : \text{キー e1 は，キー e2 より小さい．} \end{cases}$$

で定義される．

[1] **辞書** (dictionary) ともいう．

プログラム 58: シグニチャ ASC の定義

```
signature ASC = sig
  structure O : Orderd Type
  type 'a t
  val empty : 'a t
  val add : key -> 'a -> 'a t -> 'a t
  val find : key -> 'a t -> 'a
  val remove : key -> 'a t -> 'a t
end
```

7

字句解析と構文解析

この章から「SML で ML を実装 (プログラム) する」という目標で議論を進める．一般に言語を解釈するためには最初に，

1. 字句解析
2. 構文解析

を処理しなければならない．その後，指定された計算機のコード (機械語または中間言語) に変換する処理系が **コンパイラ** (compiler) であり，そのまま解釈実行する処理系が **インタプリタ** (interpreter) である．本章では，この 2 つの段階を SML で書いた例を紹介する．最初に紹介する言語は，簡単に以下の**文法** (syntax) (BNF,backus normal form で定義) としよう．

```
フレーズ ::= 'let' 識別子 'be' 式 ';
   |   '式 ';'
式 ::= 項
   |   式 '+' 項
   |   式 '-' 項
項 ::= 基底式
   |   項 '*' 基底式
   |   項 '/' 基底式
基底式 ::= 整数
   |   識別子
```

7.1 字句解析

プログラム (通常，**印刷可能文字** (printable characters) を**字句** (tokens) に切り分ける処理を**字句解析** (lexical analizer) という．以下のプログラム，

```
fun twice x=x*2;
```

を `fun`，`twice`，`x`，`=`，`x`，`*`，`2`，`;` のように意味のある単位に切り分ける処理を示す．この例では `fun` は **予約語** (reserved words) または**識別子** (identifiers)，`twice`，`x` は識別子または変数，`=`，`*` は記号 (symbols)，2 は**整数** 等である．上の「意味」は実際には次段階の「構文解析」で詳しく行う．

プログラムの最初として上のトークンを分類するユーザ定義型を定義しよう．ユーザ定義型とは，`abstype` で紹介したように「〜または〜」の形をした型定義である．コンストラクタ `datatype` を使う．

プログラム 59: データ型 `rword` と `lexis`

```
datatype rword = LET | BE;
datatype lexis = Reserved of rword | Identifier of string | Num of int
  | One of string;
```

この lexis 型は，string 型の値を持った Identifier か，int 型の値を持った Num かまたは，string 型を持った One かのどれかの値を持つことを示す．各値は Identifier("fun")，Num(1001)，One(";") のように表現する．要は，プログラムをこの lexis 型の値に分解すればよい．字句解析を行う関数を token としよう．

プログラム 60: 関数 token(1)

```
fun digit c = ("0" <= c andalso c <= "9");
fun small_alpha c = ("a" <= c andalso c <= "z");
fun large_alpha c = ("A" <= c andalso c <= "Z");
fun integer ISTREAM i = if digit (lookahead ISTREAM) then
    integer ISTREAM (10*i+ord(input(ISTREAM,1))-ord("0"))
  else i
and identifier ISTREAM id = let val c = lookahead ISTREAM in
    if (small_alpha c) orelse (large_alpha c) orelse (digit c)
      orelse c = "_" then identifier ISTREAM (id^(input (ISTREAM,1)))
    else id
  end
and native_token ISTREAM = let val c = lookahead ISTREAM in
    if (small_alpha c) orelse (large_alpha c) then
      let val id = identifier ISTREAM "" in
        case id of
          "let" => Reserved(LET)
        | "be" => Reserved(BE)
        | _ => Identifier(id)
      end
    else if digit c then Num(integer ISTREAM 0)
    else One(input(ISTREAM,1))
  end
and token ISTREAM = let val c = native_token ISTREAM in
    case c of
      One(" ") => token ISTREAM
    | One("\t") => token ISTREAM
    | One("\n") => token ISTREAM
    | _ => c
  end;
```

ここで関数 lookahead は，引数で指定した入力ストリーム ISTREAM から「次に取り出されるだろう文字を，実際には取り出さずに覗く」機能を果たす．この処理のことを **先読み** という．

datatype によるユーザ定義型は「〜または〜」という**バリアント型** (variant types) であった (C 言語での union 型または列挙型に相当)．これに対して **レコード型** (record types) は,「〜かつ〜」という形をした型である (後述)．バリアント型は同時に一つの値しか示すことができないので，処理する場合には「場合分け」が必要である．case ··· of ··· 式を使う．以下の形をしている．

プログラム 61: case ··· of ···

```
case 式 of
  パターン 1 => 式
  パターン 2 => 式
  ...
  パターン n => 式
```

パターンとして「その他」を示す場合には _ を使う．上の例では，関数 native_token の定義中で指定している．上のプログラムを実行すると，

操作 60: 関数 token の実行

```
- token std_in;↩
+↩
val it = One "+" : lexis
- token std_in;↩
1↩
val it = Num 1 : lexis
- token std_in;↩
abd↩
val it = Identifier "abd" : lexis
```

のようになる．しかし，関数 token は 1 トークン取り出す毎に一回呼出されるのみである．そこでこの関数を連続して呼び出すよう変更しよう．以下の関数 run() を用意する．

プログラム 62: 字句解析のための関数 run

```
fun print_rword x = case x of
  LET => print "LET"
| BE => print "BE";
fun print_token x = case x of
  Identifier(i) => ( print "Identifier("; print i; print ")" )
| Reserved(s) => ( print "Reserved("; print_rword s; print ")" )
| Num(n) => ( print "Num("; print n; print ")" )
| One(one) => ( print "One("; print one; print ")" );
exception End_of_system;
fun run() =
  let val stream_of_enter = std_in in
    while true do (
      flush_out std_out;
      let val result = token stream_of_enter in
        case result of
          Identifier("Quit") => raise End_of_system
        |  _ => ( print_token result; print "\n" )
      end )
  end;
```

関数 token から返る lexis 型の値は，そのままでは表示できない (直接呼出す場合はシステムが表示) ので表示するための関数 print_token，print_rword を定義している．関数 run の中では，関数 token を連続的に呼び出すために while 式を使っている[1]．この関数を実行すると以下のようになる．

操作 61: 関数 run() の実行

```
- run();↩
1+2;↩
Num(1)
One(+)
Num(2)
One(;)
let x be 10;↩
Reserved(LET)
Identifier(x)
Reserved(BE)
Num(10)
One(;)
Quit↩

uncaught exception End_of_system
```

7.2 構文解析

構文解析とは，字句解析から受渡されるトークンの組み合わせを文法の意味にマッチさせ，結果として木構造 (**構文木** (parser trees) という) に変換する処理を示す．ここで扱う文法は以下のようであった．

フレーズ ::= 式 ';'	Expr(式)
| 'let' 識別子 'be' 式 ';'	Def(識別子, 式)
式 ::= 項	項
| 式 '+' 項	Application(Variable +, Pair(式, 項))
| 式 '-' 項	Application(Variable -, Pair(式, 項))
項 ::= 基底式	基底式
| 項 '*' 基底式	Application(Variable *, Pair(項, 基底式))
| 項 '/' 基底式	Application(Variable /, Pair(項, 基底式))
基底式 ::= 整数	Number(トークン)
| 識別子	Variable(トークン)

一般に構文解析は (文法の) 木構造のルートから解析する**トップダウンパーサ** (top down parser) と，リーフから解析する **ボトムアップ パーサ** (bottom up parser) の 2 種類がある．本章ではトップダウンパーサを用いた構文解析を紹介する．

上の文法は「1 文字先読みすれば，曖昧なく構文解析できる」であった．そのため構文解析を行うためには「先読みトークン」が必要になる．構文解析処理の中では

(先読みトークン，構文木)

の **組** (pair) を受渡す．最初に構文木を定義するためのユーザ定義型を定義しよう．構文木とは，文法の「要素を成す意味」(**終端記号** (terminal symbol) (トークンのことを終端記号ともいう，構文木のリーフに相当) または **非終端**

[1]実は SML にも「繰り返し」を行う while が用意されている．しかし，実際には後述する「参照型」を使う．よって，純粋な「関数プログラミング」の範疇ではないので注意しよう

記号 (non-terminal symbol)(構文木のリーフ以外のノードに相当) からなる木である．よって 型 definition(‘定義’ の意味) と 型 expr (‘式’ の意味) を定義する．

プログラム 63: データ型 definition と expr

```
datatype definition = Def of string*expr | Expr of expr
and expr = Number of int | Variable of string | Application of expr*expr
  |  Pair of expr*expr;
```

操作 62: 構文木

```
## let x be 10;↩
Def ( x, Number 10 )
## 1+2;↩
Expr ( Application ( Variable +, Pair ( Number 1, Number 2 ) ) )
## let y be x+10/2;↩
Def ( y, Application ( Variable +, Pair ( Variable x, Application (
Variable /, Pair ( Number 10, Number 2 ) ) ) ) )
```

例えば，上のように let x be 10; というプログラムを入力すれば Def (x, Number 10) という構文木に変換し，1+2 というプログラムを入力すれば Expr (Application (Variable +, Pair (Number 1, Number 2))) という構文木に変換する．また let y be x+10/2; というプログラムを入力すれば Def (y, Application (Variable +, Pair (Variable x, Application (Variable /, Pair (Number 10, Number 2))))) という構文木に変換する．Def (x, Number 10) を実際に木構造に表現すると以下のようになる．

```
    Def
    /\
   /  \
  x  Number 10
```

Expr (Application (Variable +, Pair (Number 1, Number 2))) は,

```
    Expr
     |
     |
    Application
    /       \
   /         \
Variable +    Pair
             /  \
            /    \
      Number 1  Number 2
```

の木構造となり, Def (y, Application (Variable +, Pair (Variable x, Application (Variable /, Pair (Number 10, Number 2))))) は,

```
    Def
    /\
   /  \
```

```
  y   Application
       /       \
      /         \
   Variable +   Pair
                /   \
               /     \
      Variable x   Application
                    /      \
                   /        \
             Variable /     Pair
                            /   \
                           /     \
                    Number 10   Number 2
```

となる.

プログラム 64: 関数 phrase

```
exception Syntax_error;
fun phrase ISTREAM =
  let val (ahead,def) = definition ISTREAM (token ISTREAM) in
    if ahead <> One(";") then raise Syntax_error else def end
and definition ISTREAM ahead =
  case ahead of
    Reserved(LET) =>
      let val ahead = token ISTREAM in
        case ahead of
          Identifier(i) =>
            let val ahead = token ISTREAM in
              case ahead of
                Reserved(BE) =>
                  let val (ahead,exp) = expr ISTREAM (token ISTREAM) in
                    (ahead,Def(i,exp))
                  end
              | _ => raise Syntax_error
            end
        | _ => raise Syntax_error
      end
  | _ => let val (ahead,exp) = (expr ISTREAM ahead) in (ahead,Expr exp) end
and expr ISTREAM ahead = read_operation ISTREAM ahead expr1 ["+","-"]
and expr1 ISTREAM ahead = read_operation ISTREAM ahead expr0 ["*","/"]
and expr0 ISTREAM ahead = if check_expr_simple ahead then expr_simple ISTREAM ahead
  else raise Syntax_error
and check_expr_simple ahead =
  case ahead of
    Identifier(i) => true
  | Num(n) => true
  | _ => false
and expr_simple ISTREAM ahead =
  case ahead of
    Identifier(i) => ((token ISTREAM), Variable i)
  | Num(n) => ((token ISTREAM), Number n);
```

ここで 関数 check_expr_simple は「基底式」を先読みし，Identifier または Num ならば true を返し，そうでなければ false を返す．関数 expr0 では check_expr_simple が ture を返す場合のみ[2]expr_simple を呼出している．関数 definition では let 識別子 be 式 の構文解析を処理するが，Reserved(LET)，Identifier(i)，Reserved(BE)，式 (expr) の順で呼出していることがわかる．この中で用いている関数 read_operation は以下で定義する関数である．

[2]つまり，先読みした結果 '式' として評価しなければならない場合

プログラム 65: 関数 read_operation

```
fun mem x (front::rest) = if x = front then true else mem x rest
|  mem x [] = false;
fun read_operator oper operators =
   case oper of
     One(one) => if (mem one operators) then one else ""
   | _ => "";
fun read_operation ISTREAM ahead read_base operators =
   let fun read_rest ISTREAM a1 e1 =
     let val oper_str = read_operator a1 operators in
       if oper_str = "" then (a1,e1)
       else let val (a2,e2) = read_base ISTREAM (token ISTREAM) in
         read_rest ISTREAM a2 (Application(Variable oper_str,Pair(e1,e2))) end
     end in
       let val (ahead,exp) = read_base ISTREAM ahead in
         read_rest ISTREAM ahead exp
       end
   end;
```

上の関数 phrase を実行すると,

操作 63: 関数 phrase の実行

```
- phrase std_in;↩
let x be 20;↩
val it = Def ("x",Number 20) : definition
- phrase std_in;↩
1+2;↩
val it = Expr (Application (Variable "+",Pair (#,#))) : definition
```

となる. 値の表示に関して表示が長くなる場合は # により省略されていることに注意しよう. より発展させて, 連続的にプログラムのフレーズ (phrase) を受付けるようにしよう. そのために 関数 token の場合と同様に, 関数 run を用意する.

プログラム 66: 構文解析のための関数 run

```
exception End_of_system;
fun run() = let val stream_of_enter = std_in in
    while true do (
      print("## "); flush_out std_out;
      let val result = phrase stream_of_enter in
        case result of
          Expr(Variable("Quit")) => raise End_of_system
        | _ => ( print_definition result; print "\n" )
      end )
  end;
```

この関数 run を実行すると以下のようになる.

プログラム 67: 関数 run の実行

```
- run();↩
## let y be 10;↩
Def ( y, Number 10 )
## y;↩
Expr ( Variable y )
## 3+4;↩
Expr ( Application ( Variable +, Pair ( Number 3, Number 4 ) ) )
## let x be 10/20;↩
Def ( x, Application ( Variable /, Pair ( Number 10, Number 20 ) ) )
## Quit;↩

uncaught exception End_of_system
```

より本格的に動作するためにプロンプト ## を表示するようにした. 関数 token の場合と同じように関数 run() を実行する場合は, 構文木を表示する関数を定義しなければならない. 以下の関数が処理する.

プログラム 68: 関数 print_definition と print_expr

```
fun print_definition x = case x of
  Def(s,e) => ( print "Def ( "; print s; print ", "; print_expr e; print " )" )
| Expr(e) => ( print "Expr ( "; print_expr e; print " )" )
and print_expr x = case x of
  Variable s => ( print "Variable "; print s )
| Application(e1,e2) => ( print "Application ( "; print_expr e1;
    print ", "; print_expr e2; print " )" )
| Pair(e1,e2) => ( print "Pair ( "; print_expr e1; print ", ";
    print_expr e2; print " )" )
| Number n => ( print "Number "; print n );
```

7.3 文法を改良する

では, 文法をより ML に近づけるため改良しよう.

```
フレーズ ::= 'let' {'rec'} 識別子 '=' 式 ';'   Def( フラグ, 識別子, 式 )
   |   '式 ';'                                    Expr( 式 )
式  ::= 式4                                       式4
   |    'function' モチーフ                       Function ( モチーフ )
   |    'let' {'rec'} 識別子 '=' 式 'in' 式   Let( Def( フラグ, 識別子, 式 ), 式 )
式4 ::= 式3                                        式3
   |    式4 ',' 式3                               Pair( 式4, 式3 )
式3 ::= 式2                                        式2
   |    式3 '=' 式2                    Application( Variable =, Pair( 式3, 式2 ) )
   |    式3 NE 式2                     Application( Variable NE, Pair( 式3, 式2 ) )
   |    式3 '>' 式2                    Application( Variable >, Pair( 式3, 式2 ) )
```

```
   |    式3 '<' 式2                  Application( Variable <, Pair( 式3, 式2 ) )
   |    式3 GE 式2                   Application( Variable GE, Pair( 式3, 式2 ) )
   |    式3 LE 式2                   Application( Variable LE, Pair( 式3, 式2 ) )
式2 ::= 式1                          式1
   |    式2 '+' 式1                  Application( Variable +, Pair( 式2, 式1 ) )
   |    式2 '-' 式1                  Application( Variable -, Pair( 式2, 式1 ) )
式1 ::= 式0                          式0
   |    式1 '*' 式0                  Application( Variable *, Pair( 式1, 式0 ) )
   |    式1 '/' 式0                  Application( Variable /, Pair( 式1, 式0 ) )
式0 ::= 基底式                       基底式
   |    式0 基底式                   Application( 式0, 基底式 )
基底式 ::= 整数                      Number(トークン)
   |   識別子                        Variable(トークン)
   |   論理値                        Boolean(トークン)
   |   '(' 式 ')'                    式
   |   '()'                          Unit
モチーフ ::= モチーフ2               [モチーフ2]
   |   モチーフ '|' モチーフ2        モチーフ2 :: モチーフ
モチーフ2 ::= モチーフ1 '->' 式      (モチーフ1, 式)
モチーフ1 ::= モチーフ0              モチーフ0
   |   モチーフ1 ',' モチーフ0       Motif_pair(モチーフ1, モチーフ0)
モチーフ0 ::= 整数                   Motif_number(トークン)
   |   識別子                        Motif_variable(トークン)
   |   論理値                        Motif_boolean(トークン)
   |   '()'                          Motif_unit
```

式0は結果として「基底式の並び (1 個以上)」となる.

```
f x 1 10;
```

のように，式を並べることにより ML では「適用」を表すためである．BNF による文法の定義の中で { } により括られた部分はオプショナルを示す.

まず字句解析を処理する関数 token を改良するために，ユーザ定義型を変更する.

プログラム 69: データ型 rword と lexis

```
datatype rword = LET | TRUE | FALSE | IN | FUNCTION | REC
  | NE | LE | GE | ARROW;
datatype lexis = Reserved of rword | Identifier of string | Num of int
  | One of string;
```

予約語のための型 rword の要素が増えている．ARROW，NE，GE，LE などは 2 文字以上の記号から成るので新たに予約語として扱う.

関数 token を改良すると以下のようになる.

プログラム 70: 関数 token(2)

```
fun digit c = ("0" <= c andalso c <= "9");
fun small_alpha c = ("a" <= c andalso c <= "z");
fun large_alpha c = ("A" <= c andalso c <= "Z");
fun mem x (front::rest) = if x = front then true else mem x rest
|  mem x [] = false;
fun integer ISTREAM i = if digit (lookahead ISTREAM) then
     integer ISTREAM (10*i+ord(input(ISTREAM,1))-ord("0")) else i
and identifier ISTREAM id = let val c = lookahead ISTREAM in
     if (small_alpha c) orelse (large_alpha c) orelse (digit c)
       orelse c = "_" then identifier ISTREAM (id^(input (ISTREAM,1)))
     else id end
and native_token ISTREAM = let val c = lookahead ISTREAM in
   if (small_alpha c) orelse (large_alpha c) then
     let val id = identifier ISTREAM "" in
       case id of
         "let" => Reserved(LET)
       | "in" => Reserved(IN)
       | "true" => Reserved(TRUE)
       | "false" => Reserved(FALSE)
       | "function" => Reserved(FUNCTION)
       | "rec" => Reserved(REC)
       | _ => Identifier(id)
     end
   else if digit c then Num(integer ISTREAM 0)
   else if operator c then
     let val oper = input(ISTREAM,1) in
       let val n = lookahead ISTREAM in
         if oper = "<" andalso n = ">" then ( input(ISTREAM,1); Reserved(NE) )
         else if oper = "<" andalso n = "=" then ( input(ISTREAM,1);
           Reserved(LE) )
         else if oper = ">" andalso n = "=" then ( input(ISTREAM,1);
           Reserved(GE) )
         else if oper = "-" andalso n = ">" then ( input(ISTREAM,1);
           Reserved(ARROW) )
         else One(oper)
       end end
   else One(input(ISTREAM,1)) end
and operator c = mem c ["<",">","-"]
and token ISTREAM = let val c = native_token ISTREAM in
   case c of
     One(" ") => token ISTREAM
   | One("\t") => token ISTREAM
   | One("\n") => token ISTREAM
   | _ => c end;
```

関数 run は前回と同様に定義できるので省略する．このプログラムを実行すると以下のようになる．

プログラム 71: 関数 token の実行

```
- run();↩
let x = function x->x+1;↩
Reserved(LET)
Identifier(x)
One(=)
Reserved(FUNCTION)
Identifier(x)
Reserved(->)
Identifier(x)
One(+)
Num(1)
One(;)
let x = 10 in x;↩
Reserved(LET)
Identifier(x)
One(=)
Num(10)
Reserved(IN)
Identifier(x)
One(;)
Quit↩

uncaught exception End_of_system
```

また，前回と同様に関数 print_rword と print_token を定義しなければならない．

プログラム 72: 関数 print_rword

```
fun print_rword x = case x of
  LET => print "LET"
| TRUE => print "TRUE"
| FALSE => print "FALSE"
| IN => print "IN"
| FUNCTION => print "FUNCTION"
| REC => print "REC"
| NE => print "<>"
| LE => print "<="
| GE => print ">="
| ARROW => print "->"
fun print_token x = case x of
  Identifier(i) => ( print "Identifier("; print i; print ")" )
| Reserved(s) => ( print "Reserved("; print_rword s; print ")" )
| Num(n) => ( print "Num("; print n; print ")" )
| One(one) => ( print "One("; print one; print ")" );
```

構文解析以降の議論は，次章で行う．

8

7 章のドリル

8.1 四則演算に関する文法

前章の文法の中で四則演算に関する文法について考える.

1. 前章の文法は,

```
式 ::= 項
    |   式 '+' 項
    |   式 '-' 項
項 ::= 基底式
    |   項 '*' 基底式
    |   項 '/' 基底式
```

   であった. この定義を以下のように変更した場合, 構文としての意味として異なるかを 実際に ML でプログラミングして考察せよ.
2. 変更した文法

```
式 ::= 基底式
    |   式 '+' 基底式
    |   式 '-' 基底式
    |   式 '*' 基底式
    |   式 '/' 基底式
```

8.2 「改良した」文法の構文解析

「改良した」文法の構文解析を行う関数を紹介する。関数名は, 同じ phrase であるが以下の文法が異なる.

- function 式の追加 (再帰定義も可能)
- let···in 式の追加
- 組式の追加
- 論理型の式の追加 (それに伴った演算子の追加)
- 基底式の追加 (適用が可能となる)
- その他

最初にデータ型 definition と expr をこの文法に合うよう再定義しよう. 以下のプログラムの ⓪〜⑯に適当なプログラムを入れ完成せよ.

プログラム 73: データ型 definition, expr と motif

```
datatype definition = Def of ⓪
  |  Expr of expr
and expr = Number of int
  |  Variable of string
  |  Application of expr*expr
  |  Pair of expr*expr
  |  Boolean of bool
  |  Let of definition*expr
  |  Function of (motif*expr) list
  |  Unit
and motif = Motif_variable of string
  |  Motif_boolean of bool
  |  Motif_number of int
  |  Motif_pair of motif*motif
  |  Motif_unit
  |  Motif_arrow;
```

データ型 definition では Def の場合，再帰定義 (rec を付ける場合) か，そうでないかのフラグのために 3 つの値の組としている．データ型 expr では Boolean，Let，Function，Unit が加わっている．データ型 motif は，関数定義の際 (function) のパターンを示す型である．

関数 phrase の定義は，その後の関数 definition，expr 等の関数と共に相互再帰定義を行うが，最初の部分は「改良前」と同じである．しかし，新たに括弧 (()) の対応をチェックするための例外処理を加える．

プログラム 74: 関数 phrase の最初の部分

```
exception Cant_close_parenthesis;
fun phrase ISTREAM = let val (ahead,def) = definition ISTREAM (token ISTREAM) in
    if ahead <> One(";") then raise Syntax_error
    else def end
```

関数 phrase の定義の後には関数定義が続くためセミコロン (;) がついていないことに注意しよう．またそれらの関数は「継続」のため and を用いて定義する．

プログラム 75: 関数 definition の定義

```
and definition ISTREAM ahead =
  case ahead of
    Reserved(LET) =>
      let val ahead = ________①________ in
        case ahead of
          Reserved(REC) =>
            let val ahead = ________②_________ in
              case ahead of
                Identifier(i) =>
                  let val ahead = _______③_______ in
                    case ahead of
                      One("=") =>
                        let val (ahead,e4) = expr ISTREAM (token ISTREAM) in
                        if ahead <> Reserved(IN) then
                          ________④_________
                        else let val (ahead,e5) = expr ISTREAM (token ISTREAM) in
                          (ahead,Expr(Let(Def(true,i,e4),e5)))
                        end end
                    | _ => raise Syntax_error
                  end
              | _ => raise Syntax_error
          end
        | Identifier(i) =>
            let val ahead = token ISTREAM in
              case ahead of
                One("=") =>
                  let val (ahead,e7) = expr ISTREAM (token ISTREAM) in
                    if ahead <> Reserved(IN) then
                      ________⑤_________
                    else
                      let val (ahead,e8) = expr ISTREAM (token ISTREAM) in
                        ________⑥_________
                      end
                  end
              | _ => raise Syntax_error
            end
        | _ => raise Syntax_error
    end
  | _ => let val (ahead,exp) = (expr ISTREAM ahead) in (ahead,Expr exp) end
```

関数 definition では「let 識別子 = 式」と「let 識別子 = 式 in 式」の両方の文法について処理していることに注意しよう．また，オプショナルの rec の定義について「ある場合」と「ない場合」について分けて処理している．

次に関数 expr の定義を示す．

プログラム 76: 関数 expr の定義

```
and expr ISTREAM ahead =
  case ahead of
    Reserved(FUNCTION) =>
      let val (ahead,motif) = read_list_of_case ISTREAM (token ISTREAM) in
        (ahead,Function motif)
      end
  | _ => expr4 ISTREAM ahead
and expr4 ISTREAM ahead = read_infix ISTREAM ahead expr3 ","
  (fn e1 => fn e2 => Pair(e1,e2))
and expr3 ISTREAM ahead = read_operation ISTREAM ahead expr2
  ["=","<>","<",">","<=",">="]
and expr2 ISTREAM ahead = read_operation ISTREAM ahead ____⑦____ ["+","-"]
and expr1 ISTREAM ahead = read_operation ISTREAM ahead expr0 ["*","/"]
and expr0 ISTREAM ahead = if check_expr_simple ahead then
    let val (ahead,exp) = expr_simple ISTREAM ahead in
      if check_expr_simple ahead then sequence_of_application ISTREAM ahead exp
      else (ahead,exp)
    end
  else if ahead = One(")") then (ahead,Unit) else raise Syntax_error
and sequence_of_application ISTREAM ahead f =
  let val (ahead,exp) = expr_simple ISTREAM ahead in
    if check_expr_simple ahead then
      sequence_of_application ISTREAM ahead (_____⑧_____)
    else (ahead,_____⑧_____)
  end
and check_expr_simple ahead =
  case ahead of
    Identifier(i) => true
  | Num(n) => true
  | _____⑨_____
  | Reserved(FALSE) => true
  | One("(") => true
  | _ => false
and expr_simple ISTREAM ahead =
  case ahead of
    Identifier(i) => ((token ISTREAM), Variable i)
  | Num(n) => ((token ISTREAM),Number n)
  | Reserved(TRUE) => ((token ISTREAM), Boolean true)
  | Reserved(FALSE) => ((token ISTREAM), Boolean false)
  | One("(") =>
      let val (ahead,exp) = expr ISTREAM (token ISTREAM) in
        case ahead of
          One(")") => if exp = Unit then ((token ISTREAM),Unit)
            else ((token ISTREAM),exp)
        | _ => raise Cant_close_parenthesis
      end
```

文法の中の `Reserved(FUNCTION)` の定義では，新たにモチーフを処理する関数 `read_list_of_case` を呼出している．

プログラム 77: 関数 motif

```
and read_list_of_case ISTREAM ahead =
  let fun other_case ISTREAM a1 =
    let val oper_str = read_operator a1 ["|"] in
      if oper_str <> "|" then (a1,[])
      else let val (a2, m2)= motif ISTREAM (token ISTREAM) in
        let val oper_str = read_operator a2 ["->"] in
          if oper_str <> "->" then raise Syntax_error
          else let val (a3,e3) = expr ISTREAM (token ISTREAM) in
            let val (a4,e4) = other_case ISTREAM a3 in
              (a4,( ⑩ , ⑪ )::e4)
          end end
      end end
  end in
    let val (a2,m2) = motif ISTREAM ahead in
      let val oper_str = read_operator a2 ["->"] in
        if oper_str <> "->" then raise Syntax_error
        else let val (a3,e3) = expr ISTREAM (token ISTREAM) in
          let val (a4,e4) = other_case ISTREAM a3 in
            (a4,(m2,e3)::e4)
        end end
      end
    end
  end
and motif ISTREAM ahead = read_infix ISTREAM ahead seq_of_motif ","
  (fn m1 => fn m2 => Motif_pair(m1,m2))
and seq_of_motif ISTREAM ahead =
  let val (a1,l) = sequence_of_motif ISTREAM ahead [] in
    (a1,hd l)
  end
and sequence_of_motif ISTREAM ahead l =
  let val (ahead,exp) = motif_simple ISTREAM ahead in
    if exp <> Motif_arrow then
      sequence_of_motif ISTREAM (token ISTREAM) (exp::l)
    else (ahead,l)
  end
and motif_simple ISTREAM ahead =
  case ahead of
    Identifier(i) => (ahead,Motif_variable i)
  | Num(n) => (ahead,Motif_number n)
  | Reserved(TRUE) => (ahead, Motif_boolean true)
  | Reserved(FALSE) => (ahead, Motif_boolean false)
  | _ => (ahead, Motif_arrow);
```

以上の関数を**相互再帰** (mutual recursion) で定義するためすべて and をつけて定義する．また，上の定義では前述の関数 read_operation に加え，関数 read_operator と関数 read_infix を呼出している．

プログラム 78: 関数 read_operation，read_operator と read_infix

```
fun read_operator oper operators =
  case oper of
    One(one) => if (mem one operators) then one else ""
  | Reserved(r) => ( case r of
        GE => if (mem ">=" operators) then ">=" else ""
      | ____________⑫____________
      | ____________⑬____________
      | ARROW => if (mem "->" operators) then "->" else ""
      | _ => "" )
  | _ => "";
fun read_operation ISTREAM ahead read_base operators =
  let fun read_rest ISTREAM a1 e1 =
    let val oper_str = read_operator a1 operators in
      if oper_str = "" then (a1,e1)
      else let val (a2,e2) = read_base ISTREAM (token ISTREAM) in
        read_rest ISTREAM a2 (Application(Variable oper_str,Pair(e1,e2)))
      end
    end in
      let val (ahead,exp) = read_base ISTREAM ahead in
        read_rest ISTREAM ahead exp
      end
  end;
fun read_infix ISTREAM ahead read_base inf construct_syntax =
  let fun read_start ISTREAM a1 =
    let val (a3,e3) = read_base ISTREAM a1 in
      let val oper_str = read_operator a3 [inf] in
        if oper_str = "" then (a3,e3)
        else let val (a5,e5)= read_start ISTREAM (token ISTREAM) in
          (a5, construct_syntax ⑭ ⑮ ) end
      end
    end in read_start ISTREAM ahead end;
```

関数 run については前章の run と同一なので省略する．構文木を表示する関数群についてはあらたに定義しなければならない．

プログラム 79: 関数 print_definition, prnit_expr, print_case_list と print_motif

```
fun print_definition x = case x of
  Def( b, s,e) => ( print "Def ( "; print b; print ", ";
    print s; print ", "; print_expr e; print " )" )
| Expr(e) => ( print "Expr ( "; print_expr e; print " )" )
and print_expr x = case x of
  Variable s => ( print "Variable "; print s )
| Application(e1,e2) => ( print "Application ( "; print_expr e1;
    print ", "; print_expr e2; print " )" )
|   Pair(e1,e2) => ( print "Pair ( "; print_expr e1; print ", ";
    print_expr e2; print " )" )
| Let(d,e) => ( print "Let ( "; print_definition d; print ", ";
    print_expr e; print " )" )
| Number n => ( print "Number "; print n )
| Function cl => ⑯
| Unit => ( print "Unit" )
| Boolean b => ( print "Boolean "; print b )
and print_case_list ((m,e)::[]) = ( print "( "; print_motif m; print ", ";
    print_expr e; print " )" )
| print_case_list ((m,e)::rest) = ( print "( "; print_motif m; print ", ";
    print_expr e; print " ) | "; print_case_list rest )
| print_case_list [] = ()
and print_motif x = case x of
  Motif_variable v => ( print "Motif_variable \""; print v; print "\"" )
| Motif_boolean b => ( print "Motif_boolean "; print b )
| Motif_number n => ( print "Motif_number "; print n )
| Motif_pair (m1,m2) => ( print "Motif_pair ( "; print_motif m1;
    print ", "; print_motif m2; print " )" )
| Motif_unit => print "Motif_unit"
| Motif_arrow => print "Motif_arrow"
```

最後に以上のプログラムを実行した例を示す.

プログラム 80: 実行例

```
- run();↩
## let f = function x->1|y->y;↩
Def ( false, f, Function ( ( Motif_variable "x", Number 1 ) | (
Motif_variable "y", Variable y ) ) )
## let x = 1;↩
Def ( false, x, Number 1 )
## let y = 10 in y+1;↩
Expr ( Let ( Def ( false, y, Number 10 ), Application ( Variable +,
Pair ( Variable y, Number 1 ) ) ) )
## 10,20;↩
Expr ( Pair ( Number 10, Number 20 ) )
## 10 <= 30;↩
Expr ( Application ( Variable <=, Pair ( Number 10, Number 30 ) ) )
## (1);↩
Expr ( Number 1 )
## let rec fac = function 1->1|n->n*(fac (n-1));↩
Def ( true, fac, Function ( ( Motif_number 1, Number 1 ) | (
Motif_variable "n", Application ( Variable *, Pair ( Variable n,
Application ( Variable fac, Application ( Variable -, Pair ( Variable
n, Number 1 ) ) ) ) ) ) ) )
## (x+y);↩
Expr ( Application ( Variable +, Pair ( Variable x, Variable y ) ) )
## x y z;↩
Expr ( Application ( Application ( Variable x, Variable y ), Variable
z ) )
```

8.3 if···then···else と文字列の値を追加する

if···then···else 式と，文字列型の値を追加した以下の文法の字句解析と構文解析のプログラムを実装せよ．添付するプログラムは，テキストで示したプログラムとの差分のみを示せ．

フレーズ ::= 'let' {'rec'} 識別子 '=' 式 ';'	Def(フラグ, 識別子, 式)
| 式 ';'	Expr(式)
式 ::= 式 4	
| 'function' モチーフ	Function (モチーフ)
| 'let' {'rec'} 識別子 '=' 式 'in' 式	Let(Def(フラグ, 識別子, 式), 式)
| 'if' 式 'then' 式 'else' 式	If(式, 式, 式)

と，

式 0 ::= 基底式	基底式
| 式 0 基底式	Application(式 0, 基底式)
基底式 ::= 整数	Number(トークン)
| 識別子	Variable(トークン)
| 論理値	Boolean(トークン)
| '(' 式 ')'	式

```
   |   '()'                                  Unit
   |   文字列                                 String(トークン)
```

の部分でアンダーラインを引いた部分の文法を変更せよ.

8.4 リストの値を追加する

今までの文法にリスト型の値を追加しよう.

```
フレーズ ::= 'let' {'rec'} 識別子 '=' 式 ';'  Def( フラグ, 識別子, 式 )
   |   式 ';'                                  Expr( 式 )
式  ::= 式4
   |    'function' モチーフ                    Function ( モチーフ )
   |    'let' {'rec'} 識別子 '=' 式 'in' 式    Let( Def( 識別子, 式 ), 式 )
   |    'if' 式 'then' 式 'else' 式            If( 式, 式, 式 )
式5 ::= 式4                                    式4
   |    式5 ',' 式4                            Pair( 式5, 式4 )
式4 ::= 式3                                    式3
   |    式4 '::' 式3                           Cons( 式4, 式3 )
式3 ::= 式2                                    式2
   |    式3 '=' 式2                           Application( Variable =, Pair( 式3, 式2 ) )
   |    式3 NE 式2                            Application( Variable NE, Pair( 式3, 式2 ) )
   |    式3 '>' 式2                           Application( Variable >, Pair( 式3, 式2 ) )
   |    式3 '<' 式2                           Application( Variable <, Pair( 式3, 式2 ) )
   |    式3 GE 式2                            Application( Variable GE, Pair( 式3, 式2 ) )
   |    式3 LE 式2                            Application( Variable LE, Pair( 式3, 式2 ) )
式2 ::= 式1                                    式1
   |    式2 '+' 式1                           Application( Variable +, Pair( 式2, 式1 ) )
   |    式2 '-' 式1                           Application( Variable -, Pair( 式2, 式1 ) )
式1 ::= 式0                                    式0
   |    式0 '*' 式1                           Application( Variable *, Pair( 式0, 式1 ) )
   |    式0 '/' 式1                           Application( Variable /, Pair( 式0, 式1 ) )
式0 ::= 基底式                                 基底式
   |    式0 基底式                             Application( 式0, 基底式 )
基底式 ::= 整数                                Number(トークン)
   |   識別子                                  Variable(トークン)
   |   論理値                                  Boolean(トークン)
   |   '(' 式 ')'                              式
   |   '()'                                    Unit
   |   '[]'                                    Nil
モチーフ ::= モチーフ3                         [モチーフ3]
   |   モチーフ '|' モチーフ3                  モチーフ3 :: モチーフ
モチーフ3 ::= モチーフ2 '->' 式                (モチーフ2, 式)
モチーフ2 ::= モチーフ1                        モチーフ1
   |   モチーフ2 ',' モチーフ1                 Motif_pair(モチーフ2, モチーフ1)
モチーフ1 ::= モチーフ0                        モチーフ0
   |   モチーフ1 '::' モチーフ0                Motif_cons(モチーフ1, モチーフ0)
```

```
モチーフ 0 ::= 整数                    Motif_number(トークン)
   |   識別子                          Motif_variable(トークン)
   |   論理値                          Motif_boolean(トークン)
   |   '()'                            Motif_unit
   |   '[]'                            Motif_nil
```

8.5 構文木の改良

ここで扱った構文木は，抽象データ型ではない．そこで構文木を総称的抽象データ型で定義し，同じ機能を保つよう改良せよ．添付するプログラムは，前問までのプログラムとの差分で良い．

9

意味定義

大抵のプログラムは，プログラム全体を用意しなくても，その断片でなんらかの意味を持つ．例えば「変数 x」に注目し「値 10」を持つとか「値 2.3」を持つとか，その実行の時点，時点で意味も異なる．まず 「プログラム断片の意味」をその「値」として定義してみよう[1]．そして ⟦ x ⟧ と書き「変数 x の意味」を表す．つまり，関数 ⟦ ⟧ は，プログラム断片を引数として受渡されその意味を返す関数と考える．プログラム (式，Exp)，意味 (値，Val) を使って以下のように定義することができる．

$$⟦\ ⟧ : Exp \rightarrow Val$$

C 言語なら `int` 型変数 `x`，`float` 型変数 `y` に対して，

$$⟦\ \texttt{x=10}\ ⟧ = 10,\ ⟦\ \texttt{x=y=30}\ ⟧ = ⟦\ \texttt{x=(y=30)}\ ⟧ = ⟦\ \texttt{y=30}\ ⟧ = 30$$

$$⟦\ \texttt{x=10, y=20}\ ⟧ = 20$$

など書くことができる[2]．プログラム断片の意味を定義することにより，より大きなプログラムの意味を定義することができ，ひいては，プログラム全体のある性質 (意味) を定義することができる．では ML のプログラムの意味を定義してみよう[3]．

`10`，`2.3`，`"str"` などの定数 (c とおく) は $⟦\ c\ ⟧ = c$．変数を代表して x とすると $⟦\ \texttt{val}\ x\ \texttt{=}\ c\ ⟧ = c$．しかし，$⟦\ x\ ⟧$ のみでは，変数 x が，

```
- val x = 20;
```

などと，意味を定義する前に，**束縛** (binding) されている場合とそうでない場合があるので一意には定義できない．あえて定義するのなら，

$$⟦\ x\ ⟧ = \text{不定}$$

であろう．$⟦\ x\ ⟧ =$ エラー と定義してもよい．また,「エラー」を**ボトム** ($\bot$)(bottom) を使って定義することも多い．このように意味を通常の値だけでなく「不定」や「エラー」という値を導入することにより正確に定義することができる．

では，関数定義はどうなるのであろう．

$$⟦\ \texttt{fun twice x = x * 2}\ ⟧ = ⟦\ \texttt{val twice = fn x => x * 2}\ ⟧ = ?$$

しかし，その呼出しは，

$$⟦\ \texttt{fun twice x = x * 2 ; twice 3}\ ⟧ = 6$$

である.「関数という意味」を定義するためには新たな記法 (λ) を導入する．

[1] 正確には，値だけではない (後述).
[2] しかし, C 言語の場合は，参照への代入であって，直接変数への代入ではないので，正確ではない.
[3] 型は，独立して考えることができるのでここでは無視.

9.1 λ算法

「関数という値 (意味)」を定義するために λ 算法を導入する[4]．一般にプログラミング言語では関数定義は，

$$f(x) = x$$

のように定義し，f を**関数名** (function names)，(x) の x を**関数の引数** (function arguments)，$= x$ の x を関数の**返答** (reply) という．関数の機能「そのもの」はどのように定義するのだろう．そこで，

$$f = \lambda x.x$$

と書く．$\lambda x.x$ は，λx の x を引数として受渡され，$.x$ の x を返答して返す，関数「そのもの」を示す．容易に $g(y) = (y+1)$ なら，$g = \lambda y.(y+1)$ であることがわかる．$\lambda x.x$，$\lambda y.(y+1)$ で関数を表現するが，名前 (関数名) がついてないことに注意しよう．

変数を**置換** (substitute) する関数 $[b_1/a_1, \cdots, b_n/a_n]$ を，$[b_1/a_1, \cdots, b_n/a_n]M$ と書き，M 中の変数 $a_i (1 \leq i \leq n)$ を，$b_i (1 \leq i \leq n)$ に変換する関数と定義すると $\lambda z.[z/x]x = f$，$\lambda z.[z/y](y+1) = g$ である．つまり，λ の後の変数名は，特定されないことを意味する．$\lambda x.x$，$\lambda y.(y+1)$ のように，λ の後にある変数 x，y は，. (ピリオド) 以下の x，$y+1$ を束縛するという．また，この x，y を**束縛変数** (bind variables) という．逆に，$\lambda x.xz$，$\lambda y.(y+w)$ の場合の各 z，w を**自由変数** (free variables) という．通常，**演算子** (operators) +,−,∗,/ は，. より結びつきが強いので，$\lambda y.(y+1)$，$\lambda y.(y+w)$ などは，() を外して，$\lambda y.y+1$，$\lambda y.y+w$ と書く．また式 e の自由変数の集合を $FV(e)$ と書く．

以上の定義で，

$$[\![\texttt{fun twice x = x*2}]\!] = [\![\texttt{fun twice x = x*2 ; twice}]\!] = \lambda x.(x * 2)$$

であることがわかる．よって，

$$[\![\texttt{fun twice x = x*2 ; twice 3}]\!] = (\lambda x.(x * 2))\ 3 = 6$$

となる．このように，関数呼出しは $[\![f\ c]\!]$ のように表す．λ 算法では $f\ c$ と書き，f を c に**適用** (application) するという．

+，−，∗，/ などの演算子は，上の定義で自由に使っている．しかし，正確には，演算子も関数であり，λ を使って書かなければならない．複雑なので四則演算子などの基本的な演算子は自由に用いることにする．

課題 1

演算子 +，−，∗，/ を使わず，$\lambda x.x+1$，$\lambda y.y-w$ などを λ を使って表現せよ．また，数字 0〜9 なども λ を使って表現する方法がある，どんな方法があるか λ 算法の専門書を調べよ．

9.2 式

ML プログラム断片は，すべて同様に定義することができる．ML に限らず，大抵のプログラミング言語は，その最小構成要素 (**式** (expressions) または，**項** (terms) という) からなり，式を正確に定義することにより，プログラミング言語の**文法** (syntax) を定義することができる．ML(この ML を core-ML という) の式 $e \in Exp$ を定義してみよう (前章までは「フレーズ」として扱ってきたが，ここでは「式」として議論する．また簡単のために，リスト，レコードは除く)[5]．ここで議論する式の**領域** (domain)(集合論の集合，カテゴリ論を使って議論する場合は**カテゴリ**という) を Exp とする．同様に，論理定数領域 (boolean constant domain) $BConst$，その他の定数領域 (constant domain) $Const$，変数領域 (variable domain) Var，関数領域 (function domain) Fun，$UNIT$ 領域

[4] λ 算法の詳しい説明は他の専門書に任せ，ここではその導入のみを示す．

[5] この記法を BNF(backus normal form) という．または，ここで議論するときには，表現を簡素にするために曖昧さを含んだ BNF を使うことが多い．それを**抽象構文** (abstract syntax) という

(unit domain) $Unit$ と (ここで扱う) 意味領域 (semantic domain) Val とおく．容易に以下の関係が成り立つことがわかる．

$$Val \cong BConst + Const + Var + Pair + Fun + Unit + \{\texttt{wrong}\}_\perp$$

$$BConst = \{\texttt{true}, \texttt{false}\}_\perp$$

$$Pair = Val \times Val$$

$$Fun = Val \to Val$$

$$Unit = \{\texttt{unit}\}_\perp$$

$\cong$ の両辺は**同型** (isomorphic) であることを示し，この関係が成立する．

実行時エラーを示す wrong という値のみを持つ領域を $\{\texttt{wrong}\}_\perp$ とおく．実際この wrong という値は意味論として非常に重要である．SML では後述する型推論と合わせて意味論的に「型誤りがないプログラムは，決してこの wrong が現れない」つまり「実行する前の型チェック (型推論) で型誤りがなければ，実行時エラーは起こらない」ことが数学的に証明できる．この性質のことを**健全性** (soundness) という．

以上より式 $e \in Exp$ が**再帰的** (recursive) に定義できる．

$$e \ = \ b \mid c \mid x \mid (e, e) \mid e\ e \mid \lambda x.e \mid \texttt{let}\ x = e\ \texttt{in}\ e\ \texttt{end} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e\ \texttt{end}$$

b は論理定数を示し，c はその他の定数を示し，x は変数を示す．(e_1, e_2) は，式 e_1，e_2 の**組** (pair)，$e_1\ e_2$ は，式 e_1 へ式 e_2 を適用 (e_1 は，$\lambda x.e$ の形をしていなければならないことは容易にわかる)，$\lambda x.e$ は，関数を意味し，式 e を変数 x により λ **抽象** (λ abstraction) するという．以下 let 式と if 式である．

9.3 意味定義

上の例のような小さなプログラム断片は，局所的な値を求めることにより，その意味を定義することができるが，より大きなプログラムの意味を定義する場合には，問題が生じる．より大きなプログラムでは各変数が，どのような値 (意味) を持つかにより全体の意味が異なる．また，その値も実行の時点，時点により異なる．以下のプログラムを考えよう．

$$[\![\ \texttt{let x=10 in let y=23.1 in let z="test" in x end end end}\ ]\!] \ = \ 10$$

であるが，

$$[\![\ \texttt{let y=23.1 in let z="test" in x end end}\ ]\!] = [\![x]\!]$$

となり，自由変数 x の意味が定まらないとこの意味も定義できない．また，

$$[\![\ \texttt{let z="test" in x end}\ ]\!] = [\![x]\!]$$

はどうだろう．前者の $[\![\ x\ ]\!]$ と後者の $[\![\ x\ ]\!]$ では，変数 y の意味が加味されている場合とそうでない場合とで異なってきて当然である．

このように，意味定義の中に変数の意味も加味しないと正確な意味にならない．そこで，プログラムの時点，時点での各変数の意味 (変数は複数あることに注意) を定義した**環境** (environment) ρ を導入しよう．

$$\rho = \{\ x_1 : \nu_1, \cdots, x_n : \nu_n\ \}$$

のように，変数 x_i とその意味 $\nu_i, (1 \le i \le n)$ の組の集合とし $\rho(x_i) = \nu_i$ と書く．$\rho\{x : \nu\}$ は，$dom(\rho') = dom(\rho) \cup \{x\}, \rho'(x) = \nu, \rho'(z) = \rho(z), \text{if } z \neq x, z \in dom(\rho)$ である (直感的には，環境 ρ に $x : \nu$ を加えた環境を示す．ただし，変数 x は環境 ρ にない変数である)．

以下，この ρ を使って意味関数 $[\![\ ]\!]_\rho$ と書く．よって，

$$[\![\ \texttt{let x=10 in let y=23.1 in let z="test" in x end end end}\ ]\!]_\rho \ = \ [\![\ x\ ]\!]_{\{x:10, y:23.1, z:"test"\}} \ = \ 10$$

$$[\![\ \texttt{let y=23.1 in let z="test" in x end end}\ ]\!]_\rho\ =\ [\![\ x\ ]\!]_{\{y:23.1,z:"test"\}}$$

$$[\![\ \texttt{let z="test" in x end}\ ]\!]_\rho\ =\ [\![\ x\ ]\!]_{\{z:"test"\}}$$

と定義することができる．

結果として意味関数 $[\![\ ]\!]_\rho$ は以下のようになる．

$$[\![\ ]\!]_\rho\ :\ Exp \to Env \to Val$$

ここで $A \to B$ により領域 A から B への関数を示し，Env は環境を示す領域である．

$$Env = Var \to Val$$

さて，ML の処理系に戻ろう．例えば，

```
- val x = 1;↩
val x = 1 : int                    (* 意味定義は 1 *)
- fun twice x = x * 2;↩
val twice = fn : int -> int        (* 意味定義は fn *)
```

では 1 や fn など値として扱ってきたものが意味定義である．関数の値は fn を使って表すことができるが fn と表示されるのみである．

9.4 意味関数の定義

では，前節で紹介した ML の意味関数を定義しよう．

$$[\![\ ]\!]_\rho\ :\ Exp \to Env \to Val$$

で定義される意味関数は，式 e の各々について定義することができる．

- c の場合

$$[\![\ c\ ]\!]_\rho = c\quad \in Val\ \text{かつ}\ (\in Const\ \text{または}\ \in BConst\ \text{または}\ \in Unit)$$

  ここで，右辺は $c \in Val$ であることに注意しよう．

- x の場合

$$[\![\ x\ ]\!]_\rho = \rho(x)\quad \in Val$$

- $\lambda x.e$ の場合

$$\lambda v \in Val.[\![\ e\ ]\!]_{\rho\{x:v\}}\quad \in Val\ \text{かつ}\ \in Fun$$

- $e\ e$ の場合

$$[\![\ e_1\ e_2\ ]\!]_\rho = \underline{\text{もし}}\ e_1 \in Fun\ \underline{\text{の場合}}\ ([\![\ e_1\ ]\!]_\rho)([\![\ e_2\ ]\!]_\rho)\ \underline{\text{そうでない場合}}\ \texttt{wrong}\quad \in Val$$

- (e, e) の場合

$$[\![\ (e_1, e_2)\ ]\!]_\rho = ([\![\ e_1\ ]\!]_\rho, [\![\ e_2\ ]\!]_\rho)\quad \in Val\ \text{かつ}\ \in Pair$$

- let $x = e$ in e end の場合

$$[\![\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}\ ]\!]_\rho = [\![\ e_2\ ]\!]_{\rho\{x:[\![\ e_1\ ]\!]\}}\quad \in Val$$

- if b then e else e の場合

$$[\![\ \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ ]\!] = \underline{\text{もし}}\ b\ = \texttt{true}\ \underline{\text{の場合}}\ [\![\ e_1\ ]\!]_\rho\ \underline{\text{そうでない場合}}\ [\![\ e_2\ ]\!]_\rho\quad \in Val$$

ここで，$\lambda v \in Val.[\![\ e\ ]\!]_{\rho\{x:v\}}$ という表記は，λ 算法に似た記法であるが，全く別の領域論的表現である．領域 Val に属する値 v を引数に持つ関数を表す．

9.5 参照型

では意味関数を実行する (値を評価する) 関数を SML で定義する前に，新しい型 **参照型** (referential type) を紹介する．この型は，次節の実装の際に使う．参照型は，関数プログラミングの範囲を逸脱した概念である．今まで扱ってきた変数は，以下のように使った．

プログラム 81: 値を束縛する変数

```
val it = () : unit
- val x = 10;↩
val x = 10 : int
- val f = fn x => x;↩
val f = fn : 'a -> 'a
```

この例では整数型の値 10 に x という名前が付き，関数という値 (fn x => x) に f という値が付いた．つまり，値に対して名前がつくのである．一方，他の**手続き型言語** (procedural languages) または **命令型言語** (imperative language) では，値を格納する箱 (メモリの箱) ができ，その中に値が格納されている．

```
  +--------+
x |   10   |
  +--------+
```

このように,「変数という箱」に名前がついていると考えることができる．このような場合，名前を x という**参照** (reference) または**ポインタ** (pointers) と考えて，

```
      +---------+
x --> |    10   |
      +---------+
```

と同等であることがわかる．--> は参照を表す (C 言語と同様)．SML では，このような参照を表す変数は以下のように扱うことができる[6]

操作 64: 参照型

```
- val x = ref 10;↩
val x = ref 10 : int ref
- x;↩
val it = ref 10 : int ref
- val y = !x;↩
val y = 10 : int
- y;↩
val it = 10 : int
```

変数 x は，ref 10 のように整数型の値の前に ref をつけた値を持っている．結果として x は，整数型への参照型となる．参照型の変数はそのままでは，値を取り出すことはできない．その場合 ! という演算子 (dereference する演算子) を使ってのみ値を取り出すことができる．結果として変数 y は参照型ではなく整数型となる．参照型の変数は，上で説明したように 箱ができる のでその箱に対して**代入** (assignment) ができる．この代入は，以下のように使う．

[6] ここで注意したいのは，箱はどの参照からも「参照される」ことがなくなることが[7]ことである．その様な箱は**ごみ** (garbage) といい**ごみ集め機能** (garbage collector) により自動的に削除される．ごみ集め機能は SML の処理系に標準で備わっている．

操作 65: 代入

```
- x := 33;↩
val it = () : unit
- x;↩
val it = ref 33 : int ref
- !x;↩
val it = 33 : int
- x := "true";↩
std_in:15.1-15.11 Error: operator and operand don't agree (tycon
mismatch) operator domain: int ref * int operand: int ref * string
in expression: := (x,"true")
```

代入演算子 := を使って代入することができる．また，参照型の場合は型が宣言時に固定されている (箱の型) ので左辺の変数の中身の型 (この場合は x の中身の型 int) と異なる型を代入しようとするとエラーとなる[8]．参照型を示す ref は**構成子** (constructor) であるが SML では関数として実装されていることに注意しよう．

操作 66: 関数 ref

```
- ref;↩
val it = fn : 'a -> 'a ref
```

ここで，'1a という新しい多相型が推論されているがこれを**弱い型** (weak type) という．一方，通常の多相型を**強い型** (strong type) という．強い型，弱い型の概念は後述する型推論の知識が必要なのでここではこれ以上詳しく説明しない (後述)．しかし，通常の参照型を使うためには上の知識があれば十分である．

9.6 意味関数の実行

意味関数を実行するということは，構文木を解釈しその意味に変換することを示す．結果として ML のプログラム断片の値を求めることになる．最初に，意味 (値) を示すデータ型を定義しよう．上で定義した，領域 $BConst$ を Val_bool，領域 $Const$ を Val_number，領域 $Pair$ を Val_pair，領域 Fun を Val_closure または Val_primitive，領域 $Unit$ を Val_unit とする．

プログラム 82: データ型 value

```
datatype value = Val_number of int
|  Val_bool of bool
|  Val_pair of value * value
|  Val_closure of closure
|  Val_primitive of value -> value
|  Val_unit
withtype environment = (string * value) list
and closure = ((motif*expr) list)*environment;
```

型 environment は，文字列 (string) 型と値 (value) 型の組のリストから成り，意味関数の定義で扱った環境を表す (変数名と意味の組)．例えば変数 x が，値 10 を束縛している場合は，

```
("x", Val_number 10)
```

[8] let 式を使わなくてもよいことに注意しよう

となる. Val_number, Val_bool, Val_pair, Val_unit は, 各々 整数値, 論理値, 組, unit の意味を示す. Val_primitive は, 既定義の演算子の意味を表す関数を示す. Val_closure は, 通常の関数を表現するが, 環境を含んだ意味となっていることに注意しよう. 次に, 意味関数を実行する関数を Eval として定義する. この関数では, 新たに以下の例外処理を行なう.

プログラム 83: 例外処理

```
exception Not_found;
exception Eval_error of string;
exception Fail_filtrate;
```

```
fun filtrate value motif =
  case (value, motif) of
    (v, Motif_variable id) => [(id,v)]
  | (Val_bool b1, Motif_boolean b2) =>
      if b1 = b2 then [] else raise Fail_filtrate
  | (Val_number i1, Motif_number i2) =>
      if i1 = i2 then [] else raise Fail_filtrate
  | (Val_pair(v1,v2), Motif_pair(m1,m2)) => (filtrate v1 m1)@(filtrate v2 m2)
  | (_,_) => raise Fail_filtrate;
fun Value_application env list_of_case argument =
  case list_of_case of
    [] => raise Eval_error "Fail of filtrate"
  | ((motif, expr) ::  other_case) =>
      let val env_extend = (filtrate argument motif) @ env in
        Eval env_extend expr
      end handle Fail_filtrate => ( Value_application env other_case argument )
and Value_definition env_sequence def =
  let val (b,n,ex) = def in
    if b then raise Eval_error "let rec is not implemented."
    else (n,(Eval env_sequence ex))::env_sequence
  end
and Eval env expr =
  case expr of
    Number n => Val_number n
  | Variable id => ( (assoc id env) handle Not_found => (
      raise Eval_error (id^" is not found."); Val_unit ) )
  | Application( function, argument ) =>
      let val val_function = Eval env function;
        val val_argument = Eval env argument in
          case val_function of
            Val_primitive function_primitive => function_primitive val_argument
          | Val_closure (d,e) => Value_application e d val_argument
          | _ => raise Eval_error "application of a value is not functional."
      end
  | Pair (e1,e2) => Val_pair(Eval env e1, Eval env e2)
  | Boolean b => Val_bool b
```

```
  | Let (definition, bodys) =>
      let val Def def = definition in
        Eval (Value_definition env def) bodys
      end
  | Function(list_of_case) => Val_closure ( list_of_case, env )
  | If(e1,e2,e3) => (
      case (Eval env e1) of
        Val_bool true => Eval env e2
      | Val_bool false => Eval env e3
      | _ => raise Eval_error "if-expression is not valid." )
  | Unit => Val_unit;
```

この中で用いている関数 assoc は，2 章で定義した関数である．

関数 Eval は，意味関数を実行し環境を用いる．環境は，あらかじめ既定義の演算子や関数を定義しておかなければならない (**初期環境** (initial environment) という)．その初期環境を Val_env_initial としよう．

プログラム 84: 環境

```
fun code_number n = Val_number n;
fun code_number_fn x = case x of
    Val_number n => n
  | _ => raise Eval_error "integer is expected.\n";
fun code_bool b = Val_bool b;
fun code_bool_fn x = case x of
    Val_bool b => b
  | _ => raise Eval_error "boolean is expected.\n";
fun prim1 encoder calcul decoder =
    Val_primitive (fn v => encoder(calcul (decoder v)));
fun prim2 encoder calcul decoder1 decoder2 =
    Val_primitive( fn (Val_pair(v1,v2)) =>
      encoder(calcul((decoder1 v1),(decoder2 v2)))
  | _ => raise Eval_error "pair is expected." );
fun exiting v = ( raise End_of_system; v );
val Val_env_initial = [
  ("+", prim2 code_number ((op+):(int*int)->int)
    code_number_fn code_number_fn),
  ("-", prim2 code_number ((op-):(int*int)->int)
    code_number_fn code_number_fn),
  ("*", prim2 code_number ((op* ):(int*int)->int)
    code_number_fn code_number_fn),
  ("/", prim2 code_number ((op div):(int*int)->int)
    code_number_fn code_number_fn),
  ("=", prim2 code_bool (op=) code_number_fn code_number_fn),
  ("<>", prim2 code_bool (op<>) code_number_fn code_number_fn),
  ("<", prim2 code_bool (op<) code_number_fn code_number_fn),
  (">", prim2 code_bool (op>) code_number_fn code_number_fn),
  ("<=", prim2 code_bool (op<=) code_number_fn code_number_fn),
  (">=", prim2 code_bool (op>=) code_number_fn code_number_fn),
  ("not", prim1 code_bool not code_bool_fn),
  ("quit", Val_primitive(fn x => exiting x))];
```

最後に，関数 Eval は，前章までの出力である構文木を入力として構文木を辿りながら実行する．そのように関数 run を変更すればよい．

プログラム 85: 関数 run

```
fun run() =
  let val val_env = ref Val_env_initial; val stream_of_enter = std_in in
    while true do ( print("## "); flush_out std_out;
      let val result = phrase stream_of_enter in (
        print_definition result; print "\n";
        case result of
          Expr expr => let val ret = Eval (!val_env) expr in
              print "- : <type> = "; print_value ret; print "\n" end
        | Def def =>
            let val new_val_env = Value_definition (!val_env) def in
              case new_val_env of ((name, v)::_) => ( print (name^" : ");
                print "<type> = "; print_value v; print "\n" );
              val_env := new_val_env end )
      end handle Syntax_error => print "Parse error: Syntax error.\n"
        | Cant_close_parenthesis =>
            print "Parse error: can't find the right parenthesis.\n"
        | Eval_error str => ( print "Eval error: "; print str; print "\n" )
        | End_of_system => ( print "End of MINICAML.sml...\n";
            raise Quit_system; () ) )
  end;
```

この中で呼出されている関数 print_value は以下のようである.

プログラム 86: 関数 print_value

```
fun print_value x =
  case x of
    Val_number n => print n
  | Val_bool false => print "false"
  | Val_bool true => print "true"
  | Val_pair(v1,v2) => ( print "("; print_value v1; print ", ";
      print_value v2; print ") " )
  | Val_closure _ => print "<fun>"
  | Val_primitive _ => print "<fun>"
  | Val_unit => print "()"
```

この関数 run を実行すると以下のようになる.

操作 67: 関数 run の実行 (1)

```
- run();↩
## let x = 1;↩
Def ( false, x, Number 1 )
x : <type> = 1
## x;↩
Expr ( Variable x )
- : <type> = 1
## let y = x + 1 in y/2;↩
Expr ( Let ( Def ( false, y, Application ( Variable +, Pair ( Variable
x, Number 1 ) ) ), Application ( Variable /, Pair ( Variable y, Number
2 ) ) ) )
- : <type> = 1
## let twice = function x -> (x*2);↩
Def ( false, twice, Function ( ( Motif_variable "x", Application (
Variable *, Pair ( Variable x, Number 2 ) ) ) ) )
twice : <type> = <fun>
## twice 10;↩
Expr ( Application ( Variable twice, Number 10 ) )
- : <type> = 20
## quit();↩
Expr ( Application ( Variable quit, Unit ) )
End of MINICAML.sml...

uncaught exception Quit_system
```

ここで <type> とは後述する**型推論システム** (type inference system) で推論された型を表示する部分である．

最後に仕上げとして，構文木の表示をなくするためには関数 run の定義の 4 行目を注釈にすればよい．その結果以下のようになる．

操作 68: 関数 run の実行 (2)

```
- run();↩
## let x = 3 + 4 / 2;↩
x : <type> = 5
## x;↩
- : <type> = 5
## let y = (3,4);↩
y : <type> = (3, 4)
## x + y;↩
Eval error: integer is expected.

## y;↩
- : <type> = (3, 4)
## let rec fac = function 0->1|x->x*(fac (x-1));↩
Eval error: let rec is not implemented.
## quit();↩
End of MINICAML.sml...

uncaught exception Quit_system
```

10

9 章のドリル

10.1　文字列とリスト型を導入する

8 章で導入した 文字列 と リスト についても機能を果たすように関数 Eval を改良せよ．文字列は "test", "string" などと表現し，時間がある場合は，連結演算子 ^ も実現せよ．

操作 69: 文字列を使った実行の例

```
## "string"↩
- : <type> = "string"
## "string1"^"string2"↩
- : <type> = "string1string2"
```

リストは，コンス演算子 :: のみ追加し 1:2:3:[] などと表現する．コンス演算子は，

要素 :: リスト

と書くので最後の要素は必ず [] (**ヌル文字** (null string)) となる．

操作 70: リストを使った実行の例

```
## 1::2::3::[];↩
- : <type> = 1::2::3::[]
## hd (1::2::3::[]);↩
- : <type> = 1
## tl (1::2::3::[]);↩
- : <type> = 2::3::[]
## let f = function x -> (x::6::[]);↩
f : <type> = <fun>
## f 3;↩
- : <type> = 3::6::[]
## let g = function (x::y) -> x;↩
- : <type> = <fun>
## g (3::4::[]);↩
- : <type> = 3
```

このように，以下の条件を満たすように実現せよ．

1. 1::2::3::[] のように，通常のリストを入力し評価する．
2. 先頭要素を取り出す関数 hd と 先頭以外のリストを取り出す関数 tl を既定義の関数とせよ．
3. 関数の返値としてリストを指定することができる．
4. 関数のモチーフ (パターン) としてリストを指定することができる．

10.2 C 言語で実装

ここまでの処理 (字句解析，構文解析，意味関数の実装) を C 言語で実装せよ (文法としては，前節 の文字列とリスト型を導入したところまで).

11

計算モデル

関数プログラムの振舞を厳密に定義するために，計算モデルを導入しよう．広い意味での意味論には，**表示意味** (denotational semantics)， **公理意味** (axiomatic semantics) および**操作意味** (operational semantics) がある．表示意味は，式の意味を，抽象的に数学的な対象と解釈する意味論である．この際用いられる数学的対象が**意味領域** (semantic domain) である．公理意味は，式の間に成り立つ同値関係を定義することによって，式の意味を定義する方法である．操作意味は，式と式を評価した結果の関係を定めることによって，式の意味を定める方法である．この中で表示意味は，言語の文法構造に依存しない最も抽象的な意味論であり，通常意味論といった場合，この表示意味を指す．

さて，本章では 9 章で導入した**領域** (domain) をより詳しく議論してみよう．さらに表示意味，公理意味の議論へ発展する．領域は，プログラミング言語の断片を表現した，式，文などの集合である．その集合をモデル化するために，最初に**代数** (algebra) を導入する．その結果，その代数を使って，プログラミング言語を表現する領域を表現してみよう．

11.1 代数

代数とは，ある集合の集合と，その集合間の演算を表現した演算子の集合の組で表現することができる．例えば，$Boolean'$ 代数 (論理値を示す) は，集合として $\{true, false\}$ をもち，演算子 $\vee$(OR)，$\wedge$(AND)，$\neg$(NOT) をもつ．

$$Boolean' = (\{BOOL\}, \vee, \wedge, \neg)$$

また，Nat' 代数 (自然数を示す) は，集合として $\{0, 1, 2, \cdots\}$ [1] をもち，演算子 $zero$，$succ$ をもつと定義することができる．

$$Nat = (\{NAT\}, zero, succ)$$

代数の，各集合を **台** (carrier) といい，各台の名前を**ソート** (sort) [2] という．代数は，集合の集合を表現するため，ソートを複数定義することができることに注意しよう．本書で扱う代数を **多ソート代数** (many sorted algebra) という．

定義 1: **シグニチャと代数**

任意のソートの集合を S と，ソート間の演算の集合の組を**シグニチャ**(signature) という．シグニチャ Σ に対して Σ 代数 A は，以下の 3 つから成る．

1. 各ソート $s \in S$ に対する非空集合 A_s．A_s は，ソート s の台という．
2. 各演算子 $\alpha : s_1 \times s_2 \times \cdots \times s_n \to s$ に対する関数 $\alpha_A : A_{s_1} \times A_{s_2} \times \cdots \times A_{s_n}$.
3. 各定数 $\alpha : \epsilon \to s$ に対する A_s の元 α_A.

[1] コンピュータサイエンスでは，自然数をよく 0 から始まる値と定義する．
[2] 計算モデルでの領域と同意義である．

本書では，代数をわかりやすく表現するため以下のように表現する．$Boolean'$ 代数は，台として $\{true, false\}$ をもつが，ここでは，定数を演算子 $true$，$false$ として定義しこの演算子で生成されるものとする．

$$Boolean = (\{BOOL\}, true, false, \vee, \wedge, \neg)$$

プログラム 87: $Boolean$ 代数 (1)

```
ALGEBRA Boolean
  SORT BOOL
  OPERATORS
    true ∈ (· → BOOL), false ∈ (· → BOOL)
    _ ¬ _ ∈ (BOOL → BOOL), _ ∨ _ ∈ (BOOL BOOL → BOOL)
    _ ∧ _ ∈ (BOOL BOOL → BOOL), _ = _ ∈ (BOOL BOOL → BOOL)
```

構文的な定義として，一行に複数のソート，演算子を書く場合は，コンマ (,) で区切ることにし，アンダスコア (_) で各演算子の引数の位置を示すことにする．

Nat 代数も，台として $\{0, 1, 2, \cdots\}$ をもつが，ここでは，演算子 $zero$ と $succ$ により生成される値とする．

プログラム 88: Nat 代数

```
ALGEBRA Nat
  SORT NAT
  OPERATORS
    zero ∈ (· → NAT), succ _ ∈ (NAT → NAT)
```

上のように代数を定義しただけでは，各演算子の枠組を定義しただけで，我々の意図した台の集合と各演算子の組を表現しているとは限らない．例えば，Nat 代数では，

1. Nat 代数 A に対して $A_{NAT} = \{0, 1, 2, \cdots\}$，$A_{zero} = 0$，$A_{succ}(n) = n + 1$
2. Nat 代数 B に対して $B_{NAT} = \{0, 1\}$，$B_{zero} = 0$, $B_{succ}(n) = 1 - n$
3. Nat 代数 C に対して $C_{NAT} = \{0\}$，$C_{zero} = 0$，$C_{succ}(n) = 0$

のように定義しても上の代数の定義を満たす．この A，B，C を Nat 代数に対して **モデル** (model) という．ここでは，Nat 代数のモデル A，Nat 代数のモデル B のようにいう．

計算モデルに対しての 'モデル' もこのモデルを指す．つまり，ある代数 (例えば，$Func$ 代数と定義しよう) に対しての一つのモデルを定義することである．

11.1.1 代数の同一性

各代数 (モデル) 間の関係を定義する．各代数間では，**準同型写像**を定義することができる．

定義 2: 準同型写像

シグニチャ Σ に対して Σ 代数 A および B とする．次の条件を満たす写像の族 $h = \{h_s : A_s \to B_s\}_{s \in S}$ を A から B への準同型写像という．

- 任意の演算子 $\alpha : s_1 \times s_2 \times \cdots \times s_n \to s$ に対して $h_s \, o \, (\alpha_A) = \alpha_B \, o \, (h_{s_1} \times \cdots \times h_{s_n})$.
- 任意の定数 $\alpha : \epsilon \to s$ に対して $h_s(\alpha_A) = \alpha_B$.

ここで o は関数合成を表し，$h_{s_1} \times \cdots \times h_{s_n} : A_{s_1} \times \cdots \times A_{s_n} \to B_{s_1} \times \cdots \times B_{s_n}$ は，$h_{s_1} \times \cdots \times h_{s_n}(< \alpha_1, \cdots, \alpha_n >) =< h_{s_1}(\alpha_1), \cdots, h_{s_n}(\alpha_n) >$ で定める関数である．全ての $s \in S$ に対して h_s が上への写像 (単射) であるとき，h は 上への準同型写像という．また，同様に 1 対 1 の準同型写像も定義できる．つまり，単射の準同型写像と全射の準同型写像である．全単射の準同型写像は，**同型写像** (isomorphic) という．

例えば，前節で紹介した Nat 代数のモデル A，B，C に対して準同型写像を定義すると以下のようになる．

$$h = \{h_s : A_s \to B_s\}_{s \in \{NAT\}}$$

は $h_{NAT}(n) = n\%2$ (演算子 % は剰余を示す) と定義することにより，

$$h_{NAT}(zero_A) = zero_B$$

$A_{succ} : NAT \to NAT$ より，

$$h_{NAT}\, o\, A_{succ}(n) = (n+1)\%2 = 1 - (n\%2) = B_{succ}(h_{NAT}(n))$$

よって，A から B への準同型写像 h が存在した．同様に，

$$h = \{h_s : A_s \to C_s\}_{s \in \{NAT\}}$$

の準同型写像は $h_{NAT}(n) = 0$ より h が定義できる．

11.1.2 代数仕様

代数を定義しただけでは，複数のモデルが存在し我々の意図したモデルを定義したことにならない．そこで，より決定的なモデルを表現するために，各台の等号関係を宣言的に定義する (**公理** (axiom) という)．例えば，$Boolean$ 代数に公理を追加することができる．

プログラム 89: $Boolean$ 代数 (2)

```
ALGEBRA Boolean
  SORT BOOL
  OPERATORS
    true ∈ (· → BOOL), false ∈ (· → BOOL)
    _¬_ ∈ (BOOL → BOOL), _∨_ ∈ (BOOL BOOL → BOOL)
    _∧_ ∈ (BOOL BOOL → BOOL), _=_ ∈ (BOOL BOOL → BOOL)
  VARS
    b ∈ BOOL, b1 ∈ BOOL, b2 ∈ BOOL
  AXIOMS
    ¬ false = true, ¬ true = false
    false ∨ b = false, b ∨ false = false
    true ∧ true = true, (b1 ∨ b2) = ¬ (¬b1 ∧ ¬b2)
    (b1 = b2) = (¬b1 ∧ ¬b2) ∨ (¬b2 ∧ ¬b1)
```

このように，公理を追加することにより代数のモデルをしぼり込むことができる．公理の定義によっては，必ずしも一つのモデルを定義しているとは限らない．つまり，同じく代数を定義していることに過ぎない．そこで，このように公理を定義した (追加した)，より具体的な代数を**代数仕様** (algebraic specification) と区別することもある．

11.2 ソートの和，積

一つの代数で複数のソートを扱うことができる多ソート代数では，ソートの和，積も新たなソートとなる．ここでは，後で必要となるソートの積について定義する．

プログラム 90: ソートの積

ALGEBRA $Product$
  SORT $A \times B$
  OPERATIONS
    $< \_, \_ > \in (A\ B \rightarrow (A \times B))$
    $first\ \_ \in ((A \times B) \rightarrow A)$
    $second\ \_ \in ((A \times B) \rightarrow B)$
    $\_ = \_ \in ((A \times B)\ (A \times B) \rightarrow BOOL)$
  AXIOMS
    $(first < a, b >) = a,\ (second < a, b >) = b$
    $(< a_1, b_1 > = < a_2, b_2 >) = ((a_1 = a_2) \wedge (b_1 = b_2))$

11.3 *Integer* 代数

標準的な代数である *Integer* 代数を定義しよう．各整数の値は，その**符号** (sign) と自然数の組で表現することができる．

プログラム 91: *Integer* 代数

ALGEBRA $Integer$
  SORT $INT = BOOL \times NAT$
  OPERATIONS
    $\_ + \_ \in ((INT \times INT) \rightarrow INT)$
    $\_ * \_ \in ((INT \times INT) \rightarrow INT)$
    $- \_ \in (INT \rightarrow INT)$
    $\_ < \_ \in ((INT \times INT) \rightarrow INT)$
    $\_ = \_ \in ((INT \times INT) \rightarrow INT)$
  AXIOMS
    $- < s, n > = < \neg s, n >$
    $(< s_1, n_1 > * < s_2, n_2 >) = < s_1 = s_2, n_1 * n_2 >$
    $(< s_1, n_1 > = < s_2, n_2 >) = (s_1 = s_2 \wedge n_1 = n_2) \vee (n_1 = n_2 = zero)$

11.4 *Unit* 代数

ML の値の一つである `[]` (unit) を表現する *Unit* 代数を定義しよう．

プログラム 92: $Unit$ 代数

```
ALGEBRA Unit
  SORT Unit
  OPERATIONS
    [ ] ∈ (· → Unit)
    _ = _ ∈ (Unit × Unit → BOOL)
  AXIOM
    ([ ] = [ ]) = true
```

11.5 $Func$ 代数

では，関数プログラミング言語の計算モデルを表現する $Func$ 代数 を定義しよう．$Func$ 代数のシグニチャは，以下のようになる．

$$Func = (\{EXP\}, apply)$$

ここで扱うソートは EXP とし，$apply$ は関数適用を意味する．

プログラム 93: $Func$ 代数

```
ALGEBRA Func
  SORT EXP = BOOL + INT + UNIT + (EXP × EXP) + (EXP → EXP)
  OPERATION
    _apply_ ∈ (((EXP → EXP) EXP) → EXP)
  VARS
    f ∈ (EXP → EXP), g ∈ (EXP → EXP), x ∈ EXP
  AXIOMS
    (apply f x = apply g x) = (f = g)
```

11.6 ML から $Func$ 代数への意味関数

我々が，構築している ML と $Func$ 代数の意味が同一であることを示すため 9 章で導入した意味関数と同様の関数を定義してみよう．

$$[\![\cdot]\!]_\iota : Exp \to AEnv \to EXP$$

ここで，環境 ρ と同様に 自由変数の集合から EXP への関数である環境 ι を定義する．

$$\iota : Var \to EXP$$

また，Exp は ML の式の集合であり，EXP は $Func$ 代数のソート名である．

1. c の場合

$$[\![c]\!]_\iota = const(c)$$

ただし，$const(c)$ は，c に対する，あるソート S に属する演算子へ対応させる解釈関数を示す．例えば，

$$[\![\, true \,]\!]_\iota = const(true) = true$$

$$[\![\, 2 \,]\!]_\iota = const(2) =< true, succ(succ(zero)) >$$

となる．

2. x の場合

$$[\![x]\!]_\iota = \iota(x)$$

3. $\lambda x.e$ の場合

$$[\![\lambda x.e]\!]_\iota = \text{以下を満たす } f$$

$$\forall v \in EXP.apply\ f\ v\ =\ [\![e]\!]_{\iota\{x:v\}}\ \ (\text{ただし } x \in FV(e))$$

4. $e_1\ e_2$ の場合

$$[\![e_1\ e_2]\!]_\iota = apply\ [\![e_1]\!]_\iota\ [\![e_2]\!]_\iota$$

5. (e_1, e_2) の場合

$$[\![(e_1, e_2)]\!]_\iota =< [\![e_1]\!]_\iota, [\![e_2]\!]_\iota >$$

6. $fst(e)$ の場合

$$[\![\mathsf{let}\ fst = \lambda(e_1, e_2).e1\ \mathsf{in}\ fst(e)\ \mathsf{end}]\!]_\iota = first\ [\![e]\!]_\iota$$

7. $snd(e)$ の場合

$$[\![\mathsf{let}\ snd = \lambda(e_1, e_2).e2\ \mathsf{in}\ snd(e)]\!]_\iota = second\ [\![e\ \mathsf{end}]\!]_\iota$$

8. $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$ の場合

$$[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}]\!]_\iota = [\![\ e_2\ ]\!]_{\iota\{x:[\![e_1]\!]_\iota\}}$$

9. $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\ \mathsf{end}$ の場合

$$[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\ \mathsf{end}]\!]_\iota = \underline{\text{もし}}\ [\![e_1]\!]_\iota = true\ \underline{\text{ならば}}\ [\![e_2]\!]_\iota\ \underline{\text{そうでない場合}}\ [\![e_3]\!]_\iota$$

あるモデル A に対して，$e_1, e_2 \in Exp$ のとき，

$$[\![e_1]\!]_\iota = [\![e_2]\!]_\iota$$

ならば (意味が等しい),

$$A \models e_1 = e_2$$

と書く．与えられた等式集合 E の任意の等式 $e_1 = e_2 \in E$ に対して,

$$A \models e_1 = e_2$$

ならば,

$$A \models E$$

と書く．E を満たす任意のモデル B に対して,

$$B \models e_1 = e_2$$

ならば,

$$E \models e_1 = e_2$$

と書く．特に，等式集合 $E = \emptyset$ の場合は,

$$\models e_1 = e_2$$

と書く．

11.7 意味定義

$Func$ 代数に対して，意図したモデルを示す意味を数学的に正確に定義しよう．9 章で導入したように，意味を数学的に定義するためには意味関数を定義すればよい．ここでは，その意味を最初に集合論的に表現する．ここで議論する意味は，$Func$ 代数に対する **表示意味** (denotational semantics) である．

11.7.1 集合論的モデル

前節までの議論のように，$Func$ 代数には複数のモデルが存在する可能性がある．例えば，ソート EXP の台として，集合論で扱う通常の集合とするモデル A を考えよう．

- A_{EXP} は，構造的に定義することができる．
  1. A_{EXP} は，非空集合．
  2. $A_{EXP_1 \to EXP_2}$ は，A_{EXP_1} から A_{EXP_2} への関数の集合．
  3. $A_{EXP_1 \times EXP_2}$ は，$A_{EXP_1} \times A_{EXP_2}$．
- A_{apply} は，$apply\ f\ x\ =\ f(x)$ で与えられる関数の集合．

その他，実は，$Func$ 代数は，ソート $BOOL$，INT $A \times B$ で定義された演算子の意味も含んでいる．ソートの積に関する定義のみ示そう．

- $A_{<\,>}$ は，$<\ x_1, x_2\ >=\ (x_1, x_2)$ で与えられる関数の集合．
- A_{first} は，$first\ <x_1, x_2>=\ x_1$ で与えられる関数の集合．
- A_{second} は，$second\ <x_1, x_2>=\ x_2$ で与えられる関数の集合．

以上の定義は $Func$ 代数の定義を満たし，一つのモデルになることを確かめることができる．

11.7.2 領域論的モデル

集合論的モデルで表現される関数 $A_{EXP_1 \to EXP_2}$ は，$EXP_1 \to EXP_2$ 関数全体を示し，後の章で説明する再帰定義ができるとは限らない．再帰定義が可能である関数に限定した関数の集合のみを扱うために，より限定した領域を定義しよう．再帰定義が可能である関数とは，数学的には不動点をもつ関数と定義できる．

ソートの台として，**完備半順序集合** (CPO という) とするモデル C を定義する．

- C_{EXP} は，構造的に定義することができる．
  1. C_{EXP} は，$C_\perp$，ただし C は EXP の要素の集合．
  2. $C_{EXP_1 \to EXP_2}$ は C_{EXP_1} から C_{EXP_2} への連続関数の集合．
  3. $C_{EXP_1 \times EXP_2}$ は $C_{EXP_1} \times C_{EXP_2}$．
- C_{apply} は，$apply\ f\ x\ =\ f(x)$ で与えられる関数 $apply$ の集合．

同様に，

- $B_{<\,>}$ は，$<\ x_1, x_2\ >=\ (x_1, x_2)$ で与えられる関数の集合．
- B_{first} は，$first\ <x_1, x_2>=\ x_1$ で与えられる関数の集合．
- B_{second} は，$second\ <x_1, x_2>=\ x_2$ で与えられる関数の集合．

集合論的モデルと基本的な違いは，$C_{EXP_1 \to EXP_2}$ が C_{EXP_1} から C_{EXP_2} への連続関数の集合に限定している点である．そのためには C_{EXP} を CPO に限定する必要がある．CPO の議論は，次章で詳しくする．

12

11 章のドリル

前章で，**完備半順序集合** (以下 CPO という, complete partial order) を導入したが，その定義を最初にしよう．

12.1 完備半順序集合

集合 A 上の二項関係 $\preceq$ が以下の条件を満たすとき **半順序関係** (partial order) という．

1. (反射律) $\forall x \in A. x \preceq x$.
2. (推移律) $\forall x, y, z \in A. x \preceq y, y \preceq z \Rightarrow x \preceq z$.
3. (反対称律) $\forall x, y \in A. x \preceq y, y \preceq z \Rightarrow x = y$.

要素間で反順序が定義された集合を **半順序集合** (partial order set) という．$(A, \preceq)$ と書いて，集合 A で演算子 $\preceq$ で順序付けられた半順序集合を示す．$y \in A$，$X \subseteq A$ とする．任意の $x \in X$ に対して $x \preceq y$ のとき，y を X の**上界** (upper bound) といい $X \preceq y$ と書く．さらに，$X \preceq y$ かつ $X \preceq z$ なる任意の $z \in A$ に対して $y \preceq z$ が成り立つとき，y を X の**最小上界** (least upper bound) といい，$\sqcup X$ と書く．X が空集合でなく，かつ X の空でない任意の有限部分集合 Y が X の中に上界をもつとき，X を**有向集合** (directed set) という．

課題 1: 最小上界が存在すれば，唯一であることを示せ．

CPO とは，以下の条件を満たす半順序集合である．半順序集合 $(A, \preceq)$ とおき，

- 任意の $a \in A$ に対して $\bot_A \preceq a$ となるような**最小限** $\bot_A$ が存在する．
- A の任意の有向部分集合 X に対して，X の最小上界 $\sqcup X \in A$ が存在する．

領域論的モデルは，各集合 (ソート) を CPO に限定したモデルである．ここで，このモデル C が $Func$ 代数および，ML の表現するモデルであるかをみてみよう．

1. C_{INT} の場合
   整数の集合 $I = \{\cdots, -succ(succ(zero)), -succ(succ(zero)), zero, succ(zero), succ(succ(zero)), \cdots\}$ とする．
   I に含まれない要素 $\bot_I$ を加えた集合 $I \cup \{\bot_I\}$ の要素 x，y に対して，

   $$x \preceq y \Leftrightarrow x = y \text{ または } x = \bot_I$$

   で順序つけられる半順序集合 $(I \cup \{\bot\}, \preceq)$ を $I_\bot$ と書く．$I_\bot$ は CPO である．
2. C_{BOOL} の場合
   同様に，論理値の集合 $\{true, false\}$ に対して，$\{true, false\}_\bot$ は CPO である．
3. C_{UNIT} の場合
   [] の集合 $\{[]\}$ に対して，$\{[]\}$ は CPO である．

4. $C_{EXP_1 \to EXP_2}$ の場合 (C_{EXP_1} から C_{EXP_2} への連続関数)
   連続関数 (continuous function) とは，集合 A から B の関数 f に対して，集合 A の最小上界を保存する関数である．つまり，集合 A の任意の有向部分集合 X に対して，

$$f(\sqcup X) \ = \ sqcup\{f(x) : x \in X\} \ \in B$$

   の条件を満たす関数 f である．
   集合 A から B への連続関数を $\underline{A \to B}$ と書き，その集合を $\{\underline{A \to B}\}$ と書く．任意の 2 つの連続関数 $f, g \in \{\underline{A \to B}\}$ に対して，以下の関係を定義する．

$$f \preceq g \Leftrightarrow \ \forall x \in A. f(x) \preceq g(x)$$

   この関係は，半順序関係である．さらに $(\{\underline{A \to B}\}, \preceq)$ は CPO である．
5. $C_{EXP_1 \times EXP_2}$ の場合
   与えられた CPO A，B の積集合 $A \times B$ を以下のように定義する．

$$< a_1, a_2 > \preceq < b_1, b_2 > \quad \text{ここで } a_1 \preceq b_1,\ a_2 \preceq b_2$$

   この関係は，半順序関係があり，さらに $(A \times B, \preceq)$ は CPO である．

13

再帰定義の導入

前章の意味関数を実行するプログラムでは,

プログラム 94: 再帰定義

```
- run();↩
## let rec fac = function 0->1|x->x*(fac (x-1));↩
Eval error: let rec is not implemented.
```

であった．つまり，関数の**再帰定義** (recursive definition) が許されていない．その原因は意味 (値) を保持する datatype の宣言にある．

プログラム 95: データ型 value(1)

```
datatype value = Val_number of int
|  Val_bool of bool
|  Val_pair of value * value
|  Val_closure of closure
|  Val_primitive of value -> value
|  Val_unit
withtype environment = (string * value) list
and closure = ((motif*expr) list)*environment;
```

型 closure は，組として宣言されているがこの形では宣言された後に中の値 (この場合は，型 environment の値) を変更することができない．そのため，再帰的に環境を定義することをできなくしている．そこで，以下のように**レコード型** (record type) と参照型を使い宣言された後に変更できるようにしよう.

プログラム 96: データ型 value(2)

```
datatype value = Val_number of int
|  Val_bool of bool
|  Val_pair of value * value
|  Val_closure of closure
|  Val_primitive of value -> value
|  Val_unit
withtype environment = (string * value) list
and closure = { Definition: (motif*expr) list, Environment: (environment ref) };
```

ここで，Environment という**ラベル** (label) は，environment ref という参照型になっている．通常の値を示す ref とは異なることに注意しよう.

13.1 レコード型

では，ここで始めて使われたレコード型について説明する．レコード型は，組と同様の概念であるが以下で異なる．

- 各要素は名前 (**ラベル** (label) という) をもつ．
- 各要素は順不同である．

この 2 点が異なることにより全く異なった性質のデータ型を示す (但し，レコード型で組を表現することは可能である．実際 SML では，レコード型により組が実現されている)．レコード型は，各ラベル付き要素の積集合として表現されるが，和集合として表現されるものにバリアント型があった (前述)．

```
datatype ラベル$_1$ of 型$_1$ | ... | ラベル$_n$ of 型$_n$
```

この様に，レコード型とバリアント型は各ラベル付き要素の積，和集合の違いであることに注意しよう．

SML では，レコード型の型宣言はしなくてもよい (他の処理系ではしなければならないものもある)．

各要素の値を参照するためには，**ラベル抽出関数** (label extract function) を使う．ラベル抽出関数は，必ず # ではじめなければならない．

さくいん

【T】

【U】

【V】

【ア】

【カ】

【サ】

【タ】

【ナ】

【ハ】

【マ】

【ヤ】

【ラ】

【ワ】