目次

1． 活動の概要

この班では，関数型プログラミング言語 **Standard ML** を用いて，コンパイラを開発することを目標として活動した．活動期間は通年であるが，前期の活動としては，まず Standard ML の構文や言語機能についての勉強会を開くという形式をとった．前期活動の終盤では，コンパイラのフロントエンドの構成についての勉強会を行い，字句解析器や構文解析器について学んだ．

以下，勉強会を通じて得られた知見をまとめる．

2． 関数型プログラミング言語

関数型プログラミング言語とは，関数によってプログラムを構成するプログラミングスタイルを基本とする言語である．Standard ML の他に，LISP や Haskell，OCaml などが代表的な関数型プログラミング言語として知られている．

3． Standard ML

3.1 Standard ML の特徴

Standard ML は，プログラミング言語 ML の標準仕様，あるいは一方言として位置付けられる関数型プログラミング言語である．以下のような特徴を持っている．

- 強力な型システム
- パターンパッチング機構
- 高度なモジュールシステム
- 命令型言語的な機能

関数がプログラムの主役となる関数型言語では，関数は第一級オブジェクト(first-class object)である．ここで，第一級オブジェクトとは，変数に格納したり，関数の引数や戻り値に使用できるデータのことをいう．

3.2 開発環境

Standard ML の処理系としては，コンパイラ **mlton** と，対話的実行環境 **Standard ML of New Jersey**（以下，**smlnj**）が存在する．そのため，ソースファイルを用いてプログラムを作成することも，ソースファイルを作成せずに式の評価や型システムの動作を対話的にテストすることもできる．

mlton と smlnj は、どちらも apt，pacman，brew といったパッケージ管理システム経由で導入が可能である．このプロジェクト活動においては，macOS 環境では brew 経由で，Ubuntu 環境では apt 経由で，Arch linux 環境では pacman 経由で導入した．

3.3 式の評価と束縛，関数，コメント，ファイル

式の評価と束縛

smlnj は sml コマンドで起動する．式を入力して改行すれば評価される．複数行に渡る構文で記述している場合，改行して入力を続けることができる．

```
- 2 * 3 + 4 * 5;
val it = 26 : int
- it - 20;
val it = 6 : int
```

セミコロンは式の終端を表す．smlnj は式を1度評価するごとに，その結果を名前 it に束縛する．it を用いると，前回の式の評価結果を用いた新たな式の評価が可能である．

関数型言語では，代入は行わずに，異なる概念である**束縛**を行う．束縛は，式と名前を対応付ける操作を指す．例えば先ほど示したサンプルでは，最初は名前 it を 26 で束縛し，次は 6 で束縛している．

関数

以下で，2 つの int 型データを渡して和を得る add 関数を，2 種類の方法で定義する．

```
- fun add (x: int) (y: int): int = x + y;
val add = fn : int -> int -> int
- add 1 2;
val it = 3 : int
```

```
- fun add x y = x + y;
val add = fn : int -> int -> int
- add 1 2;
val it = 3 : int
```

どちらも挙動は全く同じだが，前者では全ての引数と戻り値の型を明記し，後者では可能な限り型を省略している．ここでは，Standard ML の型推論を最大限利用している．型推論とは，コンパイル時に不足している型の情報を推論する機能である．

コメント

```
(* コメントは (* ネスト可能 *) *)
```

ファイル

```
(* file: test1.sml *)
fun add x y = x + y
val m = 2
val n = 3
```

上の内容のソースコードを，`test1.sml` という名前で保存する．

```
- use "test1.sml";
[opening test1.sml]
val add = fn : int -> int -> int
val m = 2 : int
val n = 3 : int
val it = () : unit

- add 4 5;
val it = 9 : int
```

smlnj で `use "...";` を評価させると，文字列型データで指定したソースファイルをコンパイルして対話的実行環境にインポートする．

3.4 主なデータ型

基本データ型

型名	説明	リテラルの例
bool	論理型	true, false
int	整数型(符号付き整数型)	123, 0x1a, ~123 (負数)
word	ワード型 (符号なし整数型)	0w123, 0wx1a
real	実数型	3.14, ~3.14, 1.23e4, 1.23e~4
char	文字型	#"a", #"\n"
string	文字列型	"abc\n"

型名	説明	リテラルの例
unit	ユニット型（0-タプル）	()

複合型

型（例）	説明	リテラルの例
int * int * string	タプルの例	(123, 456, "abc")
{ a: int, b: int, c: string }	レコード型	{ a = 123, b = 456, c = "abc" }
int list	リスト型（値の列）	[3, 1, 4, 2, 5]

論理型

論理値の値は true または false である．論理型データの演算に用いる演算子として，not（否定），andalso（論理積），orelse（論理和）が挙げられる．

```
- not true;
val it = false : bool
- it orelse true;
val it = true : bool
```

数値型

数値型の演算には ~（単項マイナス），+（加算），-（減算），*（乗算），abs（絶対値）などを用いることができる．除算演算子には /（real 型用）と div（int，word 型用）の 2 種類がある．剰余演算子は mod で，int 型と word 型で利用できる．

```
- 2 * 3;
val it = 6 : int
- ~ it;
val it = ~6 : int
- abs it;
val it = 6 : int
- 10.0 / 3.0;
val it = 3.33333333333 : real
- 10 div 3;
val it = 3 : int
- 10 mod 3;
val it = 1 : int
```

これらの算術演算子は，2 つの引数が同一の型でなければ利用できない．例えば 2 + 3.0 は型エラーとなる．real(2) + 3.0 のように，適切な型に揃える必要がある．

実数型から整数型への変換には ceil（∞方向丸め），floor（-∞方向丸め），trunc（0方向丸め），round（四捨五入）といった関数が利用できる．

文字型/文字列型

\n のようなエスケープシーケンスが利用できる．また，文字列の連結には ^ 演算子を用いる．

```
- "abc" ^ "def\n"
val it = "abcdef\n" : string
```

タプル型/ユニット型

タプル（tuple）は，複数の値を組にしたものである．例えば，整数 123 と文字列 "abc" のタプルは (123, "abc") のように書き，その型は int * string と書く．

タプルの n（≥ 1）番目の要素を取り出すには #n という関数を利用する．

```
- (123, (456, "abc"));
val it = (123, (456, "abc")) : int * (int * string)
- #1 (#2 it);
val it = 456 : int
```

ユニット（unit）は，要素数が 0 のタプルで表現される．ユニット型の唯一の値は () であり，型名は unit である．ユニット型は，例えば，意味のある値を返さない関数の戻り値の型として用いられる．文字列を出力する print 関数の戻り値の型はユニット型である．

レコード型

レコード（record）はラベル付きの値の集合である．

```
- val taro = { name = "Taro", age = 25 };
val taro = {age=25,name="Taro"} : {age:int, name:string}
- #age taro
val it = 25 : int
```

上の例では，ラベルとして name と age を持つレコードを生成している．ここで作成したレコード taro の型は { age: int, name: string } であり，各要素を取り出すには #name や #age のような名前の関数（セレクタ関数）を用いる．

リスト型

リスト（list）は要素を一列に連結してデータ構造である．リテラルのシンタックスは [1, 2, 3] のようになっており，この場合の型名は int list となる．要素数 0 のリスト（空リスト）は [] または nil と書く．

```
- [1, 2, 3];
val it = [1,2,3] : int list
- nil;
val it = [] : 'a list
- [];
val it = [] : 'a list
```

多相型

以下のプログラムでは，恒等関数（引数の値をそのまま返す関数）を定義している．

```
- fun id x = x;
val id = fn : 'a -> 'a
```

`id` 関数の型は `'a -> 'a` と推論されていることがわかる．`'a` のようにアポストロフィで始まるパラメータは**型変数**（type variable）と呼ばれ，任意の型を表す．`'a -> 'a` は，`id` 関数が，例えば `int -> int` 型としても振る舞い，`char -> char` 型としても振舞うことを示している．

等値型

1 個のアポストロフィで始まる型変数は任意の型を表すが，2 個以上のアポストロフィで始まる型変数（`''a` など）は**等値型**（equality type）を表す．等値型とは，関数 `=` で等値比較可能な型である．`bool`，`int`，`char`，`string` などの離散的なデータ型は等値型であり，等値型から構成されるタプルやリストなどの複合型も等値型である．

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

すべての等値型には，比較演算子 `=` および `<>` を用いることができる．

3.5 関数定義，パターンマッチング，if，相互再帰/高階関数，関数式，部分適用，中置演算子

関数定義

以下のプログラムは，自然数の冪乗（`x` の `y` 乗）を返す関数 `power` を定義している．

```
fun power x 0 = 1
  | power x y = x * power x (y - 1)
```

関数の宣言は `fun` キーワードから始まり，`=` の左に関数名と引数，右に式を記述する．複数のケースを書きたい場合，上記の例のように縦棒 `|` を使って関数の記述を継続することができる．

上記のソースコードを `power.sml` というファイル名で保存し，`smlnj` から読み込むと，次のように動作する．

```
- use "power.sml";
[opening power.sml]
val power = fn : int -> int -> int
val it = () : unit
- power 2 8;
val it = 256 : int
- power 2 0;
val it = 1 : int
```

この `power` 関数の宣言の `1` 行目と `2` 行目を入れ替えてしまうと，`power x y` がすべてのケースにマッチしてしまうため，関数の呼び出しは終了しない．

```
fun power x y = x * power x (y - 1)
  | power x 0 = 1    (* こちらの定義には到達しない *)
```

パターンマッチング

これまでの説明で既に使用しているが，パターンには，変数や整数リテラルの他に，タプルやリスト，レコードも書くことができる．

```
- fun length nil = 0
    | length (x::xs) = 1 + length xs;
val length = fn : 'a list -> int
- length [1, 2, 3, 4, 5];
val it = 5 : int
```

次の例のように，興味のない部分にはワイルドカードと呼ばれるパターン `_` を使うこともできる．

```
- fun second (_, y) = y;
val second = fn : 'a * 'b -> 'a
- second (123, "abc");
val it = "abc" : string
```

パターンは，`val` 宣言の左辺に使うこともできる．

```
- val (a, b) = (123, "abc");
val a = 123 : int
val b = "abc" : string
```

`case` 式を用いて，式の中でパターンマッチを行うこともできる．

```
- fun length xs =
    case xs of nil => 0
             | x::xs => 1 + length xs;
val length = fn : 'a list -> int
```

`if` 式

パターンではなく論理式によって場合分けする場合には，`if` 式を用いる．あくまで式なので値を持つ．そのため，「`then` 式」「`else` 式」のどちらも省略できない．

```
- fun abs x = if x < 0 then ~x else x;
val abs = fn : int -> int
- abs ~5;
val it = 5 : int
```

相互再帰関数

次のプログラムでは，引数が偶数のときに真を返す関数 `even` と，引数が奇数のときに真を返す `odd` を定義している．

```
- fun even 0 = true
    | even n = odd (n - 1)
  and odd 0 = false
    | odd n = even (n - 1);
val even = fn : int -> bool
val odd = fn : int -> bool
- even 2;
val it = true : bool
```

相互再帰的な関数は，互いが互いを参照し合えるように，1 つの `fun` 宣言の中に記述する必要がある．そのために，この例のように `and` キーワードを用いる．

高階関数

関数を引数や戻り値に持つ関数は**高階関数**と呼ばれる．以下に示す `map` 関数が，標準で定義されている高階関数の代表的な例である．

```
- fun square x = x * x;
val square = fn : int -> int
- map square [1, 2, 3, 4, 5];
val it = [1, 4, 9, 16, 25] : int list
```

map 関数の定義は，例えば以下のように書くことができる．

```
fun map f nil = nil
  | map f (x::xs) = f x :: map f xs
```

関数式

関数を表す式は**関数式**と呼ばれ，しばしばラムダ抽象（lambda abstraction），匿名関数（anonymous function）とも呼ばれる．

以下の例は，関数式を用いて square 関数を定義したものである．

```
- val square = fn x => x * x;
val square = fn : int -> int
- square 5;
val it = 25 : int
```

関数式を用いると，map square [1, 2, 3, 4, 5] と等価な式を次のように書くこともできる．

```
- map (fn x => x * x) [1, 2, 3, 4, 5];
val it = [1, 4, 9, 16, 25] : int list
```

階乗を求める関数 fact は，関数式を用いると容易に実装できる．再帰関数は関数式の中で左辺の名前を参照するため，次のように rec キーワードが必要になる．

```
- val rec fact = fn 0 => 1 | n => n * fact (n - 1);
val fact = fn : int -> int
- fact 5
val it = 120 : int
```

部分適用

```
- fun add x y = x + y;
val add = fn : int -> int -> int
- val addTwo = add 2;
val addTwo = fn : int -> int
- addTwo 3;
val it = 5 : int
```

関数適用は左結合であるため，例えば式 add 2 3 は (add 2) 3 のように評価される．ここで add 2 は add 関数の**部分適用**（partial application）と呼ばれ，int -> int 型を持つ．addTwo 関数は，add 関数の部分適用を利用して定義されている．

関数の部分適用を考慮すると，add 関数の型は int -> (int -> int)，すなわち，引数が int 型で，戻り値が int -> int 型である関数の型と見ることができる．

今回の add 関数のように部分適用の可能な関数は，カリー化された関数（curried function）という．逆に，次に示す add' 関数のような関数は，非カリー化された関数という．

```
- fun add' (x, y) = x + y;
val add' = fn : int * int -> int
- add' (2, 3);
val it = 5 : int
```

中置演算子

ここでは pl という中置演算子を宣言する．演算子名は，関数名として使えるものなら何でも構わない．次のように書くと，pl が優先順位 6 の（左結合の）中置演算子として宣言される．

```
- infix 6 pl;
infix 6 pl
```

右結合の中置演算子を宣言するには，infix の代わりに infixr キーワードを用いる．中置演算子の定義は，普通の関数と同様に fun 宣言により記述する．例えば，中置演算子 pl の定義を次のように与えることが可能である．

```
- fun x pl y = x + y;
val pl = fn : int * int -> int
- 2 pl 3;
val it = 5 : int
```

中置演算子の直前に op キーワードを置くと，中置演算子を一時的に通常の関数として扱うことができる．

```
- op pl (2, 3)
val it = 5 : int
```

中置演算子の結合の優先順位は 0 から 9 までの数字をとる（9が最も強い）．トップレベルでは，以下のように演算子が宣言されている．

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

3.6 リスト操作関数，リストの畳み込み

ML のリスト

ML では，複数のデータを扱う際には，配列よりもリストがよく使われる．リストを記述する方法は 2 ある．以下に示す 2 つの式は同一のリストを表している．

```
1 :: 2 :: 3 :: nil
[1, 2, 3]
```

:: はしばしば **cons** と呼ばれ，第 1 引数に先頭要素，第 2 引数に後続のリストをとる．:: は右結合であるから，1 :: 2 :: 3 :: nil の結合は 1 :: (2 :: (3 :: nil)) のようになる．

ML で扱うリストは，同一の型の要素のみで構成されるリストである．つまり，[1, 1.0, #"a"] のようなリストは許されない．型が 'a である要素から構成されるリストの型を 'a list と書く．

リストを扱う関数

Standard ML には，初めからリストを扱う関数が用意されている．

```
- [1, 2] @ [3, 4, 5];  (* リストの連結 *)
val it = [1, 2, 3, 4, 5] : int list
- List.filter (fn x => x mod 2 = 1) [1, 2, 3, 4, 5];  (* filter p xs : 述語
p が真を返す要素だけを残す *)
val it = [1, 3, 5] : int list
```

リストの畳み込み

foldr は，二項演算子を使って右からリストを畳み込む関数である．二項演算子を + とすると，式 foldr + 0 [1, 2, 3] は 1+(2+(3+0)) と展開される．

```
- foldr (op +) 0 [1, 2, 3];  (* 3 + (2 + (1 + 0)) = 6 *)
val it = 6 : int
- foldr (op -) 0 [1, 2, 3];  (* 3 - (2 - (1 - 0)) = 2 *)
val it = 2 : int
```

畳み込み関数は，リスト操作関数の実装によく用いられる．例えば，map 関数は foldr を使って以下のように書くこともできる．

```
- fun map' f = foldr (fn (x, s) => f x :: s) nil;
val map' = fn : ('a -> 'b) -> 'a list -> 'b list
- map' (fn x => x * x) [1, 2, 3, 4, 5];
val it = [1, 4, 9, 16, 25] : int list
```

3.7 データ型の定義

datatype 宣言

次のプログラムは，長方形の大きさを表す rectangle 型と，長方形の面積を求める area 関数を定義したものである．

```
- datatype rectangle = RECT of real * real;
datatype rectangle = RECT of real * real
- fun area (RECT (w, h)) = w * h;
val area = fn : rectangle -> real
- val r = RECT (3.0, 4.0);
val r = RECT (3.0, 4.0) : rectangle
- area r;
val it = 12.0 : real
```

datatype キーワードで始まる宣言により，新しいユーザー定義型を定義できる．datatype 宣言で定義されるデータ型は，**代数的データ型** (algebraic data type) と呼ばれる．

この例で定義されている rectangle は**型構成子** (type constructor)，RECT は**値構成子** (value constructor) と呼ばれる．型構成子と値構成子には，同じ名前を用いることもできる．

一つの型に複数の値構成子を定義することもできる．次のプログラムは，円または長方形を表す shape 型を定義したものである．

```
- datatype shape = CIRC of real
                 | RECT of real * real
datatype shape = CIRC of real | RECT of real * real
- fun area (CIRC r) = 3.14 * r * r
    | area (RECT (w,h)) = w * h;
val area = fn : shape -> real
- val c = CIRC 2.0;
val c = CIRC 2.0 : shape
- area c;
val it = 12.56 : real
```

次のプログラムでは，曜日を表すデータ型 dayOfWeek を宣言したものである．このように，引数のない値構成子のみからなるデータ型は，特に**列挙型** (**enum**erated type) と呼ばれる．

```
- datatype dayOfWeek = SUM | MON | TUE | THU | FRI | SAT;
datatype dayOfWeek = FRI | MON | SAT | SUN | THU | TUE | WED
- fun isHoliday SAT = true
    | isHoliday SUN = true
    | isHoliday _ = false;
val isHoliday = fn : dayOfWeek -> bool
- isHoliday MON;
val it = false : bool
```

次のプログラムは，リスト型を模した型を宣言したものである．このように，型引数を導入して多相型を構成することもできる．

```
- datatype 'a list' = NIL | CONS of 'a * 'a list';
datatype 'a list' = CONS of 'a * 'a list' | NIL

- fun length' NIL = 0
    | length' (CONS (x, xs)) = 1 + length' xs;
val length' = fn : 'a list' -> int

- length' (CONS(1, CONS(2, CONS(3, NIL))));  (* length [1,2,3] 相当 *)
val it = 3 : int
```

型シノニム

存在するデータ型に別名を与えるには，以下のように type 宣言を用いる．

```
- type 'a collection = 'a list;
type 'a collection = 'a list
```

抽象データ型

値構成子が隠蔽された型を**抽象データ型**（abstract data type）という．これを用いて，ユーザーに対してデータへのアクセス手段のみを提供することができる．

次のプログラムは，スタックを表すデータ型 'a stack と，スタックを操作する関数群を宣言したものである．このプログラムでは，'a stack を抽象データ型として宣言している．

```
(* stack.sml *)
abstype 'a stack = NIL | INS of 'a * 'a stack
with
  (* 新規スタック *)
  val new = NIL

  (* 要素を先頭に追加する *)
  fun push (x, st) = INS (x, st)

  (* 先頭要素を取り出す *)
  fun pop (INS (x, st)) = (x, st)
end
```

抽象データ型は，abstype キーワードで始まる宣言により記述する．with から end までの間には，抽象データ型のデータへのアクセス手段となる関数の宣言を記述する．これを smlnj から読み込み，動作を確認する．

```
- use "stack.sml";
[opening stack.sml]
type 'a stack
val new = - : 'a stack
val push = fn : 'a * 'a stack -> 'a stack
val pop = fn : 'a stack -> 'a * 'a stack
val it = () : unit
- val st = new;
val st = - : 'a stack
- val st = push ("a", st);
val st = - : string stack
- val (x, st) = pop st;
val x = "b" : string
val st = - : string stack
```

– で実際の値構成子を隠蔽していることがわかる．

3.8 命令型言語の機能

参照型

参照型（reference type）は，値への参照を表すデータ型で，命令型言語のポインタに相当する．例えば，整数の `5` という値を参照する参照型の値は `ref 5` のように書き，その型は `int ref` のように書く．

```
- val m = ref 5;
val m = ref 5 : int ref
- val n = ref 5;
val n = ref 5 : int ref
- m = n;
val it = false : bool
```

この例で式 `m = n` の値が `false` となっているのは，名前 `m`，`n` が異なる参照型の値に束縛されているからである．参照型の値に対して使用できる関数には，次のようなものがある．

関数	型	説明
`! r`	`'a ref -> 'a`	`r` の参照する値を返す
`r := x`	`'a ref * 'a -> unit`	`r` の参照する値を `x` に書き換える

これらの関数を利用すると，命令型言語の機能である，変数への値の再代入と同様のことを実現できる．

```
- val m = ref 2;
val m = ref 2 : int ref
- !m;
val it = 2 : int
- m := 3;
```

```
val it = () : unit
- !m;
val it = 3 : int
```

評価順序

参照型の演算などといった副作用を伴うプログラムでは，式の評価順序が意味を持つ．Standard ML では，複数の式が並列に並んでいるとき，これらの式は原則として左から順に評価される．以下に示すレコード式はその一例である．

```
- { 1 = (print "1"), 2 = (print "2"), 3 = (print "3")};
123
val it = ((),(),()) : unit * unit * unit
```

複数の式の評価順序を制御するには，以下のような式を利用することもできる．

式	説明
（式1 ; … ; 式n）	式1，… ，式n を順に評価し，式n の値を全体の値とする
式1 before 式2	式1，式2（式2 は unit型）を順に評価し，式1 を全体の値とする

例外処理

トップレベルで宣言されている標準の例外には，次のようなものがある．

例外	説明
Bind	束縛の失敗
Chr	chr 関数の失敗
Div	ゼロ除算
Domain	定義域のエラー
Empty	空リストによるエラー
Fail of string	汎用の例外
Match	マッチングの失敗
Option	valOf 関数の失敗
Overflow	オーバーフロー
Size	サイズ超過
Span	span 関数の失敗
Subscript	範囲外の添字

3.9 モジュールシステム

ML には，大規模なプログラムを適当な構成単位で分割して管理するためのモジュールシステムが用意されている．モジュールシステムの構成要素は以下の 1 つである．

- ストラクチャ（Structure）：いわゆるモジュール
- シグネチャ（Signature）：ストラクチャの仕様を記述したもの
- ファンクタ（Functor）：ストラクチャを生成する関数

モジュール

Standard ML の標準モジュールには，リスト関連の関数等をまとめた List ストラクチャ，文字列操作関数等をまとめた String ストラクチャなどが存在する．これらのストラクチャに含まれる関数等を利用するには，List.filter や String.tokens のようにストラクチャ名による名前の修飾が必要となる．

```
- List.filter (fn n => n mod 2 = 0) [1,2,3,4,5,6,7,8];
val it = [2,4,6,8] : int list

- String.tokens (fn c => c = #"/") "/usr/local/bin";
val it = ["usr","local","bin"] : string list
```

ただし，次のように open 宣言を用いると，それ以降のストラクチャ名の修飾を省略することができる．

```
- open List;
- filter (fn n => n mod 2 = 0) [1, 2, 3, 4, 5, 6, 7, 8];
val it = [2, 4, 6, 8] : int list
```

ストラクチャ

次のプログラムは，連想リスト（alist）を実現するストラクチャ Alist を記述したものである．このストラクチャでは，1 つの例外，3 つのデータ型，4 つの関数を定義している．

```
(* file: alist1.sml *)

(* ストラクチャ Alist の定義 *)
structure Alist =
struct
  exception AlistExn

  type tkey = int                  (* キーの型 *)
  type tval = string               (* 値の型 *)
  type alist = (tkey * tval) list  (* 連想リストの型 *)

  (* 新規リストを作成 *)
  fun new () = nil : alist

  (* 要素を追加（キーが重複していれば例外発生）*)
  fun add (k,v) ls = if exists k ls then raise AlistExn else (k,v)::ls
```

```
  (* キー key が存在すれば true *)
  and exists key [] = false
    | exists key ((k,v)::ls) = k = key orelse exists key ls

  (* キー key に対応する値を返す *)
  fun find key [] = raise AlistExn
    | find key ((k,v)::ls) = if k = key then v else find key ls
end
```

これを対話環境に読み込んで，動作を確認する

```
- use "alist1.sml";
- open Alist;
- val ls = new ();
val ls = [] : alist
- val ls = add (1, "one") ls;
val ls = [(1, "one")] : (tkey * string) list
- val ls = add (2, "two") ls;
val ls = [(1, "one"), (2, "two")] : (tkey * string) list
- find 1 ls;
val it = "one" : string
- find 2 ls;
val it = "two" : string
- find 3 ls;
uncaught exeception AlistExn
 raised at: alist1.sml:22.27-22.35
```

`structure` から `end` までは**ストラクチャ式**と呼ばれ，この中には関数宣言や型宣言などの様々な宣言を記述できる（これ以外の形式のストラクチャ式も存在する）．`structure ストラクチャ名 = ストラクチャ式` という形の宣言を記述する．

シグネチャ

対話環境でストラクチャの宣言を行うと，次のように出力されることが確認できる．

```
- use "alist1.sml";
[opening alist1.sml]
structure Alist :
  sig
    exception AlistExn
    type tkey = int
    type tval = string
    type alist = (tkey * tval) list
    val new : unit -> alist
    val add : ''a * 'b -> (''a * 'b) list -> (''a * 'b) list
    val exists : ''a -> (''a * 'b) list -> bool
    val find : ''a -> (''a * 'b) list -> 'b
```

```
  end
val it = () : unit
```

sig から end までがストラクチャ Alist の**シグネチャ**（Signature）である．シグネチャには，ストラクチャで公開される名前に関する情報が記述される．独自のシグネチャを定義することで，ストラクチャで公開される名前を選択することができる．

次のプログラムは，独自のシグネチャ ALIST を定義した例である．ストラクチャ Alist には ALIST によるシグネチャ制約を加えている．これによって，シグネチャに記述されていない名前 exists がストラクチャの外部には非公開となり，隠蔽される．

```
(* file: alist2.sml *)

(* シグネチャ ALIST の定義 *)
signature ALIST =
sig
  exception AlistExn
  eqtype tkey
  type tval
  type alist
  val new : unit -> alist
  val add : tkey * tval -> alist -> alist
  val find : tkey -> alist -> tval
end

(* ストラクチャ Alist の定義 *)
structure Alist : ALIST =  (* シグネチャ制約を付加 *)
struct
  (* alist1 と同じ *)
end
```

sig から end まではシグネチャ式と呼ばれ，この中に公開すべきストラクチャの仕様を記述する．シグネチャに名前をつけるには，signature シグネチャ名 = シグネチャ式 という形の宣言を記述する．

対話環境でこのプログラムを読み込むと，以下のように出力される．

```
- use "alist2.sml";
[opening alist2.sml]
signature ALIST =
  sig
    exception AlistExn
    eqtype tkey
    type tval
    type alist
    val new : unit -> alist
    val add : tkey * tval -> alist -> alist
    val find : tkey -> alist -> tval
  end
```

```
structure Alist : ALIST
val it = () : unit
```

出力されるシグネチャに名前 `exists` が現れていないことが確認できる．(`Alist.exist 1 ls` を評価させるとエラーが発生する)

シグネチャ制約の透明性

ストラクチャのシグネチャ制約に用いる `:` を `:>` に置き換えると，シグネチャ制約の透明性を変更できる．`:` による制約は**透明な**制約 (transparent constraint)，`:>` による制約は不透明な制約 (opaque constraint) と呼ぶ．

```
(* file: alist3.sml *)

signature ALIST =
sig
  (* alist2 と同じ *)
end

structure Alist :> ALIST =  (* 不透明なシグネチャ制約 *)
struct
  (* alist2 と同じ *)
end
```

これを対話環境で読み込むと，以下のように式を評価させる．

```
- use "alist3.sml";
- val ls = Alist.new ();
val ls = - : Alist.alist
```

透明な制約では連想リストの中身が表示され，連想リストの実装が通常のリストであることがわかる形になっているが，不透明な制約を用いることで，`alist` 型の実体を隠蔽することに成功している．ただし，この例の不透明なシグネチャ制約は他の型の実体も隠蔽するため，`tkey` 型と `int` 型，`tval` 型と `string` 型を同じ型と見なさなくなる．これを解決するには，次のように `where type` で始まる記述を追加し，選択的に型の実体を公開する．なお，この記述はシグネチャ式の直後に置くことができる．

```
(* file: alist4.sml *)

signature ALIST =
sig
  exception AlistExn
  eqtype tkey
  type tval
  type alist
  val new : unit -> alist
  val add : tkey * tval -> alist -> alist
```

```
  val find : tkey -> alist -> tval
end
  where type tkey = int      (* tkey は int と同じだよ *)
  where type tval = string   (* tval は string と同じだよ *)

structure Alist :> ALIST =   (* 不透明なシグネチャ制約 *)
struct
  (* alist2 と同じ *)
end
```

対話環境での実行結果は以下の通りである．alist 型の実体が隠蔽されつつ，tkey，tval 型が適切に実行環境に認識されていることが分かる．

```
- use "alist4.sml";
...

- val ls = Alist.new ();
val ls = - : Alist.alist

- val ls = Alist.add (1, "one") ls;
val ls = - : Alist.alist

- val ls = Alist.find 1 ls;
val ls = "one" : Alist.tval
```

ファンクタ

これまで見てきた Alist ストラクチャでは，tkey は int，tval は string に固定されていた．次はこれらをパラメータ化し，tkey と tval に任意の型をとることができるようにする．

ファンクタ（Functor）はストラクチャにパラメータを持たせたものである．以下の例では，2 つのパラメータを持つファンクタ Alist を定義している．

```
(* file: alist5.sml *)

(* シグネチャ ALIST の定義 *)
signature ALIST =
sig
  exception AlistExn
  eqtype tkey
  type tval
  type alist
  val new : unit -> alist
  val add : tkey * tval -> alist -> alist
  val find : tkey -> alist -> tval
end

(* ファンクタ Alist の定義 *)
functor Alist (eqtype tk type tv)
```

```
  :> ALIST                  (* 不透明なシグネチャ制約 *)
  where type tkey = tk      (* tkey は tk と同じだよ *)
  where type tval = tv  =   (* tval は tv と同じだよ *)
struct
  exception AlistExn

  type tkey = tk
  type tval = tv
  type alist = (tkey * tval) list

  fun new () = nil

  fun add (k,v) ls = if exists k ls then raise AlistExn else (k,v)::ls

  and exists key [] = false
    | exists key ((k,v)::ls) = k = key orelse exists key ls

  fun find key [] = raise AlistExn
    | find key ((k,v)::ls) = if k = key then v else find key ls
end

(* ファンクタからストラクチャを生成 *)
structure IntStrAlist = Alist (type tk = int type tv = string)
```

ファンクタの宣言は，`structure` の代わりに `structure` が使われることと，ファンクタ名の直後にパラメータリストが入ること以外は，基本的にストラクチャの宣言と同じである．ファンクタ名の直後の括弧 `( … )` の中には `sig … end` の中に書けるものと同じものを書くことができ，これがパラメータリストの役割を果たす．

ファンクタからストラクチャを生成するには，`Alist (type tk = int type tv = string)` のように，ファンクタ名に続いて括弧の中に宣言を記述する．この括弧の中には `struct … end` の中に書けるものと同じものを書くことができる．

このプログラムを対話環境で動かすと，以下のようになる．

```
- use "alist5.sml";
- open IntStrAlist;
- val ls = new ();
val ls = - : alist
- val ls = add (1, "one") ls;
val ls = - : alist
- find 1 ls;
val it = "one" : tval
```

4． コンパイラのフロントエンドの構成

Standard ML の基本文法と言語機能についての勉強会を数回行なった後，コンパイラの構成を学ぶための勉強会を行った．そこで得られた知見を以下にまとめる．

ある言語によるプログラムを他の言語によるプログラムへと変換するために，コンパイラは，最初にプログラムを分解してその構造と意味を理解した上で，別の方法で組み立てなければならない．コンパイラのフロントエンドが解析を実施し，バックエンドが合成を行う．解析の過程は，通常次の 3 つへとさらに分解される．

- 字句解析（Lexical analysis）: 入力を個々の単語，あるいはトークンへと分解する．
- 構文解析（Syntax analysis）: プログラムの句構造を解析する．
- 意味解析（Semantic analysis）: プログラムの意味を計算する．

5．後期の活動について

後期活動では，コンパイラのバックエンドの構成を学び，実際にコーディングを行ってコンパイラを作成する．

6．参考文献

- Andrew W. Appel 著，神林 靖 監修/編集，滝本 宗宏 編集 『最新コンパイラ構成技法』
- 『ウォークスルー Standard ML』 http://walk.northcol.org/sml/
- 『お気楽 Standard ML of New Jersey 入門』
  http://www.geocities.jp/m_hiroi/func/index.html#sml