2. Wykład 2: Programowanie funkcyjne: datatypy, pattern matching, funkcje wyższego rzędu, wyjątki i moduły.

2.1. **Datatypy.** Wcześniej poznaliśmy pobieżnie kilka podstawowych typw w SML-u. Obecnie podamy pełną listę podstawowych typów w SML-u, jakich będziemy używać w przyszłości:

- `bool`: boole'owskie wartości `true` oraz `false`;
- `int`: liczby całkowite, na przykład `0, 1, ~1` itd.;
- `real`: liczby rzeczywiste, czyli jak kto woli zmiennoprzecinkowe, na przykład `0.0, 1.0, ~1.0` itd.;
- `char`: znaki, na przykład `#"a", #"b", #" "` itd.;
- `string`: ciągi znaków, na przykład `"hello", "newline\n", "quote"` itd.;
- $A->B$: funkcje z typu A w typ B;
- $A*B$: pary typów A i B; jeżeli e_1 jest typu A, zaś e_2 jest typu B, to wówczas (e_1, e_2) jest typu $A * B$, na przykład $(0, \texttt{true}) : \texttt{int} * \texttt{bool}$; $A*B$ nazywamy **produktem typów;**
- $A_1*A_2*\ldots*A_n$: uporządkowane n-ki typów $A_1, A_2, \ldots, A_n$, na przykład `(0, true, 1.0): int * bool * real;` oczywiście uporządkowane n-ki typów są uogólnieniem produktów typów;
- `unit`: wartość jednostkowa `()`; jedyna wartość przypisana do typu `unit` to `()`; typ ten przydaje się na przykład wtedy, gdy deklarujemy funkcję, dla której nie interesuje nas typ argumentu – funkcją taką przykładowo może być funkcja o typie `unit ->int.`

Opisane powyżej typy w zupełności wystarczają do rozwiązania większości problemów opartych o obliczenia numeryczne. Z drugiej strony jest całe mnóstwo problemów, w których zastosowanie obliczeń symbolicznych wydaje się bardziej odpowiednie, niż obliczeń numerycznych. Jako przykład rozważmy problem polegający na klasyfikowaniu obrazków w trzech kategoriach: kółek, kwadratów i trójkątów. Oczywiście możemy przypisać liczby całkowite `1, 2, 3` do opisu kształtów i przeprowadzić ich klasyfikację za pomocą funkcji o typie `image ->int` (gdzie `images` jest typem obrazków). Wadą takiego rozwiązania jest to, że w istotny sposób ogranicza ono możliwość utrzymywania kodu: nie ma bezpośredniego związku między kształtami a typem `int`, zaś programista musi cały czas pamiętać, które zmienne typu `int` opiusują kształty, a które liczby całkowite.

O wiele lepszym podejściem jest reprezentowanie każdego kształtu przez stałą symboliczną. W SML-u możemy użyć deklaracji **datatypu** do zdefiniowania nowego typu `shape` dla trzech stałych symbolicznych:

```
datatype shape = Circle | Square | Triangle
```

Każda z powyższych stałych symbolicznych ma teraz typ `shape.` Na przykład:

```
- Circle;
val it = Circle : shape
```

Odnotujmy tu, że datatypy są specjalnym sposobem definiowania nowych typów i tym samym tworzą tylko podzbiór zbioru typów. Na przykład typ funkcyjny `int ->int` jest typem, ale nie datatypem, podczas gdy każdy datatyp jest także typem.

Deklaracja datatypu jest podobna do typu zliczającego (*enumeration type*) w języku C, wszelako z jedną istotną różnicą: stałe symboliczne nie są kompatybilne z liczbami całkowitymi i nie mogą być podstawione za liczby całkowite. Tym samym podejście oparte na datatypach nie jest obarczone wadami, o których wspominaliśmy powyżej. Stałe symboliczne będziemy często nazywami **konstruktorami danych** w SML-u, lub po prostu **konstruktorami.**

Kolejną cechą datatypów w SML-u odróżniającą je od typów zliczających w C jest to, że konstruktory mogą zależeć od argumentów. Na przykład, możemy zmodyfikować powyższą deklarację datatypu dodając argumenty do konstruktorów:

```
datatype shape =
  Circle of real
| Square of real
| Triangle of real * real * real
```

Chcąc teraz utworzyć wartości o typie `shape`, musimy podać odpowiednie argumenty dla konstruktorów:

```
- Circle 1.0;
val it = Circle 1.0 : shape
- Square 1.0;
val it = Square 1.0 : shape
- Triangle (1.0, 1.0, 1.0);
val it = Triangle (1.0,1.0,1.0) : shape
```

Odnotujmy tu, iż każdy z konstruktorów może być postrzegany jak funkcja z typu danego argumentu do typu `shape`. Na przykład na `Circle` możemy patrzeć jak na funkcję typu `real ->shape`:

```
-Circle;
val it = fn : real ->shape
```

Przedyskutejemy teraz dwa ważne rozszerzenia mechanizmu deklarowania datatypów. Jako motywację do pierwszego rozszerzenia, o którym zaraz będziemy mówić, rozważmy datatyp `pair_bool` służący do dobierania w pary wartości boole'owskich oraz datatyp `pair_int` służący do dobierania w pary liczb całkowitych:

```
datatype pair_bool = Pair of bool * bool
datatype pair_int = Pair of int * int
```

Powyższe dwie deklaracje mają identyczną strukturę, ale różnią się typami argumentów. Wcześniej widzieliśmy, w jaki sposób deklaracje funkcji o tej samej strukturze, ale o argumentach różnych typów, mogą być wprowadzone za pomocą pojedynczej deklaracji z użyciem zmiennych typowych. W przypadku datatypów możemy zrobić dokładnie to samo: dla każdego typu A możemy zadeklarować nowy datatyp `datatype pair_`A w dokładnie taki sam sposób:

```
datatype pair_A = Pair of A * A
```

Zgodnie z syntaksem SML-a dla parametryzowania deklaracji datatypów zmiennymi typowymi, zmienne typowe umieszczamy przed nazwą datatypu tak, aby zaznaczyć, które zmienne typowe są lokalne dla danej deklaracji typu. Oto garść przykładów:

```
datatype 'a pair = Pair of 'a * 'a
datatype ('a, 'b) hetero = Hetero of 'a * 'b

- Pair (0, 1);
val it = Pair (0,1) : int pair
- Pair (0, true);
stdIn:5.1-5.15 Error: <tu wyskoczy jakaś error message>
- Hetero (0, true);
val it = Hetero (0,true) : (int,bool) hetero
```

Drugi rodzaj rozszerzenia mechanizmu deklaracji datatypu pozwala na użycie datatypu jako typu dla argumentów użytych przez jego własne konstruktory. Tak jak w przypadku funkcji rekursywnych, potrzebny jest do tego przynajmniej jeden konstruktor (odpowiadający przypadkowi bazowemu dla funkcji rekursywnych), który nie używa tego samego datatypu dla swoich argumentów – bez niego nie byłoby możliwe zbudowanie wartości należących do datatypu. Tego rodzaju datatypu na ogół nazywane są **rekursywnymi datatypami** i pozwalają na implementowanie rekursywnych struktur danych. Jako

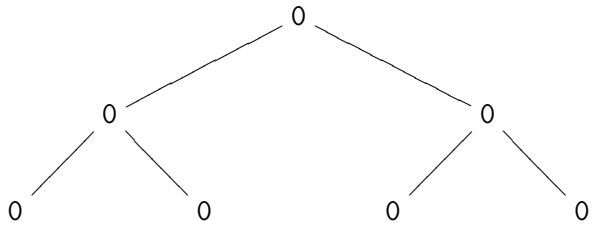
przykład rozważmy datatyp `itree` drzew binarnych, których węzły etykietowane są liczbami całkowitymi:

```
datatype itree =
  Leaf of int
| Node of itree * int * itree
```

Konstruktor `Leaf` odpowiada liściowi drzewa, który traktujemy jak węzeł etykietowany liczbą naturalną typu `int`; konstruktor `Node` odpowiada zwykłemu węzłowi, który zawiera dwa poddrzewa i liczbę całkowitą. Na przykład:

`Node (Node (Leaf 0, 0, Leaf 0), 0, Node ( Leaf 0, 0, Leaf 0))`

odpowiada następującemu drzewku:



Zauważmy, że bez użycia `Leaf` nie jest możliwe zbudowanie wartości typu `itree`.

Opisane powyżej dwa rodzaje rozszerzania mechanizmu deklarowania datatypów zastosowane wspólnie pozwalają na definiowanie **datatypów rekursywnych ze zmiennymi typowymi.** Na przykład datatyp `itree` opisany powyżej może zostać uogólniony do datatypu `tree` drzew binarnych o wartościach dowolnego typu:

```
datatype 'a tree =
  Leaf of 'a
| Node of 'a tree * 'a * 'a tree
```

Teraz argumenty dla konstruktorów `Leaf` oraz `Node` automatycznie określają typ, jaki powinien być użyty przez zmienną typową `'a`:

```
- Node (Leaf ~1, 0, Leaf 1);
val it = Node (Leaf ~1,0,Leaf 1) : int tree
- Node (Leaf "L", "C", Leaf "R");
val it = Node (Leaf "L","C",Leaf "R") : string tree
```

Zaobserwujmy, że gdy zmienna typowa `a'` zostanie przypisana do specyficznego typu (powiedzmy `int`), wartości innych typów (powiedzmy `string`) nie będą mogły być użyte. Innymi słowy, `tree` może być użyte tylko do budowy jednorodnych drzew binarnych. Poniższe wyrażenie nie skompiluje się, ponieważ nie określa jednoznacznie typu dla `'a`:

```
- Node (Leaf ~1, "C", Leaf 1);
stdIn:35.1-35.28 Error: <tu wyskoczy jakaś error message>
```

Rekursywne struktury danych spotykane są dość często zarówno w programowaniu funkcyjnym jak i imperatywnym. W połączeniu z parametrami typów, rekursywne datatypy w SML-u pozwalają programistom zaimplementować większość rekursywnych struktur danych w zwięzły i elegancki sposób. Co ważniejsze, rekursywne struktury danych zaimplementowane w SML-u pozwalają użyć kompilatora do zreorganizowania niektórych ich niezmienników, czyli pewnych warunków, które muszą zachodzić dla wszystkich przypadków wystąpienia danej struktury danych. Na przykład, odwołując się do podanego powyżej przykładu datatypu `tree`, kompilator SML-a rozpoznaje jako niezmiennik fakt, że każdy zwykły węzeł musi mieć dwa poprzedniki, które też są węzłami. Taki niezmiennik byłby trudny do wymuszenia

w programowaniu imperatywnym (czy Czytelnik potrafiłby zaimplementować w C lub C++ datatyp drzew binarnych, w których każdy zwykły węzeł ma dokładnie dwa poprzedniki?).

Na zakończenie tej sekcji opiszemy pokrótce najczęściej używany datatyp w językach funkcyjnych, a mianowicie listy. Listy obsługiwane są przez wbudowany datatyp `list` z dwoma konstruktorami: unarny konstruktor `nil` oraz binarny konstruktor `: ::`

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

`nil` oznacza pustą listę o dowolnym typie (jako że nie ma argumentu). `::` jest łącznym z prawej strony operatorem przyjmującym swoje argumenty z lewej i z prawej strony, który buduje listę typu `'a` przez sklejenie tego, co stoi po lewej stronie o typie `'a` z tym, co z stoi po prawej stronie o typie `'a`. Na przykład poniższe wyrażenie opisuje listę składającą się z elementów `1, 2, 3` wypisanych w takiej właśnie kolejności:

```
1 :: 2 :: 3 :: nil
```

Innym sposobem na stworzenie list w SML-u jest ponumerowanie ich elementów oddzielonych przecinkami `,` z nawiasami `[` oraz `]`. Na przykład `[1, 2, 3]` jest skrótem listy podanej powyżej. Obydwie notacje mogą pojawiać się równocześnie w obrębie jednej listy. Na przykład poniższe wyrażenia wszystkie są równoważne:

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

2.2. **Pattern matching.** Dotychczas zobaczyliśmy, w jaki sposób można tworzyć wyrażenia o różnych typach w SML-u. Równie ważna jest kwestia sprawdzania wartości, do których dane wyrażenia się wartościują. Dla prostych typów takich jak liczby całkowite czy uporządkowane n-ki typów, kwestia ta jest prosta do rozstrzygnięcia i wymaga jedynie odwołania się do operatorów już dostępnych w SML-u. Na przykład możemy użyć prostej arytmetyki i operatorów porównywania liczb naturalnych do sprawdzenia, czy dana liczba naturalna należy do danego przedziału; dla uporządkowanych n-ek typów (w szczególności dla produktów typów), możemy użyć operatora rzutowania $\sharp n$ do otrzymania n-tego elementu danej n-ki (na przykład `#2(1, 2, 3)` wartościuje się do `2`). Wszelako w przypadku datatypów, potrzebujemy sposobu sprawdzania który konstruktor został zastosowany w tworzeniu wartości o danym datatypie. Narzędziem umożliwiającym to w SML-u jest **pattern matching.**

Jako przykład rozważmy funkcję `length` obliczającą długość danej listy. Funkcja ta działa na zasadzie prostej rekursji:

- jeśli argumentem jest `nil`, zwraca `0`;
- jeśli argument ma postać <`head`>`::` <`tail`>, wywołuje `length` na <`tail`> i zwraca rezultat powiększony o `1`.

Tym samym `length` próbuje porównać daną listę z wystąpieniami `nil` oraz `::` w dowolnej kolejności. Ponadto, gdy dana lista zostanie porównana z jakimś wystąpieniem `::`, funkcja będzie musiała także rozpoznać argumenty dla `::` tak, aby mogła samą siebie wywołać po raz kolejny. Powyższa definicja tłumaczy się zatem na następujący fragment kodu wykorzystujący pattern matching (który, przy odrobinie wyobraźni, trochę przypomina konstruktor `switch` znany z C):

```
fun length l =
  case l of
    nil =>0
  | head :: tail =>1 + length tail
```

Odnotujmy tu, że `nil` nie jest wartością; jesr raczej wzorem (pattern), który należy porównać z `l` (czy też cokolwiek innego stoi po słowie kluczowym `case`). W podobny sposób `head :: tail` jest wzorem, który po porównaniu z listą `l` wiąże `head` z tym, co występuje po lewej stronie pierwszego wystąpienia `::`, a `tail` – z tym, co po prawej. Jeżeli takie porównanie jest możliwe do przeprowadzenia, to całe wyrażenie `case` redukuje się do wyrażenia stojącego po prawej stronie odpowiadającego `=>`. Będziemy nazywać `nil` oraz `head :: tail` **konstruktorami wzorów** z powodu ich użycia konstruktorów datatypów.

Zadanie 2. *Jakiego typu jest* `length`*?*

W przypadku `length` nie potrzebujemy `head` w drugim wzorze. W takim wypadku, w którym kolejne wzory podobnie się ze sobą nie wiążą, możemy również użyć **wzoru podkreślnika** `_`:

```
fun length l =
  case l of
    nil =>0
  | _ :: tail =>1 + length tail
```

`_` jako taki jest dobrze zdefiniowanym wzorem, który przydaje się na przykład wtedy, gdy nie trzeba rozważać wszystkich konstruktorów. Na przykład funkcja sprawdzająca, czy dana lista jest pusta może zostać zaimplementowana w następujący sposób:

```
fun testNil l =
  case l of
    nil = >true       (* jeżeli l jest pusta *)
  | =>false       (* we wszystkich pozostałych przypadkach *)
```

Niezależnie od konstruktorów związanych z datatypem `list,` powyższa analiza możliwych przypadków jest wystarczająca, jako że `_` porównuje się z dowolną wartością.

Na pattern matching w SML-u można patrzeć ja na uogólnioną wersję konstruktora warunkowego `if-then-else.` Istotnie, pattern matching można stosować do dowolnych typów, nie tylko do datatypów – na przykład `if` e `then` e_1 `else` e_2 można rozwinąć jako

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

lub równoważnie

```
case e of
  true =>e1
| _ =>e2
```

Okazuje się, że wszystkie zmienne w SML-u są również specjalną postacią wzorów, co z kolei implikuje, iż każda zmienna może być zastąpiona przez inny wzór. Poznaliśmy dwa sposoby wprowadzania zmiennych w SML-u: za pomocą słowa kluczowego `val` oraz w deklaracjach funkcji. Wobec tego to, co znajduje się bezpośrednio po słowie kluczowym `val` nie musi być pojedynczą zmienną, ale ogólniej dowolną postacią wzoru; podobnie argumenty formalne w deklaracji funkcji mogą być dowolną postacią wzoru. Jako pierwszy przykład zauważmy, że prostym sposobem odczytania poszczególnych elementów uporządkowanej n-ki jest wykorzystanie wzoru n-ki (zamiast wielokrotne używanie operatora rzutowania $\sharp n$):

```
val (x, y, z) = <jakaś uporządkowana n-ka>
```

Możemy także użyć konstruktora wzoru `head :: tail` do porównywania z listą:

```
val (head :: tail) = [1, 2, 3];
```

Tutaj `head` zostaje przypisane do `1` a `tail` do `[2, 3]`. Zauważmy wszakże, że taki wzór nie jest wyczerpujący: gdyby, na przykład, `nil` znajdowało się po prawej stronie listy, nie można by porównać `head ::`

`tail` z `nil`. Wrócimy jeszcze do opisu takich patologicznych przypadków. Jako kolejny przykład możemy przepisać wzajemnie rekursywne funkcje `even` oraz `odd` przy użyciu pattern matching:

```
fun even 0 = true
  | even n = odd (n - 1)
and odd 0 = false
  | odd n = even (n - 1)
```

2.3. **Funkcje wyższego rzędu.** Jak dotychczas zauważyliśmy, funkcje w SML-u są obiektami pierwszego rzędu – mogą być przekazane jako argument do innej funkcji i zwrócone jako rezultat aplikacji funkcji. Tym samym funkcja, która jako argument przyjmuje inną funkcję lub też zwraca jako wartość inną funkcję jest typu funkcyjnego $A->B$, gdzie A i B same zawierają typy funkcyjne. O tego typu funkcjach będziemy mówili, że są **funkcjami wyższego rzędu**. Na przykład funkcje o typach `(int ->) ->int` albo `int ->(int ->int)` są funkcjami wyższego rzędu. Oczywiście możemy też użyć zmiennych typowych w funkcjach wyższego rzędu. Na przykład `('a ->'b) ->('b ->'c) ->('a ->'c)` jest typem funkcyjnym wyższego rzędu.

Funkcje wyższego rzędu potrafią istotnie uprościć wiele zagadnień programistycznych, o ile są odpowiednio użyte. Jako przykład rozważmy funkcję wyższego rzędu `map` o typie `('a ->'b) ->'a list ->'b list`. Funkcja ta bierze funkcję f o typie `'a ->'b` oraz listę l o typie `'a list`, a następnie aplikuje f do każdego elementu l tworząc nową listę o typie `'b list`. Na przykład możemy zastosować funkcję `map` do funkcji `even` oraz `odd` zdefiniowanych powyżej:

```
- map even [1, 2, 3, 4];
val it = [false,true,false,true] : bool list
- map odd [1, 2, 3, 4];
val it = [true,false,true,false] : bool list
```

Zachowanie funkcji `map` można formalnie opisać w następujący sposób:

$$\texttt{map}f[l_1, l_2, \ldots, l_n] = [fl_1, fl_2, \ldots, fl_n] \qquad (n \geq 0) \tag{1}$$

Chcąc zaimplementować funkcję `map`, zapiszemy równanie (1) indukcyjnie rozbijając je na dwa przypadki: przypadek bazowy $n = 0$ oraz krok indukcyjny $n > 0$. Przypadek bazowy jest trywialny, jako że wówczas prawa strona równania jest pustą listą:

$$\texttt{map}f[] = [] \tag{2}$$

Z kolei krok indukcyjny łatwo zapisać korzystając z obserwacji, że $[fl_2, fl_3, \ldots, fl_n]$ można otrzymać z kolejnej aplikacji funkcji `map`:

$$\texttt{map}f[l_1, l_2, \ldots, l_n] = fl_1::f[l_2, \ldots, l_n] \qquad (n > 0) \tag{3}$$

Tym samym równania (2) i (3) pozwalają zapisać następującą definicję funkcji `map`:

```
fun map f [] = []
  | map f (head :: tail) = f head :: map f tail
```

Funkcja `map` przerabia elementy danej listy niezależnie od siebie. Jako kolejny przykład rozważmy funkcję `foldl` "zwijającą" elementy listy w lewo, która będzie przerabiała elementy danej listy sekwencyjnie z użyciem rezultatu przerabiania poprzedniego elementu do przerabiania następnego elementu. Funkcja ta będzie typu `('a * 'b ->'b) ->'b ->'a list ->'b`, gdzie `'b` oznacza typ rezultatu przerobienia elementu. Zachowanie tej funkcji można formalnie zapisać równaniem:

$$\texttt{foldl}fa_0[l_1, l_2, \ldots, l_n] = f(l_n, \ldots f(l_2, f(l_1, a_0)) \ldots) \qquad (n \geq 0) \tag{4}$$

O a_0 możemy myśleć jak o początkowej wartości akumulatora, którego wartość zmienia się w miarę, jak elementy listy są kolejno przerabiane. Tym samym równanie (4) może być rozwinięte jako następująca lista równań:

$$\begin{aligned} a_1 &= f(l_1, a_0) \\ a_2 &= f(l_2, a_1) \\ &\vdots \\ a_n &= f(l_n, a_{n-1}) = \texttt{foldl}\, f\, a_0 [l_1, l_2, \ldots, l_n]. \end{aligned}$$

Tak jak w przypadku `map`, zaimplementujemy `foldl` przepisując równanie (4) indukcyjnie. Przypadek bazowy zwraca początkową wartość a_0 akumulatora:

$$\texttt{foldl}\, f\, a_0 [] = a_0 \tag{5}$$

Przypadek indukcyjny odwołuje się rekursywnie wraz z nową wartością akumulatora:

$$\texttt{foldl}\, f\, a_0 [l_1, l_2, \ldots, l_n] = \texttt{foldl}\, f (f(l_1, a_0)) [l_2, l_3, \ldots, l_n] \qquad (n > 0) \tag{6}$$

Równania (5) i (6) dają następującą definicję `foldl`:

```
fun foldl f a [] = a
  | foldl f a (head :: tail) = foldl f (f (head :: a)) tail
```

Jako przykład możemy otrzymać sumę liczb całkowitych z listy z odwołaniem do `foldl`:

```
- foldl (fn (x, y) =>x + y) 0 [1, 2, 3, 4];
val it = 10 : int
```

Zadanie 3. *Przykładem podobnej funkcji wyższego rzędu jest funkcja* `foldr` *"zwijająca w prawo", która jest tego samego typu co* `foldl`*, ale obrabia daną listę zaczynając od jej ostatniego elementu, a kończąc na pierwszym:*

$$\texttt{foldr}\, f\, a_0 [l_1, l_2, \ldots, l_n] = f(l_1, \ldots f(l_{n-1}, f(l_n, a_0)) \ldots) \qquad (n \geq 0)$$

Zaimplementuj `foldr`*.*

2.4. **Wyjątki.** Wyjątki w SML-u dostarczają wygodnego mechanizmu obchodzenia się z błędnymi warunkami, jakie mogą się pojawić podczas obliczeń. Wyjątek jest generowany, czy **zgłaszany,** albo przez system, gdy zostanie napotkany błędny warunek, albo przez samych programistów celem przeniesienia kontroli do innej części programu. Na przykład system zgłosi wyjątek, gdy napotka na dzielenie przez zero, albo gdy żaden wzór w wyrażeniu `case` nie pasuje do danej wartości; programista może zechcieć zgłosić wyjątek, gdy argument funkcji nie będzie spełniał niezmiennika funkcji. Wyjątek może być **wychwycony** przez **obsługę wyjątków,** która analizuje wyjątek celem zadecydowania, czy należy zgłosić kolejny wyjątek, czy też powrócić do obliczeń. Tym samym nie każdy wyjątek prowadzi do przerwania obliczeń.

Wyjątek jest konstruktorem danych należącym do specjalnego wbudowanego datatypu `exn`, którego zbiór konstruktorów może być dowolnie rozszerzony przez programistów. Deklaracja wyjątku składa się z deklaracji konstruktora danych poprzedzonego słowem kluczowym `exception`. Na przykład, możemy zadeklarować wyjątek `Error` za pomocą następującego ciągu znaków:

```
exception Error of string
```

Aby zgłosić `Error`, używamy słowa kluczowego `raise`:

```
raise Error "Komunikat dla Error"
```

Jako że wyjątki są konstruktorami dla specjalnego datatypu `exn`, syntaks dla obsługi wyjątków również korzysta z pattern matching:

e `handle`
  <`pattern`$_1$>=>e_1
| ...
| <`pattern`$_n$>=>e_n

Jeżeli zostanie zgłoszony wyjątek podczas wartościowania wyrażenia e, to wówczas < `pattern`$_1$ > do < `pattern`$_n$ > są sprawdzane pod kątem dopasowania do wzorca. Jeśli < `pattern`$_i$ > odpowiada wyjątkowi, to wówczas e_i staje się nowym wyrażeniem do wartościowania; jeśli żaden wzorzec nie odpowiada wyjątkowi, to wówczas wyjątek zostaje przekazany do dalszej obsługi wyjątków, o ile taka istnieje.

Jako przykład rozważmy następujący fragment kodu:

```
exception BadBoy of int;
exception BadGirl of int;
1 + (raise BadGirl ~1) handle
BadBoy s =>(s * s)
| BadGirl s =>(s + s)
```

Podczas próby wartościowania drugiego operanda +, zostaje zgłoszony wyjątek `BadGirl` z argumentem `~1`. W następstwie proces wartościowania zostaje wstrzymany i wyrażenie jest przekazane do dalszej obsługi wyjątków. Jeśli drugi ze wzorców będzie odpowiadał przekazanemu wyjątkowi, wyrażenie `s+s` po prawej stronie `=>` staje się nowym wyrażeniem do wartościowania. Wraz z `s` zastąpionym przez argument `~1` do `BadGirl`, całe wyrażenie wartościuje się do `~2`.

Wyjątki okazują się użyteczne w całym szeregu sytuacji. Nawet w pełni rozwinięte programy często wykorzystują mechanizm wyjątków do obchodzenia się z przypadkami szczególnymi. Na przykład, gdy czasochłonne obliczenie zostaje przerwane przez dzielenie przez zero, mechanizm wyjątków może zostać wykorzystany do uratowania częściowego wyniku. Oto kilka innych przykładów zastosowań wyjątków w programowaniu funkcyjnym:

- Przy opracowywaniu programu, w którym funkcja `f` nie może być wywołana z ujemną liczbą całkowitą jako argumentem, wyjątek zostaje zgłoszony w punkcie startowym wywoływania `f`, gdy jej argument jest ujemną liczbą całkowitą.
- Wszystkie programy w SML-u, jakie będziecie oddawali jako zadania domowej, powinny się kompilować. Jak poradzić sobie w sytuacji, gdy udało Wam się zaimplementować funkcję `funEasy`, ale nie poradziliście sobie z funkcją `funHard`? Można to zrobić w następujący sposób:
  ```
  exception NotImplemented
  fun funHard _ = raise NotImplemented
  ```
  Powyższy trik działa, ponieważ `raise NotImplemented` jest typu `'a`.

2.5. **Moduły.** Programowanie modularne jest metodologią w programowaniu, w której duży program jest podzielony na niezależne mniejsze jednostki. Każdy z nich zawiera zbiór powiązanych funkcji, typów etc. które mogą być łatwo użyte w innych zadaniach programistycznych. SML dostarcza rozbudowane zaplecze dla programowanie modularnego za pomocą **struktur** i **sygnatur**. Struktura, czyli jednostka programowania modularnego w SML-u, jest kolekcją deklaracji zgodnych ze specyfikacjami zadanymi przez daną sygnaturę.

Struktura jest kolekcją funkcji, typów, wyjątków i innych elementów zawartych wewnątrz konstrukcji `struct -- end`; sygnatura jest kolekcją specyfikacji powyższych deklaracji zawartych wewnątrz konstrukcji `sig -- end`. Na przykład struktura po lewej stronie poniżej odpowiada sygnaturze po prawej:

```
struct                                  sig
  type 'a set = 'a list                   type 'a set
  val emptySet : 'a set = nil             val emptySet : 'a set
  fun singleton x = [x]                   val singleton : 'a ->'a set
  fun union s1 s2 = s1 @ s2               val union : 'a set ->'a set ->'a set
end                                     end
```

Pierwsza linia kodu w sygnaturze wymaga aby deklaracja typu `'a set` została podana w odpowiadającej jej strukturze; każda deklaracja typu, której rezultatem jest nowy typ `'a set` jest tu do zaakceptowania. W powyższym przykładzie używamy deklaracji typu z wykorzystaniem słowa kluczowego *type*, ale deklaracja datatypu taka jak

```
datatype 'a set = Empty | Singleton of 'a | Union of 'a set * 'a set
```

jest również do zaakceptowania, o ile inne elementy w tej samej strukturze są tak samo przedefiniowane. Druga linia kodu w sygnaturze mówi, że zmienna `emptySet` typu `'a set` musi być zdefiniowana w odpowiadającej jej strukturze. Struktura definiuje zmienną `emptySet` typu 'a list, która odpowiada `'a set` w definicji `'a set`. Trzecia linia w sygnaturze oznacza, że zmienna `singleton` typu `'a ->'a set`, lub, równoważnie, funkcja `singleton` typu `'a ->'a set` musi być zdefiniowana w odpowiadającej jej strukturze. Znowu, `singleton` w strukturze jest typu `'a ->łist`, który jest równy `'a ->'a set` w definicji `'a set`. Przypadek `union` jest podobny, przy czym występujący tam operator `@` pełni funkcję dopisywania jednej listy do drugiej.

Tak jak w przypadku zwykłych wartości, struktury i sygnatury mogą mieć nadane nazwy. Używamy słów kluczowych `structure` i `signature` następująco:

```
structure Set =                         signature SET =
struct                                  sig
...                                     ...
end                                     end
```

Dostęp do elementów struktury `Set` może być osiągnięty przy użyciu notacji `.` podobnej do tej, jakiej używamy w języku C (na przykład `Set.set, Set.emptySet` itd.)

W jaki sposób możemy wyspecyfikować to, że struktura `Set` odpowiada sygnaturze `SET`? Jednym ze sposobów, aby to osiągnąć, jest wymuszenie **transparentnego ogranicznika** pomiędzy `Set` i `SET` przy użyciu dwukropka (`:`):

```
structure Set : SET = ...
```

Ograniczenie przez `:` mówi, że `Set` odpowiada `SET`; program nie będzie się kompilował, jeżeli `Set` nie będzie implementował jakiejś specyfikacji w `SET`. Innym sposobem jest wymuszenie **odwrotnego ogranicznika** pomiędzy `Set` oraz `SET` przy użyciu symbolu `:>`:

```
structure Set :> SET = ...
```

Ogranicznik `:>` mówi nie tylko to, że `Set` odpowiada `SET`, ale także to, że wyłącznie te deklaracje typów, jakie wyszczególnione są w `SET` będą widoczne na zewnątrz. Aby lepiej zrozumieć tę różnicę, przyjrzyjmy się następującemu fragmentowi kodu:

```
signature S = sig
  type t
end
```

```
structure Transparent : S =
struct
  type t = int
  val x = 1
end

structure Opaque :>S =
struct
  type t = int
  val x = 1
end
```

Po pierwsze odnotujmy, że zarówno struktura `Transparent` jak i `Opaque` odpowiadają sygnaturze `S`. Jako że `S` nie deklaruje zmiennej `x`, nie ma możliwości dostępu poprzez `Transparent.x` oraz `Opaque.x`. Różnica pomiędzy `Transparent` i `Opaque` leży w widoczności definicji typu `t`. W przypadku `Transparent`, definicja `t` jako `int` jest eksportowana na zewnątrz. Tym samym następująca deklaracja jest do zaakceptowana, ponieważ wiadomo, że `Transparent.t` jest w istocie `int`:

```
- val y : Transparent.t = 1;
val y = 1 : Transparent.t
```

W przypadku `Opaque` definicja `t` pozostaje nieznana dla świata zewnętrznego, co spowoduje, że następująca deklaracja zostanie odrzucona:

```
- val z : Opaque.t = 1;
stdIn:3.5-3.21 Error: <jakaś wiadomość o błędzie>
```

Odwrotny ogranicznik w SML pozwala programistom uzyskać odpowiedni poziom abstrakcji danych ukrywając szczegóły ich implementacji w strukturze. Celem użycia struktur o zadanych odwrotnych ogranicznikach (na przykład tych, które zawarte są w bazowej bibliotece SML-a lub zostały napisane przez innych programistów) potrzeba wyłącznie odczytać ich sygnatury aby zobaczyć, jakie wartości są eksportowane. Dość często spotyka się zatem dokładną dokumentację w sygnaturach, której towarzyszy całkowity brak dokumentacji struktur.

SML dostarcza również nowego narzędzia, zwanego **funktorami,** o którym można myśleć jak o funkcjach zdefiniowanych na strukturach. Funktor bierze na wejściu strukturę o pewnej sygnaturze i generuje na wyjściu nową strukturę wyspecializowaną zgodnie ze strukturą na wejściu. Jako że wszystkie struktury generowane przez funktor dzielą ten sam fragment kodu z definicji, zastosowanie funktora rozszerza możliwości ponownego wykorzystania kodu w programowaniu modularnym.

Celem zilustrowania użycia funktorów, rozważmy sygnaturę zbiorów i wartości wraz z relacją porządku:

```
datatype order = LESS | EQUAL | GREATER

signature ORD_SET =
sig
  type item                              (* typ elementów *)
  type set                               (* typ zbiorów *)
  val compare : item * item ->order      (* relacja porządku *)
  val empty : set                        (* zbiór pusty *)
  val add : set ->item ->set             (* dodaj element *)
  val remove : set ->item ->set          (* usuń element *)
end
```

Funkcja `compare` porównuje dwie wartości typu `item` celem określenia ich relatywnego rozmiaru (tzn. mniejsze, równe lub większe) i tym samym specyfikuje relacje porządkującą o typie `item`. Struktura implementująca sygnaturę `ORD_SET` może wykorzystać taką relację porządkującą na typie `item`. Na przykład, może zdefiniować `set` jako `item list` wraz z niezmiennikiem, zgodnie z którym wartości w każdym zbiorze są przechowywane w porządku wstępującym zgodnie z `compare` i wykorzystują ten niezmiennik podczas implementowania operacji na `set`.

Rozważmy teraz dwie struktury o sygnaturze `ORD_SET`:

```
structure IntSet : ORD_SET =
struct
  type item = int
  type set = item list
  fun compare (x, y) =
    if x<y then LESS
    else if x>y then GREATER
    else EQUAL
  val empty = []
  fun add s x = ...
  fun remove s x = ...
end
```

```
structure StringSet : ORD_SET =
struct
  type item = string
  type set = item list
  fun compare (x, y) =
    if String.<(x, y) then LESS
    else if String.>(x, y) then GREATER
    else EQUAL
  val empty = []
  fun add s x = ...
  fun remove s x = ...
end
```

Jeżeli dwie struktury założą ten sam niezmiennik na typie `set` (np. wartości mają być przechowywane w rosnącym porządku), kod dla funkcji `add` oraz `remove` może być identyczny w obydwu strukturach. Wówczas obydwie struktury mogą zawierać ten sam kod za wyjątkiem definicji o typie `item` i funkcji `compare`. Funktory rozszerzają zakres powtórnego wykorzystania kodu poprzez umożliwienie programistom napisania wspólnego fragmentu kodu dla obydwu struktur za jednym razem.

Poniższy funktor generuje `IntSet` oraz `StringSet` przy danych na wejściu odpowiednich strukturach. W pierwszej kolejności definiuje sygnaturę `ORD_KEY` dla struktur na wejściu celem dostarczenia typów lub wartości specyficznych dla `IntSet` i `StringSet`:

```
signature ORD_KEY =
sig
  type ord_key
  val compare : ord_key * ord_key ->order
end
```

Funktor `OrdSet` bierze strukturę `OrdKey` o sygnaturze `ORD_KEY` i generuje strukturę o sygnaturze `ORD_SET`:

```
functor OrdSet (OrdKey : ORD_KEY) : ORD_SET =
struct
  type item = OrdKey.ord_key
  type set = item list
  val compare = OrdKey.compare
  val empty = []
  fun add _ _ = ...
  fun remove _ _ = ...
end
```

Celem wygenerowania `IntSet` oraz `StringSet`, potrzebujemy odpowiadających im struktur o sygnaturze `ORD_KEY`:

```
structure IntKey : ORD_KEY =
struct
  type ord_key = int
  fun compare (x, y) =
    if x <y then LESS
    else if x >y then GREATER
    else EQUAL
end
```

```
structure StringKey : ORD_KEY =
struct
  type ord_key = string
  fun compare (x, y) =
    if String.<(x, y) then LESS
    else if String.>(x, y) then GREATER
    else EQUAL
end
```

Przy danych `IntKey` oraz `StringKey` na wejściu, `OrdSet` wygeneruje odpowiadające struktury o sygnaturze `ORD_SET`:

```
structure IntSet = OrdSet (IntKey)
structure StringSet = OrdSet (StringKey)
```