1. Wykład 1: Wprowadzenie do programowania funkcyjnego.

W wykładzie tym zajmiemy się podstawowymi ideami stojącymi u podstaw programowania funkcyjnego, czyli programowania w językach funkcyjnych. Wszystkie podane tu przykłady napisane są w Standard ML (który odtąd będziemy skracali jako SML), aczkolwiek przetłumaczenie ich na dowolny inny język funkcyjny nie powinno nastręczać większych trudności, jako że wszystkie języki funkcyjne opierają się na tej samej konstrukcji. Rezultaty wszystkich podanych tu przykładów zostały wygenerowane w interaktywnym trybie SML of New Jersey.

Jako że zajmujemy się tu przede wszystkim stroną teoretyczną programowania funkcyjnego, czytelnik bliżej zainteresowany poznaniem SML odsyłany jest niniejszym do innych źródeł.

1.1. **Paradygmat programowania funkcyjnego.**

Historycznie rzecz biorąc, w rozwoju języków programowania wyszczególnić można kilka paradygmatów. Każdy z nich skupia się na innym aspekcie programowania, wykazując swoje zalety w pewnych zastosowaniach oraz słabości w innych. Przykładowo, programowanie obiektowe wykorzystuje mechanizm rozszerzania klas obiektów do wyrażenia relacji pomiędzy różnymi obiektami. Programowanie funkcyjne, jak sama nazwa wskazuje, kładzie wyjątkowy nacisk na rolę funkcji jako podstawowych komponentów programów. Połączone z odpowiednim wykorzystaniem zasobów modułowych, okazuje się być doskonałym narzędziem w rozwoju oprogramowania wielkoskalowego.

Programowanie funkcyjne bywa często porównywane z programowaniem imperatywnym, gdy mowa o jego cechach charakterystycznych. W programowaniu imperatywnym, podstawowym komponentem są komendy. Typowy program składa się z szeregu komend, które po wywołaniu prowadzą do uzyskania określonego efektu. Tym samym program napisany w stylu imperatywnym można postrzegać jako instrukcję "jak coś policzyć" – a więc jak uporządkować tablicę danych, jak dodać element do danej listy, jak ustalić ścieżkę w drzewie itp. Dla kontrastu, programowanie funkcyjne w naturalny sposób zachęca programistów do skupienia się na tym, "co policzyć", jako że każdy fragment kodu musi mieć przypisaną pewną wartość.

Aby lepiej zrozumieć tę różnicę, rozważmy typową konstrukcję warunkową `if-then-else`. W typowym języku imperatywnym (na przykład w C, Javie czy Pascalu) następujący fragment kodu wygląda całkiem niewinnie i nie wzbudza żadnych podejrzeń; w istocie, w programowaniu imperatywnym tego typu kodowanie jest często nie do uniknięcia:

```
if (x == 1) then
  x = x+1;
```

Powyższy program wywołuje komendę zwiększającą wartość `x` gdy jest ona równa `1`; jeżeli wartość ta jest różna od `1`, komenda nie zostaje wywołana – nic złego się tu nie dzieje. Rozważmy teraz następujący fragment kodu napisany w pewnym hipotetycznym języku funkcyjnym:

```
if (x == 1) then x + 1
```

Wraz z brakującym określeniem ścieżki `else` powyższy kod nie ma sensu: każdy fragment kodu musi mieć przypisaną pewną wartość, ale powyższy kod nie ma przypisanej żadnej wartości, gdy `x` jest różne od `1`. Z tego powodu bezwzględnie konieczne jest określenie przypadku `else` w każdym języku funkcyjnym.

Tak jak w przypadku każdego paradygmatu programowania, prawdziwa siła programowania funkcyjnego może zostać właściwie doceniona dopiero po zdobyciu odpowiedniego doświadczenia. Niezależnie od tego, w odróżnieniu od innych paradygmatów programistycznych, programowanie funkcyjne jest zbudowane w oparciu o mnóstwo fascynujących koncepcji teoretycznych, całkowicie niezależnych od jego zalet czy wad w budowaniu oprogramowania. Tym samym programowanie funkcyjne, zdaniem autora, powinno być pierwszym krokiem w poznawaniu teorii języków programowania.

1.2. **Wyrażenia i wartości.** W SML-u programy składają się z **wyrażeń,** które zmieniają się od prostych stałych do złożonych funkcji. Każde wyrażenie może mieć przypisaną doń **wartość,** zaś proces redukowania wyrażenia do wartości nazywamy jest **wartościowaniem,** lub, bardziej uczenie, **ewaluacją.** Powiemy, że wyrażenie **wartościuje** się do danej wartości, gdy proces wartościowania się zatrzymuje. Zauważmy przy tym, że wartość jest specjalnym rodzajem wyrażenia.

1.2.1. *Przykład: liczby naturalne.* Stała naturalna `1` jest wyrażeniem, które samo jest też wartością. Wyrażenie naturalne `1+1` samo nie jest wartością, ale wartościuje się do naturalnej wartości `2`. Możemy spróbować znaleźć wartość związaną z wyrażeniem przez przypisanie mu typu i użycie średnika w SML-u:

```
- 1 + 1;
val it = 2 : int
```

Druga linia powyżej oznajmia, że wynikiem wartościowania danego wyrażenia jest wartość `2` (póki co nie przejmujemy się adnotacją typu `: int`, zajmiemy się tym za chwilę).

Wszystkie operacje arytmetyczne w SML-u mają nazwy znane z innych języków programowania (na przykład `+, -, *, div, mod`). Wyjątkiem jest unarny operator negacji, który zamiast przez `-` oznaczany jest przez `~`. Na przykład `~1` jest liczbą całkowitą ujemną, ale `-1` nie wartościuje się do liczby naturalnej.

1.2.2. *Przykład: wartości boole'owskie.* Stałymi boole'owskimi w SML-u są `true` i `false,` zaś konstrukcja warunkowa ma postać `if` e `then` e_1 `else` e_2, gdzie e, e_1 i e_2 są wyrażeniami oraz e musi wartościować się do wartości boole'owskiej. Na przykład:

```
- if 1 = ~1 then 10 else ~10;
val it = ~10 : int
```

Tutaj `1 = ~1` jest wyrażeniem, które porównuje dwa podwyrażenia `1` oraz `~1` dla sprawdznia ich równości (w istocie `=` również jest wyrażeniem – funkcją dwuargumentową). Jako że te dwa podwyrażenia nie są równe, `1 = ~1` wartościuje się do `false,` co z kolei powoduje, iż całe wyrażenie wartościuje się do `~10`.

W SML-u dostępne są też jako operatory logiczne `andalso, orelse` oraz `not`. Operatory binarne implementują wewnętrznie "spięcia," ale robienie "spięć" w językach funkcyjnych nie ma znaczenia, ponieważ wartościowanie wyrażenia nigdy nie powoduje efektów ubocznych.

Zadanie 1. *Czy potraficie uprościć* `if` e `then true else false`*?*

1.3. **Zmienne. Zmienne** są "pojemnikami" na wartości. Jako wyrażenia, wartościują się do wartości, którą zawierają. Używamy słowa kluczowego `val` chcąc inicjalizować zmienną. Na przykład zmienna `x` jest inicjalizowana wyrażeniem naturalnym `1+1` w następujący sposób:

```
- val x = 1 + 1;
val x = 2 : int
```

Zauważmy, że musimy wyposażyć wyrażenie, które będziemy chcieli użyć w obliczeniach, w początkową wartość dla `x`, albowiem w SML-u nie przypisujemy żadnych "defaultowych" wartości do zmiennych (w istocie przypisywanie takich "defaultowych" wartości zmiennym stałoby w sprzeczności z całą filozofią programowania fukcyjnego, o czym za chwilę). Po inicjalizowaniu zmiennej `x`, możemy użyć jej w innych wyrażeniach:

```
- val y = x + x;
val y = 4 : int
```

Powiemy, że dana zmienną jest **przypisana** do danej wartości, gdy została zainicjalizowana.

W odróżnieniu od zmiennych w językach imperatywnych, zmienna w SML-u jest "niemutowalna" (chciałby się napisać "niezmienna"...) w takim sensie, że jej zawartość nigdy się nie zmienia. Innymi

słowy, zmienna jest przypisana do danej wartości na dobre (to dlatego nie ma sensu deklarować zmiennej bez jej inicjalizowania, lub deklarować ją z "defaultową" wartością). Można powiedzieć, że w takim znaczeniu nazwa "zmienna" nie jest zbyt akuratna, jako że jej zawartość tak naprawdę się nie zmienia. Pomimo tej "niemutowalności" zmienne znajdują zastosowanie w językach funkcyjnych. Rozważmy przykład deklaracji lokalnej w SML-u, w której zero lub więcej zmiennych lokalnych zostaje zadeklarowanych przed wartościowaniem końcowego wyrażenia:

```
let
  val x = 1
  val y = x + x
in
  y + y
end
```

Tutaj deklarujemy dwie zmienne lokalne `x` oraz `y` przed wartościowaniem `y+y`. Jako że `y` jest dwukrotnie dodawane w końcowym wyrażeniu, możemy zaoszczędzić czas deklarując `y` jako zmienną lokalną zamiast rozwijać obydwa wystąpienia `y` w `x + x`. Użycie zmiennej lokalnej `y` poprawia też czytelność kodu.

Jakkolwiek może się to wydać zaskakujące (szczególnie dla kogoś przekonanego, że wykonywanie serii poleceń jest jedynym sposobem, aby coś policzyć...), "niemutowalność" zmiennych jest w istocie jedną z cech odróżniających programowanie funkcyjne od imperatywnego – bez tego typu ograniczenia, nie byłoby wielkiej różnicy między programowaniem funkcyjnym i imperatywnym, ponieważ komendy (na przykład służące do odświeżania zawartości zmiennej) są dostępne w obydwu paradygmatach programowania.

1.4. **Funkcje.** W kontekście języków funkcyjnych możemy myśleć o funkcji tak, jak myślimy o niej w matematyce, a zatem jak o "czarnej skrzynce", która jednoznacznie odwzorowuje dane na wejściu na dane na wyjściu. Tym samym deklarowanie funkcji w SML-u jest w zasadzie równoważne definiowaniu funkcji w matematyce, co z kolei implikuje, iż definiowanie funkcji matematycznej daje się łatwo przetłumaczyć na funkcję w SML-u. Kilka ciekawych przykładów pojawi się za chwilę, gdy zaczniemy mówić o rekursji – póki co skupmy się na koncepcji funkcji jako takiej i jej syntaksie w SML-u.

Do zadeklarowania funkcji używamy słowa kluczowego `fun`. Na przykład, możemy zadeklarować funkcję `incr` która zwraca daną liczbę naturalną zwiększoną o jeden:

```
- fun incr x = x + 1;
val incr = fn : int ->int
```

Tutaj `x` zwany będzie **formalnym argumentem,** albowiem służy tylko do zarezerwowania miejsca dla **faktycznego argumentu.** `x + 1` będziemy nazywać **zawartością funkcji.**

Możemy też stworzyć funkcję bez nazwy za pomocą słowa kluczowego `fn`. Poniższy fragment kodu tworzy taką samą funkcję jak `incr` i przechowuje ją w zmiennej `incr`:

```
- val incr = fn x =>x + 1;
val incr = fn : int ->int
```

Powyższe dwie deklaracje są oczywiście równoważne.

Aplikacja funkcji polega na podstawieniu faktycznego argumentu w miejsce formalnego argumentu w zawartości funkcji, a następnie na wartościowaniu otrzymanego wyrażenia. Na przykład, aplikacja funkcji `incr 0` (czyli zastosowanie funkcji `incr` do `0`) wartościuje do liczby naturalnej `1` w następujący sposób:

```
  incr 0
↦ (fn x =>x + 1) 0
```

```
↦ 0 + 1
↦ 1
```

Jako wyrażenie, funkcja jest już wartością. Intuicyjnie funkcja jest "czarną skrzynką", której wewnętrzna zasada działania jest ukryta i tym samym nie może być bardziej zredukowana. W rezultacie, zawartość funkcji jest wartościowana tylko wtedy, gdy pojawia się aplikacja funkcji. Na przykład funkcja bez nazwy `fn x =>0 + 1` nie wartościuje się do `fn x =>1`; tylko wtedy, gdy zostanie zaaplikowana to faktycznego argumentu (który w tym wypadku ignorujemy), jej zawartość zostanie wartościowana.

Ważną cechą programowania funkcyjnego jest to, że funkcje nie są traktowane inaczej niż elementarne wartości takie jak wartości boole'owskie czy naturalne. Na przykład funkcja może być przechowywana w zmiennej (jak w powyższym przykładzie), przekazana jako faktyczny argument do innej funkcji, czy nawet zwrócona jako wartość innej funkcji. Tego typu wartości często nazywane są **obiektami pierwszego rzędu** w żargonie programistycznym, albowiem tworzą najbardziej podstawowy element programu. Tym samym funkcje są obiektami pierwszego rzędu w programowaniu funkcyjnym. W istocie okazuje się, że program napisany w języku funkcyjnym może być traktowany jakby składał się wyłącznie z funkcji i niczego innego, jako że elementarne wartości mogą być również zakodowane jako funkcje. Wrócimy jeszcze do tego tematu.

Kilka interesujących przykładów traktowania funkcji jako obiektów pierwszego rzędu pojawi się w dalszym toku, a na razie ograniczymy się do przykładu ilustującego jak funkcja może być wartością zwracaną przez inną funkcję. Rozważmy następujący fragment kodu, gdzie deklarujemy funkcję `add`, która oblicza sumę dwóch liczb naturalnych:

```
- fun add x y = x + y;
val add = fn : int ->int ->int
```

Czy potrafilibyśmy, podobnie jak poprzednio, podać funkcję bez nazwy odpowiadającą `add`? Poniższa naiwna próba nie jest nawet zgodna z syntaksem SML-a:

```
- val add = fn x y =>x + y;
<tu wyskakuje jakaś error message >
```

Powodem, dla którego SML nie chce przełknąć powyższego kodu jest to, że każda funkcja w SML-u może mieć tylko jeden argument. Funkcja `add` podana powyżej zdaje się mieć dwa argumenty, ale w istocie jest to tylko zakamuflowana postać funkcji, która jako argument bierze liczbę naturalną, a jako wartość zwraca inną funkcję:

```
- val add = fn x y =>(fn y =>x + y);
val add = fn : int ->int ->int
```

Innymi słowy, gdy aplikujemy funkcję `add` do argumentu `x`, zwrócona zostaje nowa funkcja `fn y =>x + y,` która z kolei zwraca `x+y`, gdy jest aplikowana do argumentu `y`. Tym samym dozwolone jest aplikować `add` do pewnej liczby naturalnej, aby uzyskać nową funkcję tak jak tu:

```
- val incr = add 1;
val incr = fn : int ->int
- incr 0;
val it = 1 : int
- incr 1;
val it = 2 : int
```

Teraz powinno być jasne jak przebiega wartościowanie `add 1 + 1`:

```
  add 1 1
↦ (fn x =>(fn y =>x + y)) 1 1
↦ (fn y =>1 + y) 1
```

```
↦ 1 + 1
↦ 2
```

1.5. **Typy.** Pisanie dokumentacji jest integralną częścią dobrego programowania – bez odpowiedniej dokumentacji żaden kod nie jest łatwy do czytania, chyba że jest "self-explanatory". Ważność dokumentacji (na którą czasami kładziony jest przesadny nacisk w elementarnych kursach programowania) czasami prowadzi jednak do błędnego mniemania, iż nadmiernie rozbudowana dokumentacja jest lepsza od zwięzłej. Oczywiście nic nie może być bardziej odległe od prawdy – na przykład absurdalnie rozbudowana dokumentacja dotycząca funkcji `add` bardziej przeszkadza niż pomaga czytelnikowi kodu:

```
(* Funkcja bierze dwa argumenty i zwraca ich sumę.
* Obydwa argumenty muszą być liczbami naturalnymi.
* Jeśli nie są, rezultat jest nieprzewidywalny.
* Jeśli suma jest zbyt wielka, może nastąpić overflow.
* ...
*)
```

Problemem zawsze jest to, że cokolwiek zostaje napisane w dokumentacji, nie może być formalnie sprawdzone przez kompilator, a więc zmuszeni jesteśmy ufać autorowi dokumentacji. Nieuniknioną konsekwencją jest to, że każdy błąd w dokumentacji może tylko dodatkowo zdezorientować czytelnika, zamiast pomóc mu w zrozumieniu kodu. Wprawdzie w przypadku funkcji `add` potrafimy formalnie dowieść, że rezultatem istotnie jest suma dwóch argumentów, ale w jaki sposób mielibyśmy uwzględnić taki dowód w dokumentacji?

Z drugiej strony, krótka dokumentacja, która może zostać formalnie sprawdzona przez kompilator, często jest bezużyteczna. Na przykład załóżmy na chwilę, że rozszerzyliśmy syntaks SML-a tak, aby można było adnotować każdą funkcję liczbą jej argumentów:

```
argnum add 2            (* to nie jest prawidłowy syntaks SML-a, bawimy się tylko na niby..
fun add x y = x + y;
```

Tutaj `argnum add 2`, jako fragment kodu, orzeka iż funkcja `add` ma dwa argumenty. Kompilator oczywiście może sprawdzić, czy `add` faktycznie ma dwa argumenty, ale taka własność funkcji `add` nie wydaje się zbyt użyteczna.

Typy są dobrym kompromisem między wyrazistością i prostotą: zawierają użyteczną informację o kodzie (są wyraziste) i mogą być formalnie sprawdzone przez kompilator (ich działanie cechuje prostota). Nieformalnie typ jest zbiorem wartości tego samego rodzaju. Na przykład wyrażenie, które wartościuje się do liczby naturalnej lub stałej boole'owskiej ma typ `int` lub `bool`, odpowiednio. Funkcja `add` jest **typu funkcyjnego** `int ->(int ->int)`, ponieważ przy danej liczbie naturalnej typu `int`, zwraca funkcję typu `int ->int`, która z kolei bierze liczbę naturalną i zwraca liczbę naturalną. Chcąc wykorzystać typy jako część dokumentacji, możemy bezpośrednio adnotować każdy formalny argument jego typem; typ tego, co funkcja zwraca, jak również typ każdego podwyrażenia zawartości funkcji mogą być bezpośrednio wyspecyfikowane:

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

Kompilator SML-a sprawdza, czy podane typy są prawidłowe; jeśli nie, zgłasza błąd:

```
- fun add (x:int) (y:bool) : int = x+y
stdIn:2.23-2.28 Error: <tu wyskoczy jakaś error message >
```

Być może `add` jest zbyt prostym przypadkiem, aby w pełni pokazać zalety typów, ale w dalszym ciągu poznamy niezliczone przykłady, w których typ funkcji wyjaśnia o co chodzi.

Innym ważnym zastosowaniem typów jest pomoc w debuggowaniu. W programowaniu imperatywnym zakończona sukcesem kompilacja rzadko kiedy jest gwarancją poprawności kodu – na ogół kompilujemy program, uruchamiamy go i dopiero wtedy zaczynamy debuggowania badając rezultaty działania wykonywalnego kodu. W programowaniu funkcyjnym z odpowiednio bogatym systemem typów jest zupełnie inaczej: zaczynamy debuggowanie przed uruchomieniem kodu wykonywalnego poprzez badanie rezultatów kompilacji, które z reguły zawierają całe mnóstwo błędów typowych. Oczywiście w programowaniu funkcyjnym udana kompilacja również nie gwarantuje poprawności programu, ale programy, które jakoś się skompilowały, na ogół poprawnie działają.

1.6. **Rekursja.** Liczne zagadnienia w informatyce wymagają procedur iteratywnych służących osiągnięciu określonego celu – na przykład dodanie kolejnych liczb naturalnych od 1 do 100, posortowanie tablicy, przeszukiwanie drzew binarnych itp. Z racji swej prominentnej pozycji w programowaniu, obliczenia iteratywne są obsługiwane przez wbudowane konstruktory we wszystkich języka oprogramowania. Na przykład w języku C mamy do czynienia z konstruktorami bezpośrednio implementującymi obliczenia iteratywne takimi jak konstruktor pętli `loop`:

```
for (i = 1; i <= 10; i++)
  sum += 1;
```

Powyższy przykład używa jako indeksu zmiennej `i`, która zmienia wartości od `1` do `10`. Okazuje się, że nie możemy bezpośrednio przetłumaczyć takiego fragmentu kodu na czysty SML (to znaczy bez użycia mutowalnych referencji), ponieważ żadna zmienna w SML-u nie może zmieniać swej wartości. Czyżby miało to oznaczać, że SML jest gorszy od C pod względem swej siły wyrazu? Oczywiście nie: SML obsługuje **obliczenia rekursywne,** które mają taką samą siłę wyrazu jak obliczenia iteratywne.

Typowe obliczenie rekursywne polega na rozłożeniu danego problemu na mniejsze problemy, rozwiązaniu ich po kolei, a następnie połączeniu poszczególnych rozwiązań w odpowiedź na główny problem. Jest ważne, aby zdawać sobie sprawę, że w obliczeniach rekursywnych otrzymane mniejsze problemy dalej rozwiązywane są rekursywnie. Jako że otrzymany w ten sposób ciąg coraz to "mniejszych" problemów musi być skończony (w przeciwnym razie obliczenie nie mogłoby się zakończyć), obliczenie rekursywne zmienia kierunek w momencie dojścia do problemu, który nie jest już rozkładalny, na przykład w miejscu, gdy jest on natychmiastowo rozwiązywalny. Tym samym typowa postać rekursji składa się z **przypadków bazowych,** które specyfikują warunki, w których rekursja się zatrzymuje oraz **kroków indukcyjnych,** specyfikujących sposób, w jaki dany problem jest rozkładany na "mniejsze" problemy.

Jako przykład rozważmy funkcję rekursywną `sum`, która dodaje liczby naturalne od `1` do danego argumentu `n`; zauważmy, że nie możemy użyć słowa kluczowego `fn`, jako że musimy odwołać się do tej samej funkcji wewnątrz jej zawartości:

```
- fun sum n =
    if n = 1 then 1      (* przypadek bazowy *)
    else n + sum (n-1);  (* krok indukcyjny *)
val sum = fn : int ->int
```

Obliczenie `sum 10` będzie zatem wyglądało następująco:

```
  sum 10
↦ if 10 = 1 then 1 else 10 + sum (10 - 1)
↦ if false then 1 else 10 + sum (10 - 1)
↦ 10 + sum (10 - 1)
↦ 10 + sum 9
↦* 10 + 9 + ... 2 + sum 1
↦ 10 + ... 2 + (if 1 = 1 then 1 else 1 + sum (1 - 1))
```

```
↦ 10 + ... 2 + (if true then 1 else 1 + sum (1 - 1))
↦ 10 + ... 2 + 1
```

Tak jak w przypadku obliczeń iteratywnych, obliczenia rekursywne również mogą wpaść w nieskończoną pętlę w przypadku, gdy nigdy nie zostanie osiągnięty przypadek bazowy. Na przykład gdybyśmy chcieli wywołać funkcję `sum` do ujemnej liczby całkowitej, wpadlibyśmy w nieskończoną pętlę. W większości przypadków jednak nieskończone pętle są skutkami błędów programistycznych w zawartości funkcji, nie zaś wywołania funkcji z błędnym argumentem. Tym samym do dobrej praktyki należy zaprojektowanie funkcji rekursywnej przed napisaniem kodu. Dobrym sposobem projektowania takich funkcji jest zapisanie odpowiedniego równania. Na przykład równanie opisujące funkcję `sum` wyglądałoby następująco:

$$\begin{aligned} sum(1) &= 1 \\ sum(n) &= 1 + sum(n-1) \quad \text{jeśli } n > 1. \end{aligned}$$

Gdy zapiszemy już poprawnie takie równanie, jego przetłumaczenie na SML jest kwestią rutyny.

SML umożliwia również użycie funkcji wzajemnie rekursywnych. Słowo kluczowe `and` jest używane za każdym razem, gdy deklarujemy dwie lub więcej wzajemnie rekursywnych funkcji. Poniższy fragment kodu deklaruje dwie wzajemnie rekursywne funkcje `even` oraz `odd`, które rozstrzygają, czy dana liczba naturalne jest parzysta, czy nieparzysta:

```
fun even n =
  if n = 0 then true
  else odd (n - 1)
and odd n =
  if n = 0 then false
  else even (n - 1)
```

Rekursja na pierwszy rzut oka może się wydawać niezręcznym narzędziem nie najlepiej przystosowanym do obliczeń iteratywnych. Być może jest tak dlatego, że podejście iteratywne, które w istocie jest intuicyjnie prostsze do ogarnięcia, nasuwa się jako pierwsze skojarzenie – pamiętajmy wszakże, że jest to pierwsze skojarzenie po latach indoktrynacji przez bezmyślne programowanie imperatywne ;) Po przyzwyczajeniu się do programowania funkcyjnego Czytelnik szybko zauważy, że rekursja w istocie wcale nie jest niezręcznym narzędziem, ale być może najbardziej eleganckim środkiem w programowaniu, przy czym słowo "elegancki" rozumiemy tu jako synonim "łatwy w użyciu". Być może najważniejsza nauka, jaką Czytelnik wyniesie z tego wykładu, jest następująca: w programowaniu zawsze myśl rekursywnie!

1.7. **Typy polimorficzne.** W programowaniu często zdarza się, że wielokrotnie używamy tego samego fragmentu kodu tylko z niewielkimi modyfikacjami. Tym samym pożądane jest, aby móc napisać dany fragment i uzależnić go od pewnego parametru, który będziemy odpowiednio zmieniać za każdym razem, gdy będziemy potrzebowali zmodyfikowanej wersji fragmentu kodu, osiągając w ten sposób pewien stopień "reużywalności" naszego programu. Użyteczność tego typu rozwiązania jest oczywista. Na przykład, gdy odkryjemy błąd w kodzie, nie musimy sprawdzać wszystkich miejsc, w których zastosowaliśmy dany fragment.

Kwestia w jaki sposób uzyskać taką "reużywalność" jest jednak dość subtelna. Na przykład, w języku C uzyskujemy ją za pomocą makr, które wszelako są głównym źródłem niespodziewanych błędów. Templaty w C++ są bezpieczniejsze w użyciu, ponieważ ich parametrami są typy, ale mimo wszystko nie jest to nic więcej niż złożone makra. SML, dla odmiany, oferuje nam mechanizm, który nie tylko jest bezpieczny w użyciu, ale ma także solidne podstawy teoretyczne – mowa o **parametrycznym polimorfizmie.**

Jako prosty przykład rozważmy funkcję identycznościową:

```
val id = fn x = x;
```

Jako że nie specyfikujemy typu zmiennej `x`, może ona przyjmować argumenty o dowolnym typie. Semantycznie taka inwokacja funkcji `id` nie stwarza żadnego problemu, ponieważ jej zawartość nie potrzebuje wiedzieć, do jakiego typu `x` jest przypisana. Ta obserwacja sugeruje, że pojedyncza deklaracja `id` jest koncepcyjnie równoważna nieskończonej liczbie deklaracji, z których wszystkie mają taką samą zawartość funkcji:

```
  val_int id = fn (x:int) = x;
  val_bool id = fn (x:bool) = x;
  val_int→int id = fn (x:int ->int) = x;
...
```

Gdy `id` jest aplikowana do argumentu o typie A, kompilator SML-a automatycznie wybiera odpowiednią deklarację dla `id` typu A.

System typowania SML-a w zwarty sposób reprezentuje wszystkie deklaracje o tej samej strukturze przez jedną deklarację przy użyciu **zmiennej typowej:**

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

Tutaj zmienna typowa `'a` może być odczytana jako "dowolny typ α". Tym samym typ `id` oznajmia, że przy danym argumencie typu α zwrócona zostanie wartość o typie α. Możemy też bezpośrednio wyspecyfikować typy zmiennych, jakie mogą się pojawić w deklaracji wypisując je przed zmienną:

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

O typach bez zmiennych typowych będziemy mówili, że są **monotypami** (lub **typami monomorficznymi**), zaś o typach ze zmiennymi typowymi, że są **politypami** (lub **typami polimorficznymi**). System typowania SML-a pozwala zamieniać zmienną typową monotypem, ale nie politypem.