Metody programowania 2017

Lista zadań na pracownię nr 4

Na poprzedniej liście zadań pisaliśmy parser, czyli fragment implementacji języka programowania, który znajduje się na samym początku łańcucha przetwarzania programu. Celem tej listy zadań będzie napisanie kolejnych etapów, czyli sprawdzania typów oraz interpretacji. Przewidziane jest również zadanie dodatkowe polegające na kompilacji na maszynę stosową. Jednak zanim będziemy przetwarzać programy w bardziej złożonych językach (takich jak np. HDML), zaczniemy od czegoś prostego, czyli od wyrażeń arytmetyczno-logicznych. Na następnych listach zadań będziemy powoli urozmaicać rozważany język, tak by na koniec semestru mieć zaimplementowany prosty język programowania.

Wyrażenia arytmetyczno-logiczne

Składnia

Na poprzedniej liście zadań zobaczyliśmy formalną definicję składni języka. Ponieważ pisanie parsera nie jest celem tego zadania, tym razem poprzestaniemy na nieformalnym opisie (i referencyjnej implementacji zamieszczonej w serwisie SKOS). Poniżej opiszemy elementy składni rozważanego języka.

Zmienne zaczynają się literą lub podkreśleniem, po którym następuje ciąg znaków alfanumerycznych, podkreśleń i apostrofów. W treści zadania zmienne będziemy oznaczać metazmiennymi x, x_1, x_2, ..., tzn. każde wystąpienie x oznacza jakąś zmienną.

Liczby całkowite będziemy oznaczać metazmiennymi n, n_1, n_2, ...

Wyrażenia będziemy oznaczać metazmiennymi e, e_1, e_2, ... (tzn. każde wystąpienie e oznacza jakieś wyrażenie). Na wyrażenia składają się:

- *zmienne*,
- *liczby całkowite*,
- *stałe boolowskie* `true` i `false`,
- *wyrażenie z operatorem unarnym*: $\oplus e$,
- *wyrażenie z operatorem binarnym*: $e_1 \oplus e_2$,
- *wyrażenie let*: `let` x `=` e_1 `in` e_2,
- *wyrażenie warunkowe*: `if` e_1 `then` e_2 `else` e_3.

Operatory wraz z ich łącznością i priorytetem przedstawia poniższa tabelka:

operatory	łączność	priorytet
`-`	unarny	7
`* div mod`	w lewo	6
`+ -`	w lewo	5
`= <> < > <= >=`	brak	4
`not`	unarny	3
`and`	w lewo	2
`or`	w lewo	1

Programy są to wyrażenia w których pewne zmienne reprezentują zewnętrzne parametry. Np. program (wyrażenie) e z dwoma parametrami x_1, x_2 zapisujemy jako `input` x_1 x_2 `in` e. Jeśli program nie ma parametrów, to preambułę `input...in` pomijamy. Np. poprawnym programem jest wyrażenie `2+2`.

Parser zamieszczony w serwisie SKOS zamienia opisaną składnię konkretną na składnię abstrakcyjną, gdzie wszystkie elementy składni są wyrażone za pomocą algebraicznych typów danych. Szczegóły tej reprezentacji można znaleźć w pliku `AST.hs`. Poniżej przedstawiamy tylko definicję wyrażeń:

```
data Expr p
  = EVar    p Var                                -- Zmienna
  | ENum    p Integer                            -- Literał całkowitoliczbowy
  | EBool   p Bool                               -- Stała boolowska
  | EUnary  p UnaryOpeator (Expr p)              -- Wyrażenie operatorowe unarne
  | EBinary p BinaryOperator (Expr p) (Expr p) -- Wyrażenie operatorowe
  | ELet    p Var (Expr p) (Expr p)              -- Wyrażenie let
  | EIf     p (Expr p) (Expr p) (Expr p)         -- Wyrażenie warunkowe
```

Każdej opisanej powyżej konstrukcji języka odpowiada jeden konstruktor w typie `Expr p`. Np. wyrażenie `2+4` reprezentowane jest jako `EBinary` p `BAdd (ENum 2) (ENum 4)`, natomiast wyrażenie `let x =` e_1 `in` e_2 zapiszemy jako `ELet` p `"x"` e_1 e_2, gdzie p jest pewną wartością typu `p` będącego parametrem typu `Expr`. Tylko po co nam ten parametr `p`? Jak widzimy, każdy konstruktor wyrażenia zawiera element tego typu. Taki zabieg pozwala podpiąć pewną informację pod każdy element abstrakcyjnego drzewa rozbioru. Parser zostawia tam pozycje w kodzie źródłowym, ale można ten mechanizm wykorzystać w innym celu, np. typ `Expr [Var]` może reprezentować abstrakcyjne drzewo rozbioru, w którym w każdym wierzchołku trzymamy zmienne wolne. Mając element typu `Expr p` dodatkową informację (pozycję) umieszczoną w korzeniu możemy pobrać za pomocą funkcji `getData :: Expr p -> p`, zdefiniowanej w module `AST`.

System typów

W naszym języku istnieją programy, które są poprawne składniowo, ale nie mają większego sensu, np.

1. `2 + true`,
2. `input x in if 5 then false else x`,
3. `input x in x + y`.

W pierwszym programie dodajemy liczbę 2 do czegoś, co nie jest liczbą, w drugim programie warunek w instrukcji warunkowej nie jest wyrażeniem boolowskim, natomiast w trzecim programie mamy niezdefiniowaną zmienną `y`.

Aby uniknąć takich błędnych programów, wprowadzimy system typów, który oddzieli poprawne programy od tych niepoprawnych. Poprawne wyrażenia podzielimy na takie, które wyliczają się do liczby oraz takie, które wyliczają się do wartości boolowskiej. Zatem w tym języku mamy tylko dwa typy: `int` oraz `bool`. Typy będziemy oznaczać metazmiennymi τ, τ_1, τ_2, ...

Wyznaczając typ wyrażenia, potrzebujemy znać nie tylko samo wyrażenie, ale też typy wszystkich zmiennych, które które w nim występują. Informację o typach zmiennych trzymamy w tak zwanym środowisku, czyli funkcji częściowej ze zbioru zmiennych w zbiór typów. Środowiska będziemy oznaczać metazmiennymi Γ, Γ_1, Γ_2, ...

Zdefiniujmy relację, która przypisuje typy do wyrażeń. Będzie to 3-arna relacja wiążąca ze sobą środowisko, wyrażenie oraz typ. Stwierdzenie, że środowisko Γ, wyrażenie e oraz typ τ są w tej relacji będziemy zwyczajowo zapisywać $\Gamma \vdash e :: \tau$ oraz czytać: „przy środowisku Γ wyrażenie e ma typ τ”. Relację tę zdefinujemy jako najmniejszą relację zamkniętą na pewien zestaw reguł, który teraz omówimy.

Zmienne

Zmienne mają taki typ, jaki jest przypisany im w środowisku, co wyrażamy następującą regułą:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}$$

Gdybyśmy używali Prologa do zaimplementowania sprawdzania typów, to taką regułę wyrazilibyśmy następującą klauzulą:

```
infer_type(Gamma, var(_, X), T) :- env_lookup(Gamma, X, T).
```

Predykat `env_lookup` wyciąga typ `T` przypisany do zmiennej `X` w środowisku Γ, a jego implementacja zależy od przyjętej reprezentacji środowiska. Gdybyśmy w tym celu użyli list asocjacyjnych (biblioteka `assoc`), to jego definicja wyglądała by następująco:

```
env_lookup(Gamma, X, T) :- get_assoc(X, Gamma, T).
```

Należy zwrócić uwagę, że środowisko jest funkcją częściową, więc nie wszystkim zmiennym przypisujemy typy. Jeśli podczas sprawdzania typów napotkamy taki przypadek, powinniśmy zgłosić błąd o niezdefiniowanej zmiennej. W Prologu zapisalibyśmy to następująco:

```
infer_type(Gamma, var(Pos, X), T) :-
   env_lookup(Gamma, X, T), !;
   error(Pos, undefined_variable(X)).
```

Stałe

Liczby całkowite mają zawsze typ `int`, a stałe boolowski mają zawsze typ `bool`. Mamy zatem reguły

$$\frac{}{\Gamma \vdash n :: \texttt{int}} \qquad \frac{}{\Gamma \vdash \texttt{true} :: \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} :: \texttt{bool}}$$

które w Prologu wyglądałyby następująco

```
infer_type(_Gamma, num(_, _N), int).
infer_type(_Gamma, bool(_, _B), bool).
```

Operatory unarne

Mamy tylko dwa operatory unarne: unarny minus oraz negacja. Pierwszy z nich oczekuje wyrażenia typu `int` i tworzy wyrażenie typu `int`, natomiast drugi oczekuje wyrażenie typu `bool` i tworzy wyrażenie typu `bool`. Zapiszemy to następującymi regułami:

$$\frac{\Gamma \vdash e :: \texttt{int}}{\Gamma \vdash \texttt{-}e :: \texttt{int}} \qquad \frac{\Gamma \vdash e :: \texttt{bool}}{\Gamma \vdash \texttt{not } e :: \texttt{bool}}$$

W Prologu te reguły moglibyśmy zapisać następująco:

```
infer_type(Gamma, unary(_, neg, E), int) :-
  infer_type(Gamma, E, int).
infer_type(Gamma, unary(_, not, E), bool) :-
  infer_type(Gamma, E, bool).
```

Jednak przy takiej definicji próba wyprowadzenia typu wyrażenia `not 7` po prostu zawiedzie, bez podawania powodu. Problem ten możemy rozwiązać definiując dodatkowy predykat `check_type` działający w trybie $(+,+,+)$ (w przeciwieństwie do `infer_type` działającego w trybie $(+,+,-)$), który sprawdza czy wyrażenie ma podany typ, a jeśli tak nie jest to zgłasza błąd:

```
check_type(Gamma, E, ExpectedType) :-
  infer_type(Gamma, E, ActualType),
  ( ExpectedType = ActualType, !
  ; position(E, Pos),
    error(Pos, types_dont_match(ActualType, ExpectedType))
  ).
```

Przy pomocy tego predykatu możemy napisać lepszą wersję wyprowadzania typów dla operatorów unarnych, która zgłasza błąd w przypadku nieprawidłowego ich użycia:

```
infer_type(Gamma, unary(_, neg, E), int) :-
  check_type(Gamma, E, int).
infer_type(Gamma, unary(_, not, E), bool) :-
  check_type(Gamma, E, bool).
```

Operatory binarne

Operatory binarne możemy podzielić na trzy grupy:

- arytmetyczne (`+ - * div mod`), które operują na liczbach i zwracają liczbę,
- porównania (`= <> < > <= >=`), które operują na liczbach i zwracają wartość boolowską,
- boolowskie (`and or`), które operują na wartościach boolowskich i również zwracają wartość boolowską.

Jeśli przyjmiemy, że $\otimes$ oznacza operator arytmetyczny, $\lessgtr$ oznacza operator porównania, a $\diamond$ oznacza operator boolowski, to reguły typowania dla tych operatorów wyglądają następująco:

$$\frac{\Gamma \vdash e_1 :: \texttt{int} \qquad \Gamma \vdash e_2 :: \texttt{int}}{\Gamma \vdash e_1 \otimes e_2 :: \texttt{int}} \qquad \frac{\Gamma \vdash e_1 :: \texttt{int} \qquad \Gamma \vdash e_2 :: \texttt{int}}{\Gamma \vdash e_1 \lessgtr e_2 :: \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 :: \texttt{bool} \qquad \Gamma \vdash e_2 :: \texttt{bool}}{\Gamma \vdash e_1 \diamond e_2 :: \texttt{bool}}$$

A w Prologu:

```
infer_type(Gamma, binary(_, Op, E1, E2), int) :-
  is_arithmetic_op(Op),
  check_type(Gamma, E1, int),
  check_type(Gamma, E2, int).
infer_type(Gamma, binary(_, Op, E1, E2), bool) :-
  is_comparison_op(Op),
  check_type(Gamma, E1, int),
  check_type(Gamma, E2, int).
infer_type(Gamma, binary(_, Op, E1, E2), bool) :-
  is_boolean_op(Op),
  check_type(Gamma, E1, bool),
  check_type(Gamma, E2, bool).
```

Wyrażenie `let`

Wyrażenie `let` przypisuje zmiennej wartość jednego wyrażenia i pozwala jej użyć w drugim wyrażeniu. Typ tej nowej zmiennej jest taki, jak typ pierwszego wyrażenia. Reguła typowania wygląda więc następująco:

$$\frac{\Gamma \vdash e_1 :: \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 :: \tau_2}{\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 :: \tau_2}$$

Zapis $\Gamma[x \mapsto \tau_1]$ oznacza *rozszerzenie środowiska*, czyli nowe środowisko, w którym zmiennej x przypisujemy typ τ_1, a pozostałym zmiennym taki typ jaki był im przypisany w środowisku Γ.

Taką regułę również możemy zapisać w Prologu:

```
infer_type(Gamma, let(_, X, E1, E2), T2) :-
  infer_type(Gamma, E1, T1),
  env_extend(Gamma, X, T1, Gamma2),
  infer_type(Gamma2, E2, T2).
```

Predykat `env_extend` odpowiada za rozszerzanie środowiska. Gdybyśmy użyli list asocjacyjnych do reprezentacji środowisk, to jego implementacja wyglądałaby następująco:

```
env_extend(Gamma, X, T, RGamma) :- put_assoc(X, Gamma, T, RGamma).
```

Wyrażenie warunkowe

Pierwsze podwyrażenie wyrażenia warunkowego powinno być typu `bool`, natomiast pozostałe dwa powinny mieć ten sam typ, który jest też typem całego wyrażenia:

$$\frac{\Gamma \vdash e_1 :: \texttt{bool} \qquad \Gamma \vdash e_2 :: \tau \qquad \Gamma \vdash e_3 :: \tau}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 :: \tau}$$

co można zaimplementować w Prologu:

```
infer_type(Gamma, if(_, E1, E2, E3), T) :-
  check_type(Gamma, E1, bool),
  infer_type(Gamma, E2, T),
  check_type(Gamma, E3, T).
```

Uwagi dotyczące implementacji w Haskellu

Dla podanej implementacji w Prologu, predykat `infer_type` działa w trybie $(+,+,-)$, co oznacza, że wyznacza on typ znając środowisko oraz wyrażenie. Analogicznie można w Haskellu zdefiniować funkcję o sygnaturze:

```
infer_type :: Environment Type -> Expr p -> Type
```

gdzie `Environment a` jest typem reprezentującym funkcje częściowe ze typu `Var` w typ `a` (pomocny będzie moduł `Data.Map` z biblioteki standardowej). Jednak taki wybór nie jest najlepszy ponieważ nie mamy wtedy możliwości zgłoszenia błędu w przypadku niepoprawnych programów. Znacznie lepszym wyborem będzie:

```
infer_type :: Environment Type -> Expr p -> Either (Error p) Type
```

gdzie `Error p` jest typem reprezentującym błąd (trzeba go zdefiniować samemu), natomiast `Either` jest typem zdefiniowanym w bibliotece standardowej jako:

```
data Either a b = Left a | Right b
```

Całe programy

Parametry programów jak i ostateczna wartość powinna być typu `int`. Zdefiniujemy relację, która mówi o poprawności całego programu za pomocą reguły:

$$\frac{[x_1 \mapsto \texttt{int}, \ldots, x_n \mapsto \texttt{int}] \vdash e :: \texttt{int}}{\vdash \texttt{input } x_1 \ldots x_n \texttt{ in } e}$$

W prologu zapiszemy ją jako:

```
valid(program(Xs, E)) :-
  env_init(Xs, Gamma),
  check_type(Gamma, E, int).
```

gdzie `env_init` tworzy początkowe środowisko. Jeśli środowisko reprezentujemy za pomocą list asocjacyjnych, to może on wyglądać następująco:

```
env_init(Xs, Gamma) :-
  map_list(variable_init_typing, Xs, Pairs),
  list_to_assoc(Pairs, Gamma).

variable_init_typing(X, (X-int)).
```

Semantyka

Chociaż wydaje się, co poszczególne konstrukcje języka znaczą, to aby uniknąć niejednoznaczności, formalnie zdefiniujemy semantykę języka. W rozważanym języku mamy tylko dwa rodzaje wartości: liczby oraz wartości boolowskie (`tt` oraz `ff`). Do oznaczania wartości użyjemy metazmiennych v, v_1, v_2,..., do wartości boolowskich — b, b_1, b_2,..., natomiast liczby, podobnie jak literały całkowitoliczbowe będziemy oznaczać metazmiennymi n, n_1, n_2, ...

Aby obliczyć wartość wyrażenia np. `x+7`, potrzebujemy znać nie tylko samo wyrażenie, ale również wartości wszystkich zmiennych lokalnych (w przypadku wyrażenia `x+7` musimy znać wartość zmiennej `x`). Dlatego podobnie jak w przypadku systemu typów, elementem semantyki będzie środowisko, czyli funkcja częściowa ze zbioru zmiennych w zbiór wartości. Środowiska będziemy oznaczać metazmiennymi ρ, ρ_1, ...

Semantykę wyrażeń zadamy za pomocą semantyki naturalnej dużych kroków, czyli definiując relację wiążącą środowisko i wyrażenie z wartością. Stwierdzenie, że środowisko ρ, wyrażenie e oraz wartość v są w tej relacji zapiszemy $\rho \vdash e \Downarrow v$ i będziemy czytać: „przy środowisku ρ wyrażenie e oblicza się do wartości v". Podobnie jak dla systemu typów, relację tę zdefiniujemy podając zbiór reguł.

Zmienne i stałe

Zmienne obliczają się do takiej wartości, jaka jest im przypisana w środowisku, natomiast stałe od razu obliczają się do odpowiadającej im wartości:

$$\frac{\rho(x) = v}{\rho \vdash x \Downarrow v} \qquad \frac{}{\rho \vdash n \Downarrow n} \qquad \frac{}{\rho \vdash \texttt{true} \Downarrow \texttt{tt}} \qquad \frac{}{\rho \vdash \texttt{false} \Downarrow \texttt{ff}}$$

Operatory

Reguły dla wszystkich operatorów wyglądają podobnie, więc podamy tylko regułę dla dodawania:

$$\frac{\rho \vdash e_1 \Downarrow n_1 \qquad \rho \vdash e_2 \Downarrow n_2}{\rho \vdash e_1 \texttt{ + } e_2 \Downarrow n_1 + n_2}$$

Zauważmy, że oba znaki plusa w tej regule są inne: ten pierwszy (+) jest kawałkiem składni, operatorem który buduje większe wyrażenie z dwóch mniejszych. Natomiast ten drugi (+) jest matematycznym obiektem — funkcją obliczającą sumę dwóch liczb całkowitych.

Semantyka operatorów jest niezdefiniowana, jeśli któryś z argumentów jest niepoprawnego typu: jeśli któryś z argumentów operatora arytmetycznego lub porównania policzy się do wartości boolowskiej, lub któryś z argumentów operatora logicznego policzy się do liczby. Dodatkowo operacja dzielenia i operacja modulo jest niezdefiniowana, jeśli jej drugi argument jest równy zero. W takim wypadku należy zgłosić błąd.

Zauważmy, że policzenie operatora wymaga policzenia obu argumentów[1]. Na przykład wyrażenie `1 > 0 or 3 div 0 = 0` nie ma wartości (przy dowolnym środowisku).

Wyrażenie `let` oraz wyrażenie warunkowe

Semantyka wyrażenia `let` oraz wyrażenia warunkowego zadana jest następującymi regułami

$$\frac{\rho \vdash e_1 \Downarrow v_1 \qquad \rho[x \mapsto v_1] \vdash e_2 \Downarrow v_2}{\rho \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Downarrow v_2}$$

$$\frac{\rho \vdash e_1 \Downarrow \texttt{tt} \qquad \rho \vdash e_2 \Downarrow v}{\rho \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v} \qquad \frac{\rho \vdash e_1 \Downarrow \texttt{ff} \qquad \rho \vdash e_3 \Downarrow v}{\rho \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v}$$

[1] Oznacza to, że nasz język jest gorliwy. Implementując jego ewaluator w leniwym języku takim jak Haskell, trzeba uważać, by interpretowany język nie wyszedł przypadkiem też leniwy.

Zauważmy, że podczas obliczania wyrażenia warunkowego obliczana jest tylko jedna gałąź. Zatem obliczanie wyrażenia: `if true then 42 else 1 div 0` ma sens, pomimo, że w drugiej gałęzi występuje dzielenie przez zero.

Zadanie, część 1.
Termin zgłaszania w serwisie SKOS: 15 maja 2017 6:00 AM CEST

Napisz zestaw testów dla sprawdzania typów i interpretowania wyrażeń arytmetyczno-logicznych. Należy posłużyć się następującym szablonem (znajdującym się również w serwisie SKOS):

```
-- Wymagamy, by moduł zawierał tylko bezpieczne funkcje
{-# LANGUAGE Safe #-}
-- Definiujemy moduł zawierający testy.
-- Należy zmienić nazwę modułu na {Imie}{Nazwisko}Tests gdzie za {Imie}
-- i {Nazwisko} należy podstawić odpowiednio swoje imię i nazwisko
-- zaczynające się wielką literą oraz bez znaków diakrytycznych.
module ImieNazwiskoTests(tests) where

-- Importujemy moduł zawierający typy danych potrzebne w zadaniu
import DataTypes

-- Lista testów do zadania
-- Należy uzupełnić jej definicję swoimi testami
tests :: [Test]
tests =
  [ Test "inc"      (SrcString "input x in x + 1") (Eval [42] (Value 43))
  , Test "undefVar" (SrcString "x")                TypeError
  ]
```

Znaczenia poszczególnych pól pojedynczego testu można znaleźć w pliku `DataTypes.hs` zamieszczonym w serwisie SKOS.

Wymogi formalne
Należy zgłosić pojedynczy plik o nazwie *imię_nazwisko*`_tests.tar.bz2` gdzie za *imię* i *nazwisko* należy podstawić odpowiednio swoje imię i nazwisko bez wielkich liter i znaków diakrytycznych. Nadesłany plik powinien być poprawnym skompresowanym archiwum `tar.bz2` nie zawierającym żadnego katalogu. W archiwum powinny znajdować się **tylko**:

- Plik źródłowy napisany w Haskellu o nazwie o nazwie *ImięNazwisko*`Tests.hs`, gdzie za *Imię* i *Nazwisko* należy podstawić odpowiednio swoje imię i nazwisko zaczynające się wielką literą oraz bez znaków diakrytycznych. Plik ten powinien być napisany w Haskellu przy użyciu podzbioru *SafeHaskell*[2] i powinien definiować moduł eksportujący wartość `tests` typu `[Test]`.
- Wszystkie pliki źródłowe z programami w języku wyrażeń arytmetyczno-logicznych do których odwołują się testy (jeśli źródło programu podane jest za pomocą konstruktora `SrcFile`). Takie pliki powinny mieć rozszerzenie `.pp4`.

Rozwiązania nie spełniające wymogów formalnych nie będą oceniane!

Zadanie, część 2.
Termin zgłaszania w serwisie SKOS: 22 maja 2017 6:00 AM CEST

Napisz moduł eksportujący funkcje `typecheck` oraz `eval`, które odpowiednio sprawdzają typ oraz obliczają programy w języku wyrażeń arytmetyczno-logicznych. Należy posłużyć się następującym szablonem (znajdującym się również w serwisie SKOS):

[2] Wszystkie omawiane na wykładzie i ćwiczeniach elementy Haskella powinny mieścić się w tym podzbiorze.

```
-- Wymagamy, by moduł zawierał tylko bezpieczne funkcje
{-# LANGUAGE Safe #-}
-- Definiujemy moduł zawierający rozwiązanie.
-- Należy zmienić nazwę modułu na {Imie}{Nazwisko} gdzie za {Imie}
-- i {Nazwisko} należy podstawić odpowiednio swoje imię i nazwisko
-- zaczynające się wielką literą oraz bez znaków diakrytycznych.
module ImieNazwisko (typecheck, eval) where

-- Importujemy moduły z definicją języka oraz typami potrzebnymi w zadaniu
import AST
import DataTypes

-- Funkcja sprawdzająca typy
-- Dla wywołania typecheck vars e zakładamy, że zmienne występujące
-- w vars są już zdefiniowane i mają typ int, i oczekujemy by wyrażenia e
-- miało typ int
-- UWAGA: to nie jest jeszcze rozwiązanie; należy zmienić jej definicję.
typecheck :: [Var] -> Expr p -> TypeCheckResult p
typecheck = undefined

-- Funkcja obliczająca wyrażenia
-- Dla wywołania eval input e przyjmujemy, że dla każdej pary (x, v)
-- znajdującej się w input, wartość zmiennej x wynosi v.
-- Możemy założyć, że wyrażenie e jest dobrze typowane, tzn.
-- typecheck (map fst input) e = Ok
-- UWAGA: to nie jest jeszcze rozwiązanie; należy zmienić jej definicję.
eval :: [(Var,Integer)] -> Expr p -> EvalResult
eval = undefined
```

Wymogi formalne
Należy zgłosić pojedynczy plik o nazwie *ImięNazwisko*`.hs` gdzie za *Imię* i *Nazwisko* należy podstawić odpowiednio swoje imię i nazwisko zaczynające się wielką literą oraz bez znaków diakrytycznych. Plik ten powinien być napisany w Haskellu przy użyciu podzbioru *SafeHaskell* i powinien definiować moduł eksportujący funkcje `typecheck` oraz `eval` tak jak opisano w załączonym szablonie. **Rozwiązania nie spełniające wymogów formalnych nie będą oceniane!**

Uwaga
W serwisie SKOS umieszczono plik `Prac4.hs` pozwalający uruchamiać napisane rozwiązanie na przygotowanych testach. Sposób jego uruchamiania znajduje się w komentarzu wewnątrz pliku.