Pierwszy Parser, czyli Haskell nie dla geniuszy

Niniejsze materiały mają na celu przedstawienie paru rzeczy:

- testowania kodu w Haskellu (HSpec, QuickCheck)
- szeroko pojętych "dobrych praktyk" przy pisaniu kodu Haskella
- niektórych mniej lub bardziej zaawansowanych haskellowych pojęć
- radzenia sobie z kłodami, które pod nogi programisty rzuca ekosystem Haskella

Warto zaznaczyć, że choć materiały w dużej części omawiają tworzenie parsera przy użyciu biblioteki `megaparsec`, to jest on raczej pretekstem do przekazania powyższych informacji, niż celem samym w sobie (choć jest to bardzo dobra i nowoczesna biblioteka do pisania parserów).

1. Parsery -- wprowadzenie

Czym jest parser? Czymś, co przekształca tekst (w Haskellu może być to np. `String` albo `Text`) na jakiś ustrukturyzowany język formalny (najczęściej abstrakcyjne drzewo syntaktyczne jakiegoś języka). Przykładowo, możemy chcieć sparsować napis `"2 + 2 * 2"` do następującego obiektu: `Add (Lit 2) (Mul (Lit 2) (Lit 2))`. Dalej łatwo moglibyśmy napisać program, który wyliczy wartość takiego wyrażenia arytmetycznego.

```haskell
data Tree = Add Tree Tree
          | Mul Tree Tree
          | Lit Int

parse :: String -> Either String Tree
parse _ = undefined

eval :: Tree -> Int
eval (Lit n)     = n
eval (Add t1 t2) = eval t1 + eval t2
eval (Mul t1 t2) = eval t1 * eval t2
```

Ćwiczenie 1.1

Dopisz do powyższego programu implementację funkcji `parse`, używając w tym celu standardowych funkcji z modułu `base` i zwykłych operacji na napisach, typu `split`. Niepoprawne wejście powinno zwracać `Left <errorMessage>`. Jakie są zalety i wady takiego podejścia?

Ćwiczenie 1.2

Dopisz do powyższego programu również odejmowanie i dzielenie. Kto ustala priorytety operacji? Funkcja `parse`, czy funkcja `eval`?

Oczywiście, parserów możemy używać do dużo bardziej skomplikowanych zadań: parsowania języków programowania, plików w formatach takich, jak JSON czy YAML czy niestandardowych logów. My stworzymy swój własny język, oparty na Lispie, który będziemy sukcesywnie rozwijać.

Megaparsec

Pisanie parserów od zera jest wykonalne, ale zbędne: istnieją do tego wspaniałe narzędzia. My wybierzemy `megaparsec`, gdyż wydaje się obecnie być "state of the art" bibliotek do tworzenia parserów. Cały pomysł opiera się na pisaniu prostych parserów, "rozumiejących" najprostsze konstrukcje (pojedyncze znaki czy słowa) i składaniu ich w bardziej zaawansowane parsery.

Projekt w Haskellu

Na dobry początek zaczniemy tworzyć nasz projekt.

```
$ stack new parser-example
```

W ten sposób dostaniemy nowy projekt `stack`, który będziemy mogli rozwijać. Żeby użyć pakietu `megaparsec`, musimy go dodać jako zależność projektu w `package.yaml`. W polu `dependencies` dodajemy `- megaparsec`. Teraz możemy wywołać `stack install` i pakiet powinien się ściągnąć i zainstalować. Stwórzmy również plik `src/Parsers.hs`, w którym będziemy tworzyć nasze parsery. Zaczniemy od deklaracji modułu i importów:

```
module Parsers where

import           Text.Megaparsec      (Parsec)
import qualified Text.Megaparsec.Char as P
```

Pierwszy parser

Nim przystąpimy do pisania faktycznych parserów, ułatwimy sobie życie: podstawowym typem w `megaparsecu` jest `Parsec`, który ma trzy parametry:

- typ błędów, zwracanych przez parser
- typ wejścia parsera
- typ zwracany przez parser

My, na potrzeby tego wykładu, będziemy pisać parser, który jako błędy zwraca napisy typu `String` (choć w produkcyjnym kodzie napisalibyśmy własne typy na błędy) i typu `String` jest jego wejście (choć w produkcyjnym kodzie lepiej używać typu `Text` z pakietu `text`). W takim razie możemy zdefiniować sobie dokładnie to:

```
type Parser = Parsec String String
```

Zwróćmy uwagę, że trzeci parametr pozostawiamy "wolny", więc będziemy mogli stworzyć np. coś typu `Parser Foo`.

Zaczniemy od najprostszego możliwego przykładu: parsowania jednego znaku. Jako, że będziemy pisać język lispopodobny, będzie to znak `'('`. Do naszego pliku `src/Parsers.hs` dopiszmy funkcję:

```
openingParen :: Parser Char
openingParen = P.char '('
```

Przetestujmy ją w GHCi (`stack ghci`):

```
> import Text.Megaparsec (parse)
> parse openingParen "" "(hello)"
Right '('
> parse openingParen "" "hello)"
Left **BOOM**
```

Zignorujmy drugi parametr funkcji `parse`, który nie będzie nam potrzebny, i który możemy z powodzeniem ustawić jako pusty napis. Pierwszy parametr to parser, jakiego funkcja ma użyć, a trzeci to wejście, które chcemy sparsować. Widzimy zatem, że nasz prosty parser, zgodnie z tym, jak go napisaliśmy, czyta jeden znak z wejścia i zależnie od tego, czy jest tym, czego się spodziewa, zwraca albo `Right '('` (sukces) albo `Left <straszny błąd>`. `<straszny błąd>` będziemy docelowo chcieli przerobić na coś zrozumiałego dla ludzi, więc coś pokroju `"Couldn't parse the input: expected an opening paren"`. Całkiem rozsądnym pytaniem będzie: co dzieje się z resztą wejścia? O tym w późniejszych rozdziałach.

Trochę zamieszania dla ułatwienia

(uwaga: to, co tutaj pokażemy, jest niby-proste, a jednak warto wiedzieć, jak sobie radzić ze skomplikowanymi sygnaturami)

Jak widzimy, funkcja `parse` ma parametr, którego nie będziemy używać. Dodatkowo, jej typ jest nieco zbyt skomplikowany, jak dla naszych potrzeb. Zróbmy więc prostą rzecz: napiszmy własną funkcję `parse`, która będzie miała dobry typ i jeden argument mniej.

Musimy zacząć od importu:

```
import qualified Text.Megaparsec as P
```

Importujemy w ten sposób, bo gdy nazwiemy naszą funkcję `parse`, nie będziemy mieć konfliktów nazw.

Następnie zdefiniujmy sobie naszą funkcję:

```
parse parser input = P.parse parser "" input
```

(uwaga: parametr input możnaby po obu stronach usunąć, przerabiając funkcję na wariant bezpunktowy, ale w zasadzie w ten sposób jest czytelniej).

Jaki ma typ? W GHCi:

```
> :t parse
parse :: Text.Megaparsec.Parsec e s a -> s ->
Either(Text.Megaparsec.Error.ParseErrorBundle s e) a
```

Hmm, zbyt skomplikowane. Ale przecież mamy `type Parser = Parsec String String`! Czyli możemy zacząć pisać sygnaturę:

```
parse :: Parser a -> String -> Either _ a
parse parser input = P.parse parser "" input
```

Nie jesteśmy jeszcze pewni, co będzie tym dziwnym pierwszym parametrem `Either`. Nie szkodzi, kompilator nam pomoże. Jeśli zastąpimy go `_` i spróbujemy skompilować kod, dostaniemy:

```
* Found type wildcard `_'
    standing for `P.ParseErrorBundle String String'
```

Świetnie! Zatem możemy sobie dla ułatwienia zdefiniować:

```
type ParserError = P.ParseErrorBundle String String
```

i uzupełnić naszą funkcję:

```
parse :: Parser a -> String -> Either ParserError a
parse parser input = P.parse parser "" input
```

Całkiem ładna sygnatura! Pozbywamy się polimorfizmu błędów i strumienia wejściowego, ale zyskujemy dużo większą czytelność kodu. Dla mnie -- ekstra.

Testy pierwszego parsera

Jak każdy szanujący się programista, napiszemy do naszego programu testy jednostkowe. Zaczniemy od dodania do zależności w `package.yaml` pakietu `hspec` (standardowy framework do testów jednostkowych w Haskellu). Wystarczy, że dodamy go w polu `dependencies` w sekcji `test` -- reszta aplikacji nie potrzebuje modułu do testów, więc często nie będzie potrzeby ich budować.

Skoro mamy już taki pakiet, możemy zacząć pisanie testów. W pliku `test/Spec.hs` zaimportujemy sobie dwie rzeczy:

1. narzędzia do testów:

```
import Test.Hspec
```

2. moduł, który będziemy testować:

```
import Parsers
```

(uwaga: ogólnie rzecz biorąc chcemy używać wyłącznie importów kwalifikowanych (`import qualified M as N`) albo importować konkretne symbole `import M (a, b)`), ale w testach do naszego prostego przykładu możemy na chwilę sobie pozwolić na odrobinę niedbałości).

Testy piszemy w formie: `<ten obiekt> powinien robić <to>`. Hspec ma do tego ładne funkcje, zapiszmy więc w jego języku "parser `openingParen` powinien parsować napis `"("` i zwracać `Right '('`". W zasadzie sprowadza się to do przetłumaczenia tego na angielski:

```
describe "The open paren parser" $
    it "should parse the \"(\" string" $
        parse openingParen "(" `shouldBe` Right '('
```

Całkiem czytelnie. Oczywiście "The open [...]" oraz "should parse [...]" to tylko komentarze dla programisty, pomocne przy wyświetlaniu wyników testu -- tylko ostatnia linijka mówi Hspecowi, co faktycznie ma przetestować. Tak czy owak, bardzo intuicyjne podejście, bardzo podobne do RSpeca (zresztą: zbieżność nazw jest nieprzypadkowa).

Musimy to jeszcze tylko ubrać w funkcję i wywołać:

```
openingParenSpec :: Spec
openingParenSpec = describe "The open paren parser" $ do
    it "should parse the \"(\" string" $
        parse openingParen "(" `shouldBe` Right '('

main :: IO ()
main = hspec $ openingParenSpec
```

Co robi funkcja `hspec`? Popatrzmy na jej typ: `hspec :: Spec -> IO ()`. Po prostu wykonuje testy.

Możemy teraz w konsoli uruchomić nasz test:

```
$ stack test
```

(Wyjście pomijamy, ale powinniśmy zobaczyć zielony napis, informujący o powodzeniu testu).

Możemy dopisywać następne testy do tego samego parsera dodając kolejne bloki `it "[...]"` w środku funkcji `describe` (kod poniżej). Co jednak ze sprawdzeniem, czy parser zgłąsza błąd wtedy, kiedy trzeba? Błędy są na razie dość zawiłe, więc nie będziemy sprawdzać, czy parser zwrócił konkretny błąd -- wystarczy nam upewnić się, że dla złego wejścia zwrócił `Left`. Do tego celu przyda nam się funkcja `isLeft :: Either l r -> Bool`, która co prawda znajduje się w pakiecie `extra`, ale czytelnik dopisze ją sam, jako ćwiczenie.

Ćwiczenie 1.3

Napisz funkcję `isLeft` opisaną wyżej. `isLeft $ Left undefined` powinno zwrócić `True`, `isLeft $ Right $ error "an error"` powinno zwrócić `False`. Uwaga: przetestuj funkcję również na tych konkretnie przykładach.

Mamy zatem:

```
openingParenSpec :: Spec
openingParenSpec = describe "The open paren parser" $ do
    it "should parse the \"(\" string" $
        parse openingParen "(" `shouldBe` Right '('

    it "should fail on the \"hello\" string" $
        isLeft (parse openingParen "hello") `shouldBe` True
```

To już coś! (Zwróćmy uwagę na dodane `do`, które umożliwia nam użycie wielu `it`).

Ćwiczenie 1.4

Dopisz jeszcze po jednym przykładzie testowym na poprawne i niepoprawne parsowanie.

Powtarzalność

Czyżbyśmy robili przysłowiową "robotę głupiego"? Wydaje się, że przykłady testowe, które piszemy dla różnych wejść, są wszystkie strasznie podobne. Czy nie ma jakiejś biblioteki, która zrobiłaby to za nas? Poszukała wielu przypadków (w tym brzegowych) tak, żeby zweryfikować nie tyle konkretne przypadki, co pewne właściwości?

Tak! Istnieje, i nazywa się QuickCheck!

Dodamy do naszego `package.yaml` dodamy zależność `quickcheck` (najlepiej w sekcji `test` -- tak, aby wszystkie frameworki do testów nie musiały się dołączać do "produkcyjnego" kodu, zwiększając czas budowania) i do pliku `Spec.hs` linijkę:

```
import Test.QuickCheck
```

Choć QuickCheck w żaden sposób nie zależy od HSpeca, dobrze się z nim integruje, dostarczając ładnej składni, którą już znamy. Spróbujmy uogólnić testy, które pisaliśmy poprzednio. Chcielibyśmy sprawdzić, czy "parser `openingParen` zwraca `Right '('` dla każdego napisu zaczynającego się od otwierającego nawiasu. Popatrzmy:

```
openingParenPropSpec :: Spec
openingParenPropSpec = describe "A single character parser" $ do
    it "should accept this character" $ property $
        \s -> parse openingParen ('(':s) `shouldBe` Right '('
```

Wciąż wygląda to prawie jak język angielski. Różnica jest taka, że teraz, zamiast podawać konkretny przypadek, który musi być prawdziwy, podajemy pewną właściwość. Za pomocą funkcji `property` mówimy: "dla dowolnego (napisu) `s` musi zachodzić [...]". QuickCheck automagicznie potrafi sprawdzić wiele różnych wartości `s`, dobrze sobie radząc z szukaniem przypadków brzegowych.

Ćwiczenie 1.5

Dopisz do `openingParenPropSpec` test na napisy, które powodują, że parser zwraca `Left`. Jak upewnić się, że napis będzie dowolny, **ale** nie będzie się zaczynał od `(`?

Ćwiczenie 1.6

Napisz test, który zachodzi dla **prawie** wszystkich przypadków i sprawdź, czy QuickCheck go znajdzie.

Pytanie

Jak to się dzieje, że możemy dodać `property` i lambdę zamiast po prostu `x shouldBe y`? Sprawdź typy funkcji `it` i `property`. Pomocne mogą się okazać, w GHCi, komendy `:t` i `:i`.

2. Rozwijanie parsera

Potrafimy parsować jeden znak -- przydałoby się jednak umieć coś więcej. Parsery możemy składać, tak, by z prostszych kawałków złożyć coś, co potrafi parsować bardziej skomplikowane rzeczy. Zacznijmy dość prosto: parser, który akceptuje nawias otwierający **lub** zamykający:

```
paren :: Parser Char
paren = openingParen <|> closingParen

-- dodaliśmy oczywiście jeszcze tą funkcję:
closingParen :: Paren Char
closingParen = P.char ')'
```

Możemy przetestować, czy wszystko działa:

```
> parse paren "(hello)"
Right '('
> parse paren ")hello"
Right ')'
> parse paren "hello"
Left <okropny błąd>
```

Wydaje się działać. Oczywiście, nie poprzestaniemy na wydawaniu się: dopiszemy testy.

Ćwiczenie 2.1

Napisz w pliku `test/Spec.hs` dopisz:

1. Testy jednostkowe, sprawdzające, czy parser `paren` działa dla otwierających i zamykających nawiasów.
2. Testy jednostkowe, sprawdzające, czy parser `paren` poprawnie zwraca rezultat `Left <...>` dla niepoprawnych napisów.
3. Testy QuickChecka, sprawdzające, czy dowolny napis zaczynający się od `(` lub `)` jest akceptowany przez parser.

Prawdziwe oblicze alternatywy

Zastanówmy się przez chwilę, czym jest magiczne `<|>`, którego używamy? Czy to jakaś magiczny operator Megaparseca? Nie do końca:

```
> :t (<|>)
(<|>) :: GHC.Base.Alternative f => f a -> f a -> f a
```

`Alternative` to typeklasa, która jest abstrakcją stworzoną do właśnie takich celów.

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

Zobaczmy, jak działa to dla `Maybe`:

```
> Nothing | Nothing
Nothing
> Just 42 | Nothing
Just 42
> Nothing | Just 42
Just 42
> Just 42 | Just 43
Just 42
```

Widać zachowanie bardzo podobne, jak w przypadku naszych parserów. Swoją drogą, dla list `<|>` to po prostu konkatenacja.

Rozwijanie, ciąg dalszy

Na razie jesteśmy w stanie parsować pojedyncze znaki, co jest imponujące, ale nie jesteśmy w stanie wiele zrobić. Jak zatem połączyć parsery w dłuższe sekwencje?

Spróbujmy na początek trywialnego przykładu: sparsujemy otwierający nawias, a następnie nawias zamykający. Nie jest to samo w sobie zbyt użyteczne, ale nauczymy się przydatnych rzeczy. Przede wszystkim: Megaparsec modeluje rzeczy następujące po sobie w sekwencji tak, jak większość Haskellowego światka: za pomocą monad. Instancje dla parserów napisane są tak, by po kolei parsowały i "zjadały" kawałki wejściowego strumienia. Jeśli któryś z parserów nie zaakceptuje wejścia, cała sekwencja zwróci błąd.

Najpierw przyglądniemy się, jak to działa w praktyce, a dopiero później zaglądniemy "pod maskę", by poznać nieco lepiej ideę działania parserów i ich składania.

```
emptyParens :: Parser Char
emptyParens = openingParen >> closingParen
```

Możemy sprawdzić w GHCi, czy działa tak, jakbyśmy się tego spodziewali:

```
> parse emptyParens "()"
Right ')'
```

Zwróćmy uwagę, że każdy nasz parser ma typ `Parser Char`, więc zawsze będzie zwracał tylko jeden znak. Docelowo chcemy, żeby parser zwrócił coś bardziej ustrukturyzowanego (AST) -- zajmiemy się tym w dalszych odcinkach.

Ćwiczenie 2.2

Napisz testy (HSpec + QuickCheck) testujące parser `emptyParens`.

3. Parser monadyczny pod maską

Powiedzieliśmy, że przyglądniemy się, jak parser działa "pod maską". Zamiast oglądać implementację Megaparseca (która jest mocno zoptymalizowana -- w większości języków obniża to czytelność kodu, a co dopiero w Haskellu), spróbujmy napisać własny, prosty parser. (Pożyczymy implementację z tej odpowiedzi na Stack Overflow, ale nasz parser będzie miał jeszcze instancję monady).

Zacznijmy od definicji typu `Parser`:

```
type Error = String
newtype Parser a = P { unP :: String -> (String, Either Error a) }
```

Parser to po prostu `newtype` nad funkcją, która przyjmuje napis wejściowy i zwraca parę: pierwszy element to pozostały fragment napisu wejściowego. Drugi to rezultat parsowania, który może być albo `Right <to, co zwraca parser>`, albo `Left <błąd parsowania>`. Łatwo zauważyć, że taka definicja sama w sobie nie pozwoli nam na zbyt wielką swobodę składania parserów (w końcu: moglibyśmy napisać parser jako jedną funkcję z wielką plątaniną `if`-ów, ale po to programujemy funkcyjnie, żeby rzeczy dobrze się "komponowały"). Powyżej widzieliśmy, że instancje `Alternative` i `Monad` dają naszym parserom duże możliwości -- napiszmy je

więc. (W dzisiejszych czasach każda monada musi być również aplikatywem, więc musimy napisać instancje: `Functor`, `Applicative` i `Monad`).

Najpierw instancja `Functor`:

```
instance Functor Parser where
  fmap f (P st) = P $ \stream -> case st stream of
    (rest, Left err) -> (rest, Left err)
    (rest, Right a ) -> (rest, Right (f a))
```

Dość proste: bierzemy nasz stary parser, wyciągamy z niego funkcję parsującą, a następnie tworzymy nową funkcję: taką, która zaaplikuje starą funkcję na swoim parametrze. Jeśli rezultat to `Right`, to zaaplikujemy `f` w środku. Jeśli rezultatem był błąd `Left`, zostawimy go jak jest. Warto zwrócić uwagę, że `st stream` nie wykona się od razu -- dopiero, jak ktoś "odpakuje" obiekt typu `Parser`. To jeden z uroków leniwej ewaluacji: w większości przypadków kopie nas w kostkę i spowalnia programy, ale czasami pozwala na eleganckie abstrakcje.

Następnie instancja `Applicative`:

```
instance Applicative Parser where
  pure a = P (\stream -> (stream, Right a))
  P f1 <*> P xx = P $ \stream0 -> case f1 stream0 of
    (stream1, Left err) -> (stream1, Left err)
    (stream1, Right f ) -> case xx stream1 of
      (stream2, Left err) -> (stream2, Left err)
      (stream2, Right x ) -> (stream2, Right (f x))
```

Funkcja `pure` jest prosta: zwraca swój argument "opakowany" w całą maszynerię parsera. Funkcja `<*>` (na którą ponoć mówi się "ap") robi podobną rzecz, którą robiło `fmap`, tylko dwa razy.

Ćwiczenie 3.3

Mając instancję `Applicative`, nie jest tak trudno wymyślić instancję monady (swoją drogą -- istnieją też parsery aplikatywne, ale my chcemy naśladować Megaparseca). Napisz instancję `Monad` dla naszego parsera (tak naprawdę wystarczy napisać metodę `>>=`).

Wskazówka

Sygnatury są następujące:

```
instance Monad Parser where
    return :: a -> Parser a
    return = pure

    (>>=) :: Parser a -> (a -> Parser b) -> Parser b
    p >>= f = ???
```

Parsowanie pojedynczych znaków

Mamy monady, więc moglibyśmy już parsować sekwencje znaków. Ale nie umiemy parsować pojedynczych znaków! Faktycznie -- nieco się zapędziliśmy. Nasz parser wymaga jeszcze nieco "pracy u podstaw". Zdefiniujmy sobie funkcję, która pozwoli podglądnąć znak i zaakceptować go, jeśli spełnia podany w argumencie predykat. Oto ona:

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy f = P $ \stream -> case stream of
    []                  -> ([], Left "end of stream")
    (c:cs) | f c        -> (cs, Right c)
           | otherwise -> (cs, Left "did not satisfy")
```

Ćwiczenie 3.4

Używając funkcji `satisfy`, napisz funkcję `char`, parsującą konkretny znak. Jej sygnatura to: `char :: Char -> Parser Char`.

Alternatywy (4?)

Megaparsec potrafił parsować `to` **lub** `tamto`. Wypada nam dopisać to do naszego zabawkowego parsera, żeby zrozumieć, jak to może działać. Interesuje nas funkcja, która dostanie dwa parsery i wykona następujące kroki:

1. Uruchomi pierwszy parser na danym wejściu. Jeśli je zaakceptował, powinna zwrócić dobry rezultat.
2. Jeśli pierwszy parser odrzucił wejście, zignoruje jego rezultat i uruchomi drugi parser. Wtedy już nie ma wyboru: musi po prostu zwrócić jego rezultat. Oczywiście funkcja nie tyle te kroki wykona, co zwróci funkcję, która je wykona, gdy zostanie uruchomiona. A to wszystko jeszcze opakowane w parserowy newtype!

Kod może wyglądać tak:

```
orElse :: Parser a -> Parser a -> Parser a
orElse (P f1) (P f2) = P $ \stream0 -> case f1 stream0 of
    (stream1, Left _)  -> f2 stream1
    (stream1, Right a) -> (stream1, Right a)
```

To mały krok dla ludzkości, ale wielki dla naszego małego parsera: teraz możemy zdefiniować dla niego instancję `Alternative`. No więc proszę:

```
instance Alternative Parser where
    empty = P $ \stream -> (stream, Left "empty")
    (<|>) = orElse
```

Nasza biblioteka robi się już całkiem użyteczna. Gdybyśmy jednak nie czuli potrzeby nieustannego doskonalenia naszego kodu, zapewne pisalibyśmy w języku innym, niż Haskell. Warto zatem nadmienić, że klasa `Alternative` ma jeszcze metody `some` oraz `many`. Mają defaultowe implementacje, ale możemy je przesłonić wydajniejszymi (podobnie jest na przykład z klasą `Monad`, która ma defaultową implementację `>>`, ale można napisać własną, jeśli z jakichś powodów mamy na nią dobry pomysł). Zróbmy to jako ćwiczenie:

Ćwiczenie 3.5

Napisz metody:

```
many :: Parser a -> Parser [a]
some :: Parser a -> Parser [a]
```

które dostaną parser jako argument i będą akceptować odpowiednio: 0 lub więcej oraz 1 i więcej powtórzeń wejścia, które akceptuje ich parser argument. Przykładowo: `some $ char 'a'` zaakceptuje napisy `"ab"` i `"aaaabb"`, a odrzuci `"bb"`. `many $ char 'a'` zaakceptuje wszystkie z nich.

Jazda próbna parsera

Do pełni szczęścia brakuje nam tylko drobnej funkcji owijającej wywołania parsera (dla wygody):

```
parse :: Parser a -> String -> Either Error a
parse parser input = snd $ (unP parser) input
```

Teraz możemy uruchomić `stack ghci` i sprawdzić, czy wszystko parsuje się zgodnie z naszymi oczekiwaniami:

```
> let emptyParens = char '(' >> char ')'
> parse emptyParens "()"
Right ')'
> parse emptyParens ")"
Left "did not satisfy"
> parse emptyParens "("
Left "end of stream"
```

Podobnie możemy sprawdzić metody `some`, `many` oraz `<|>`.

Ćwiczenie 3.6

Dopisz w HSpecu i QuickChecku testy naszego zabawkowego parsera. Możemy go dla wygody umieścić w pliku `src/MockParser.hs`. Należy jednak pamiętać, by w testach importować go w sposób kwalifikowany (np. `import qualified MockParser as MP`) -- w przeciwnym razie nazwy funkcji będą konfliktowały z Megaparsekiem.

Uwagi końcowe i przemyślenia

Rozumiemy już, jaki pomysł stoi za takim podejściem do parsowania i widzimy, jak sprytnie napisane instancje `Monad` oraz `Alternative` pozwalają nam na składanie prostych parserów w bardziej skomplikowane. Ale chwileczkę... Czy na pewno jest konieczne, żeby nasz parser był monadą? Czyżbyśmy wykonali pracę na darmo? Przecież istnieją aplikatywne parsery! W tym celu zróbmy dwie rzeczy: jedno ćwiczenie (ćwiczonko) i jedno przemyślenie.

Ćwiczenie 3.7

Jedyną rzeczą z monady, z jakiej korzystaliśmy, jest operator `>>`. Pozwala nam na parsowanie jednej rzeczy po drugiej, w sekwencji. Da się ten sam efekt osiągnąć tylko przy pomocy instancji `Applicative` naszego parsera. Napisz funkcję `andThen :: Parser a -> Parser b -> Parser b`, która wykona dwa parsery w sekwencji tak, jak widzieliśmy powyżej.

Wskazówka

Pomocna może się okazać funkcja `seq`, która skądinąd jest dość ciekawa.

Przemyślenie

Co "potrafi" parser monadyczny, czego nie potrafiłby parser aplikatywny?

Wkazówka

Patrząc na ćwiczenie powyżej: czym się różni (w kontekście naszego parsera) `>>` od `>>=`? Co ignoruje `>>`?

Być może ciekawsza dyskusja to ewentualna przewaga parsera aplikatywnego nad monadycznym: wykracza to nieco poza zakres tego kursu, więc zainteresowanych pozostaje mi odesłać do internetu, który jest pełen opracowań na ten temat. Nam wystarczy wiedza, że Megaparsec jest parserem monadycznym, więc z takim będziemy mieć przez resztę kursu do czynienia -- dla naszych potrzeb będzie to na pewno dobra decyzja.

4. Prawdziwy parser

To, co stworzyliśmy w poprzednich rozdziałach naszego kursu, daje nam w miarę dobry ogląd Megaparseca, ale zauważmy, że nie potrafimy zrobić jednej, bardzo istotnej rzeczy: parsować tekstu do haskellowych struktur danych. Do tej pory typy parserów to `Parser Char` -- niezbyt użyteczne. Teraz przystąpimy już do parsowania prawdziwego języka programowania, dobrze zdefiniowanego i posiadającego swoją gramatykę.

Sam język nazywa się WHILE i struktury danych pożyczymy sobie z tego tutoriala. Gramatyka jest następująca:

```
a   ::= x | n | - a | a opa a
b   ::= true | false | not b | b opb b | a opr a
opa ::= + | - | * | /
opb ::= and | or
opr ::= > | <
S   ::= x := a | skip | S1; S2 | ( S ) | if b then S1 else S2 | while b do
S
```

Skoro wiemy już **co** będziemy parsować, ustalmy struktury danych, **do** których będziemy parsować. Opiszmy każdą linijkę powyższej gramatyki i stwórzmy odpowiadający jej typ danych w Haskellu.

Wyrażenia arytmetyczne

Wyrażenia arytmetyczne to cokolwiek, co w rezultacie zwraca liczbę. W naszym przypadku może to być:

1. zmienna zawierająca liczbę całkowitą,
2. stała całkowitoliczbowa,
3. unarny minus, zaaplikowany do wyrażenia arytmetycznego (negacja liczby),
4. binarna operacja na dwóch liczbach (np. dodawanie).

Zapisując to w Haskellu zauważymy, że tak naprawdę powstaje nam rzecz niezwykle podobna do naszej gramatyki powyżej. To, jak algebraiczne typy danych (ADT) świetnie nadają się do wyrażania drzew syntaktycznych, jest jednym z powodów, dla których Haskell jest tak fajny do pisania parserów.

```
data AExpr = IntVar  String              -- 1.
           | IntLit  Int                 -- 2.
           | UMinus  AExpr               -- 3.
           | ABinary ABinOp AExpr AExpr  -- 4.
```

Wyrażenia logiczne (boolowskie)

Analogicznie, jak powyżej: wyrażenia boolowskie to cokolwiek, co w wyniku zwraca wartość logiczną (zwróćmy uwagę, że ich argumenty niekoniecznie muszą być wartościami logicznymi, jak w przypadku porównań liczb). W naszym prostym języku będziemy mieć następujące możliwości:

1. literał boolowski: true lub false,
2. unarna negacja (~a),
3. binarna operacja logiczna (i/lub),
4. binarna operacja porównania dwóch liczb całkowitych.

Mamy tutaj subtelny wybór: możemy zaimplementować stałe boolowskie jako jeden konstruktor z polem typu `Bool`, albo jako dwa konstruktory -- tak naprawdę nie ma to większego znaczenia. Skłaniamy się ku pierwszemu rozwiązaniu, żeby typ wyglądał podobnie do wyrażeń arytmetycznych.

```
data BExpr = BoolLit Bool               -- 1.
           | BNeg    BExpr              -- 2.
           | BBinary BBinOp BExpr BExpr -- 3.
           | BRel    ARelOp AExpr AExpr -- 4.
```

Operatory arytmetyczne

Nasz język wspiera: dodawanie, odejmowanie, mnożenie i dzielenie. Tworzymy na nie osobny typ (jest używany jako parametr w konstruktorze `ABinary` typu `AExpr`). W istocie jest to "tag", którego używamy do oznaczenia

konkretnej operacji (zwróćmy uwagę, że te konstruktory nie przyjmują żadnych parametrów). Tutaj zatem nie ma większej filozofii:

```
data ABinOp = Add | Sub | Mul | Div
```

Pozostałe operatory binarne

Podobnie, jak powyżej: mamy logiczną koniunkcję i alternatywę.

```
data BBinOp = And | Or
```

Oraz porównania:

```
data ARelOp = Less | Greater
```

Wyrażenia (statements)

Ostatnim (i bodaj najważniejszym) kawałkiem naszego języka są wyrażenia. Dla uściślenia: mówiąc "expression", mamy na myśli zdanie języka, które zwraca wartość. Mówiąc "statement", mamy na myśli coś, co może, ale nie musi zwracać wartości: może się po prostu wykonywać. Zdanie może być jednym i drugim, tj. możemy powiedzieć, że jednym z rodzajów statementu jest po prostu samo expression (jego wartość będzie wtedy zapewne zignorowana). W naszym języku co prawda to rozróżnienie nie jest bardzo widoczne, ale wydaje się istnieć w środowisku konsensus co do tej terminologii. W tym tekście będziemy używać angielskich nazw "expression" i "statement", żeby uniknąć nieporozumień.

Przypomnijmy sobie, jak zdefiniowaliśmy statement:

```
S   ::= x := a | skip | S1; S2 | ( S ) | if b then S1 else S2 | while b do
S
```

W naszym przypadku, statement może być jedną z następujących rzeczy:

1. podstawieniem wyrażenia arytmetycznego pod zmienną (identyfikowaną przez jej nazwę),
2. słowem kluczowym `skip`,
3. następującymi po sobie w sekwencji statementami (zauważmy, że w ten sposób można budować dowolnie długie sekwencje, więc równie dobrze możemy powiedzieć, że to po prostu sekwencja dowolnej ilości wyrażeń),
4. statementem w nawiasach,
5. wyrażeniem warunkowym, wykonującym jedno albo drugie "statement", w zależności od logicznego warunku,
6. pętlą `while`.

Struktura danych, która to będzie reprezentować, może wyglądać następująco:

```
data Stmt = Assign String AExpr    -- 1.
          | Skip                   -- 2.
          | Seq   [Stmt]           -- 3.
          | If    BExpr Stmt Stmt  -- 5.
          | While BExpr Stmt       -- 6.
```

Chwila... A co z numerem pięć (wyrażeniem w nawiasach)? Zastanówmy się, dlaczego nie jest to potrzebne, jako ćwiczenie.

Lekser

Czym jest lekser? Czy nie mieliśmy pisać parserów? Lekser jest, mówiąc ogólnie, preprocesorem, który przerabia strumień wejściowy (tekst) na ciąg tokenów. W ten sposób parser może być prostszy i działać wydajniej. W dużym uproszczeniu sprowadza się to do inteligentnego dzielenia tekstu po białych znakach i usuwania komentarzy. Nasz język wspiera dwa rodzaje komentarzy (jak w C): `//` i `/* */`. Na szczęście nie musimy implementować usuwania komentarzy samemu: Megaparsec wspiera to "out-of-the-box", zaraz zobaczymy jak.

Stwórzmy moduł `src/Lexer.hs` i zaimportujmy w nim `Text.Megaparsec.Char.Lexer`. Najpierw napiszmy funkcję, która będzie potrafiła przechodzić "pomiędzy" tokenami -- zjadając białe znaki i wszystkie rzeczy, które nasz parser ignoruje (komentarze). Nazwiemy go `spaceConsumer`:

```
import           Text.Megaparsec.Char       (space1)
import qualified Text.Megaparsec.Char.Lexer as L

spaceConsumer :: Parser ()
spaceConsumer = L.space L.space1 lineCmnt blockCmnt
  where
    lineCmnt  = L.skipLineComment "//"
    blockCmnt = L.skipBlockComment "/*" "*/"
```

Słowem wyjaśnienia: `L.space` służy wykrywaniu spacji/białych znaków/komentarzy. Jako parametrów oczekuje:

- parsera, który akceptuje spacje, ale nie puste wejście (my używamy `space1` z modułu `Text.Megaparsec.Char`)
- parsera akceptującego linijkę z komentarzem (używamy `L.skipLineComment`, podając mu, jak wygląda komentarz "linijkowy")
- parsera akceptującego blok komentarza (używamy `L.skipBlockComment`, podając mu, jak zaczyna i kończy się komentarz blokowy)

Następnie zdefiniujemy wrapper, który przyjmie parser i oprócz wykonania go, usunie spacje/białe znaki/komentarze **po nim**. Użyjemy do tego celu gotowej funkcji `L.lexeme`, którą częściowo zaaplikujemy, podając jako pierwszy argument nasz `spaceConsumer`. Uwaga: jeśli nie zaimportujemy modułu `Text.Megaparsec.Char.Lexer` w sposób kwalifikowany, dostaniemy konflikt nazw.

```
lexeme :: Parser a -> Parser a
lexeme = L.lexeme spaceConsumer
```

Kolejną przydatną funkcją będzie `symbol`, której podamy stały ciąg znaków, a ona da nam parser, który akceptuje dokładnie ten ciąg (dodatkowo zjadając spacje). Po raz kolejny tylko częściowo aplikujemy predefiniowaną funkcję z Megaparseca:

```
symbol :: String -> Parser String
symbol = L.symbol spaceConsumer
```

Zdefiniujemy sobie jeszcze parę przydatnych narzędzi:

1. coś w nawiasach:

```
parens :: Parser a -> Parser a
parens = between (symbol "(") (symbol ")")
```

Uwaga: `between` pochodzi z modułu `Control.Applicative.Combinators` z pakietu `parser-combinators`, ale jest reeksportowany przez moduł `Text.Megaparsec`, więc możemy ograniczyć liczbę nowych importów.

2. liczba:

```
integer :: Parser Integer
integer = lexeme L.decimal
```

3. średnik:

```
semi :: Parser String
semi = symbol ";"
```

Słowa kluczowe

Wykrywając słowa kluczowe, musimy mieć na uwadze pewną subtelność: o ile `not` jest słowem kluczowym, o tyle `notANumber` już nie. Czyli: słowo kluczowe musi być całością leksemu, a nie przedrostkiem jakiegoś dłuższego identyfikatora. Nie jest to trudne do zdefiniowania:

```
rword :: String -> Parser ()
rword w = (lexeme . try) (string w >> notFollowedBy alphaNumChar)
```

Zobaczmy, co się tutaj dzieje:

1. Każemy parserowi sparsować dokładnie taki napis, jak podaliśmy (np. `"while"`), a następnie zaakceptować go tylko, jeśli nie ma za nim żadnego "nie-białego" znaku. Nitpick: zamiast `>>` możemy tutaj użyć `*>` z modułu `Control.Applicative`, które nakłada słabsze wymagania na swoje operandy (co w ogólnym przypadku jest korzystne). Okazuje się, że te dwa operatory robią to samo -- wspominaliśmy, że `>>` w żaden sposób nie wykorzystuje tego, że jego operandy są w monadach.
2. Obkładamy to wszystko w `try`, którego sygnatura jest dość ciekawa: `try :: MonadParsec e s m => m a -> m a`. Czyżby nic nie robiło? `try`, jak sama nazwa wskazuje, spróbuje sparsować wejście używając parser przekazanego mu jako argument. Co więcej, jeśli parser-argument zwróci błąd, `try` również. Jednak, co istotne, "odda" wejście skonsumowane przez parser-argument. To ważne, gdy chcemy wycofywać się ze ścieżek parsowania (przykładowo: byliśmy przekonani, że parsujemy wyrażenie `while`, a okazało się to być tak naprawdę identyfikatorem).
3. Obkładamy wszystko leksemem, celem pochłonięcia białych znaków i komentarzy.

Zdefiniujmy sobie również listę wszystkich słów kluczowych:

```
rws :: [String]
rws =
["if","then","else","while","do","skip","true","false","not","and","or"]
```

Pozostaje nam parsowanie identyfikatorów (w naszym języku w praktyce to tylko nazwy zmiennych). Tutaj przyda się jeszcze dookreślenie, co może być identyfikatorem: napis składający się z nie-białych-znaków, zaczynający się od litery (czyli `hello` jest ok, ale `12angrymen` już nie). Popatrzmy:

```
identifier :: Parser String
identifier = (lexeme . try) (ident >>= check)
    where
        ident   = (:) <$> letterChar <*> many alphaNumChar
        check x = if x `elem` rws
                  then fail $ "keyword " ++ show x ++ " cannot be an
identifier"
                  else return x
```

Robimy tu parę rzeczy:

1. Parsujemy identyfikator dokładnie tak, jak sobie to zdefiniowaliśmy, za pomocą `ident` (opiszemy go dokładnie za chwilę).
2. Sprawdzamy, czy sparsowany identyfikator nie jest przypadkiem jednym ze słów kluczowych.
3. Robimy po raz kolejny trick z `try` i `lexeme` (opisany powyżej).

Trudne sygnatury: studium przypadku

Uwaga: w kolejnym akapicie popatrzymy na prostą funkcję, ale użyjemy tego jako pretekst do wyjaśnienia dość skomplikowanej sygnatury funkcji. Pomoże nam to nie dostawać ataku serca przy czytaniu dokumentacji do haskellowych bibliotek.

Parser `ident` robi bardzo prostą rzecz: parsuje pojedynczą literkę (`letterChar`), a następnie dołącza do niej listę znaków sparsowaną przez `many alphaNumChar`. Popatrzmy na typy:

1. `(:)` ma typ: `a -> [a] -> [a]`
2. `letterChar` ma typ `(MonadParsec e s m, Token s ~ Char) => m (Token s)`. O rany! Ale popatrzmy bliżej, okaże się, że to nic strasznego. `MonadParsec e s m` to w naszym przypadku po prostu `Parser s` (dokładnie z tego powodu definiowaliśmy sobie `type Parser = Parsec String String`). Pozostaje jeszcze magiczne `Token s ~ Char`. Nie wnikając w to zbytnio, możemy potraktować `~` jako `==`, tyle, że na typach. Czyli tak naprawdę w ten sposób funkcja `letterChar` wymaga czegoś od parsera, w kontekście którego działa: `s` -- drugi argument w `MonadParsec e s m` to typ strumienia wejściowego. W naszym przypadku to `String`, czyli `[Char]`. Tutaj wymagamy, żeby zaaplikowanie funkcji-na-typach `Token` na typie `s` dawało `Char`. Czyli mówimy: "podaj mi tutaj taką monadę `m`, żeby miała instancję `MonadParsec e s m`. Ale nie jakąkolwiek, tylko taką, żeby `Token s` było `Char`". Duża generyczność i bardzo silny system typów powoduje niestety takie zawroty głowy -- wszystko ma swoją cenę.
3. Uprośćmy ten typ, biorąc pod uwagę powyższe: w naszym kontekście funkcja `letterChar` ma typ `Parser Char`. Łatwiej?
4. Chcemy zaaplikować `(:)` na znaku, który sparsuje `letterChar`, a nie na całym parserze, więc używamy `fmap` (`<$>`), żeby działać wewnątrz parsera (pamiętamy instancję `Functor`?). Zatem `(:) <$> letterChar` ma typ `Parser ([Char] -> [Char])`.
5. `many alphaNumChar` ma typ (znowu upraszczając do naszego kontekstu) `Parser [Char]`.
6. Chcemy zaaplikować funkcję typu `Parser ([Char] -> [Char])` do `Parser [Char]`. Jak to zrobić? Hmm... Czy to nam czegoś nie przypomina? No właśnie: `(<*>) :: Applicative f => f (a -> b) -> f a -> f b`! Super: `(:) <$> letterChar <*> many alphaNumChar` ma typ `Parser Char` i robi dokładnie to, czego potrzebujemy w tym przypadku.

Lekcja: W Haskellu bardzo pomaga "podążanie za typami" -- jeśli mamy parę funkcji, które robią to, co chcemy, sposoby na ich poskładanie są narzucane przez system typów. Możemy, tak jak powyżej, składać je po kawałku i patrzeć na typy, jakie nam wychodzą. Często sygnatury funkcji bibliotecznych bywają mocno skomplikowane, bo projektowane są do działania na jak najbardziej ogólnych typach. Jednakże w większości przypadków, po zastosowaniu do naszego konkretnego przypadku, bardzo się uproszczą.

Ćwiczenie 4.1

Tą małą funkcję można napisać nie aplikatywnie, a monadycznie. Choć nie jest to w tym wypadku konieczne, spróbuj przepisać powyższą funkcję na monady.

Pytanie: czy dałoby się napisać `(ident >>= check)` bez użycia monad, a jedynie aplikatywów?

Ćwiczenie 4.2

Napisz testy leksera:

1. Parsowanie napisów i liczb aż prosi się o użycie QuickChecka.
2. Sprawdzenie, czy poprawnie parsowane są słowa kluczowe nie wymaga QuickChecka, wystarczy sprawdzić dla każdego elementu listy. Uwaga: należy sprawdzić też, czy parser poprawnie odrzuca wejście, w którym słowo kluczowe jest przedrostkiem dłuższego słowa.
3. Przy testach parsowania identyfikatorów QuickCheck może być pomocny.

Opcjonalna część dla zuchwałych: Type Families i obliczenia na typach

W "Studium przypadku" powyżej widzieliśmy funkcje działające na typach (`Token s`). Jak to się robi?

Odpowiedź na to pytanie jest następująca: Type Families, czyli "rodziny typów". Są bardzo podobne do typeklas, ale działają poziom wyżej, na typach. Można ich używać do wyliczania bardziej skomplikowanych predykatów w sygnaturach, ale też do bardzo zaawansowanych obliczeń na typach jak na przykład mapy działające na poziomie typów (zob. biblioteka `vinyl`). Do działania potrzebują rozszerzenia `{-# LANGUAGE TypeFamilies #-}` na górze pliku, w którym są użyte. Poniżej przedstawimy sobie bardzo prosty przykład użycia takiej rodziny typów.

Warto jeszcze nadmienić, że rodziny typów mogą być dwojakiego rodzaju: otwarte i zamknięte. Zamknięte są definiowane wewnątrz typeklas, otwarte -- same w sobie. Definiowane zamkniętych rodzin ma pewną przewagę: kompilator jest w stanie sam sprawdzać równość dwóch typów. Minusem są ograniczone możliwości ich rozszerzania. Przestudiujmy tworzenie najpierw zamkniętej, a potem otwartej rodziny typów.

Closed type families

Przykład jest wart więcej, niż tysiąc słów, więc pokażemy przykład i będzie jasne, do czego można tego używać. Wyobraźmy sobie, że potrzebujemy pisać obliczenia na czasie. W Haskellu służy do tego pakiet `time`, w którym mamy dwa podstawowe typy: `UTCTime` do wyrażania timestampów oraz `NominalTimeDiff`, wyrażający różnicę pomiędzy dwoma czasami. Nie możemy bezpośrednio ich dodawać, bo nie miałoby to sensu (są w innych jednostkach i mają różne interpretacje). Jest jednak funkcja `addUTCTime`, która pobiera `NominalTimeDiff` i dodaje go do `UTCTime`. Można również zwyczajnie dodawać do siebie `NominalTimeDiff`. Czy możemy napisać funkcję, która pozwoli na jednolite dodawanie czasu? Zobaczmy:

```
import           Data.Time (NominalDiffTime, UTCTime)
import qualified Data.Time as Time

class Add a b where
    type Res a b
    add :: a -> b -> Res a b
```

Mówimy tutaj: w klasa `Add` ma jedną metodę, `add`, która bierze obiekt typu `a`, obiekt typu `b` i zwraca... No właśnie: wyliczony na podstawie `a` i `b` typ wynikowy. Intuicyjnie:

- `UTCTime` + `NominalDiffTime` = `UTCTime`
- `NominalDiffTime` + `UTCTime` = `UTCTime`
- `NominalDiffTime` + `NominalDiffTime` = `NominalDiffTime`
- `UTCTime` + `UTCTime` nie ma sensu: czym jest 12:45 + 1:46? Napiszemy to, całkiem dosłownie, w deklaracjach instancji `Add`:

```
instance Add NominalDiffTime NominalDiffTime where
    type Res NominalDiffTime NominalDiffTime = NominalDiffTime
    add = (+)

instance Add UTCTime NominalDiffTime where
    type Res UTCTime NominalDiffTime = UTCTime
```

```
    add = flip Time.addUTCTime

instance Add NominalDiffTime UTCTime where
    type Res NominalDiffTime UTCTime = UTCTime
    add = Time.addUTCTime
```

Możemy teraz dodawać do siebie timestampy i różnice czasu!

Open type families

Na nieco bardziej wymyślny przykład pozwolimy sobie w przypadku otwartych rodzin typów. Użyjemy ich jako pretekstu do pokazania dwóch kolejnych, dość zaawansowanych ficzerów Haskella: polimorfizmu wyższych rzędów oraz tzw. type applications. Przykład będzie dotyczył bardzo prostej rzeczy: porównywania **typów**. Intuicyjnie: floatów jest więcej, niż intów, doubli więcej, niż floatów itd. Wiadomo oczywiście, jak napisać typeklasę, która dla poszczególnych typów parametrów będzie zwracała taki lub inny napis. Trick jednak polega na tym, by porównać typy **podczas kompilacji**, czyli na poziomie systemu typów. To oczywiście trywialny przykład, ale bardziej zaawansowane bazują na tym samym pomyśle.

Zacznijmy od podstaw. Skoro porównanie ma się odbyć na poziomie typów, zwracane przez nie wartości muszą być typami. Zdefiniujmy sobie:

```
data LT
data GT
data EQ
```

Ważne: to trzy oddzielne typy, a nie konstruktory jednego typu. Swoją drogą, są to tzw. **phantom types**: nie mają żadnego konstruktora, czyli nie da się utworzyć ich wartości. Napiszmy im jeszcze instancje `Show`:

```
instance Show LT where
    show _ = "LT"

instance Show GT where
    show _ = "GT"

instance Show EQ where
    show _ = "EQ"
```

Instancje muszą ignorować argument funkcji `show` -- nie ma żadnych wartości naszych typów. Takie sytuacje to jedna z nielicznych sytuacji, kiedy leniwa ewaluacja pomaga, a nie przeszkadza.

Mamy napisać funkcję na typach, więc piszemy:

```
type family TCompare a b
```

W ten sposób określamy, ile argumentów będzie przyjmować nasza funkcja, tutaj: dwa. Następnie piszemy instancje rodziny typów dla poszczególnych parametrów. Zuważmy, jak bardzo to przypomina pisanie kolejnych pattern-matchy w zwykłej funkcji:

```
type instance TCompare Int   Int   = EQ
type instance TCompare Float Int   = GT
type instance TCompare Int   Float = LT
type instance TCompare Float Float = EQ
```

To dość intuicyjne i bardzo podobne, do pisania zwykłej funkcji:

```
data CompRes = LT | GT | EQ

intCompare :: Int -> Int -> CompRes
intCompare 0 0 = EQ
intCompare 0 1 = LT
intCompare 1 0 = GT
-- i tak dalej...
```

Widzimy, że w naszym przykładzie z type families mamy:

- typy danych zamiast konstruktorów/wartości jednego typu
- typy (`Int`, `Float`, [...]) zamiast wartości tych typów (`0`, `1`, [...])
- deklarację `type family [...]` zamiast sygnatury funkcji
- kolejne `type instance [...]` zamiast kolejnych pattern matchy

Na razie wszystko jest naprawdę proste. Większość z nas zadaje sobie jednak pytanie: jak możemy użyć takiej funkcji-na-typach? W Haskellu typów używamy (explicite) w deklaracjach danych i w sygnaturach funkcji. Deklaracje danych to nieco osobny temat i zainteresowanych odsyłamy do powiązanego tematu: data families. My zajmiemy się sygnaturami funkcji. Zróbmy prostą rzecz: funkcję, która przyjmuje dwa argumenty, porównuje ich typy przy użyciu naszego `TCompare`, a następnie wypisuje jego tekstową reprezentację. Proszę zapiąć pasy, napiszemy wesołą sygnaturę funkcji.

```
tcompare :: forall a b c. (c ~ TCompare a b, Show c) => a -> b -> String
tcompare _ _ = show (undefined :: c)
```

Ktoś (słusznie) mógłby uznać, że trochę przesadziliśmy. My jednak chcemy nauczyć się nie tyle pisać takie kody, co przestać się ich bać, gdy zobaczymy je w cudzym kodzie. Zacznijmy wyjaśnianie tej funkcji od jej lewego-górnego rogu, czyli słowa kluczowego `forall`.

Kwantyfikowane typy danych

`forall`, które widzimy w naszej sygnaturze (w `forall a b c`) jest tym samym, co odwrócone "A" w dowodach na zajęciach teorii mnogości. Tam służy nam ono do opisywania zbiorów. Tutaj mamy do czynienia z

typami, ale jedno ich rozumienie to właśnie jako zbiory wartości. Zastanówmy się, co mówi nam taka sygnatura funkcji:

```
f :: a -> a
```

Mówi nam, tak naprawdę: "funkcja `f` DLA DOWOLNEGO typu `a`, jeśli dostanie wartość tego typu, to zwróci wartość tego samego typu. Ano właśnie, dla dowolnego typu. Okazuje się, że Haskell w takiej sytuacji sam dopisuje sobie taki duży kwantyfikator i sygnatura tak naprawdę wygląda tak:

```
f :: forall a. a -> a
```

Zastanówmy się przez chwilę, jakie to ma implikacje. Czy takich funkcji jest wiele? Popatrzmy jeszcze raz: **DLA DOWOLNEGO** typu `a`, ta funkcja ma nam zwrócić wartość typu `a`. Czy tą definicję spełnia na przykład funkcja `g x = x + 1`? Przecież, jeśli podamy jej liczbę, to zwróci nam liczbę. Tutaj jednak sprawa rozbija się o "dla dowolnego". Ta funkcja zwróci wartość tego samego typu tylko dla typów numerycznych, czyli nie dla wszystkich. I faktycznie, Haskell jej typ podaje jako:

```
g :: forall a. Num a => a -> a
```

Okazuje się, że funkcja o sygnaturze takiej, jak `f`, istnieje **tylko jedna** i jest to funkcja identycznościowa (`f x = x`). Oczywiście takie wnioskowanie można przeprowadzić tylko w czysto funkcyjnym języku. Można też dowodzić sobie dużo ciekawszych własności na podstawie sygnatur funkcji, a nawet automatycznie generować w ten sposób testy (AutoSpec). Zainteresowanych odsyłam do fenomenalnego artykułu Philipa Wadlera.

Chwila! Skoro Haskell sam dopisuje sobie słówko `forall` przy sygnaturach, po co w ogóle je pisać? Okazuje się, że są przypadki, w których chcemy, żeby polimorfizm zadziałał nieco inaczej, niż domyślnie. Popatrzmy na funkcję:

```
poly :: (forall a. a -> a) -> Bool
poly f = (f 0 < 1) == f True
```

Teraz to nie tyle nasza funkcja jest polimorficzna, co spodziewa się w pełni polimorficznej funkcji jako argumentu! Teraz możemy przetestować, czy powyżej napisałem prawdę!

```
> poly id
True
> poly (+1)
<booom!>
```

I faktycznie, okazuje się, że `(+1)` nie jest w pełni polimorficzne! Zależnie od tego, jak głęboko zagnieżdżone mamy kwantyfikatory, uzyskujemy polimorfizm różnych rzędów. Wszystko można włączyć za pomocą

rozszerzenia `RankNTypes`.

Równość typów

Uzyskiwanie polimorfizmu wyższych rzędów to jednak nie jedyna rzecz, do której `forall` się nam przyda. Bo oto okazuje się, że jeśli za jego pomocą wprowadzimy sobie zmienne "typowe", możemy zacząć wykonywać na nich obliczenia i do nich "podstawiać". Popatrzmy:

```
forall a b c. (c ~ TCompare a b)
```

Teraz mówimy tak: "dla wszystkich typów `a`, `b` i `c` takich, że `c` to to samo, co `TCompare a b`". Od tej pory kompilator będzie wiedział, że jeśli w funkcji pojawi się gdzieś typ `c`, może on zostać wyliczony na podstawie typów `a` i `b`. W naszej funkcji `tcompare` nakładamy jeszcze na niego dodatkowe ograniczenie: typ `c` musi mieć instancję `Show`.

Uwaga ogólna: o sygnaturach typów można myśleć jako o zawężaniu zbioru wartości, które tworzą ten typ.

Typy bez wartości

Typy bez wartości wcale nie są bezwartościowe. W naszym przypadku, znowu korzystając z leniwości Haskella, możemy napisać: `show (undefined :: c)` i liczyć, że implementacja `show` dla typu `c` będzie sobie potrafiła poradzić, nie używając swojego argumentu. My napisaliśmy instancje `Show` właśnie w ten sposób: `show _ = "LT"` -- będzie okej!

Uwaga ogólna: nawet mając taką pewność, należy unikać używania `undefined` w kodzie. To niestety proszenie się o produkcyjne błędy. W książce/wykładzie na pewno produkcyjny błąd nie będzie miał miejsca, ale zawodowi programiści powinni czuć się ostrzeżeni.

No dobrze, wiemy już chyba wszystko. Nasza naprawdę-skomplikowana-funkcja najpierw wylicza sobie (podczas kompilacji) rezultat `TCompare a b`, a potem stara się dopasować dla znalezionego typu instancję `Show`. Przetestujmy:

```
> TF.tcompare (undefined :: Float) (undefined :: Int)
"GT"
> TF.tcompare (undefined :: Int) (undefined :: Int)
"EQ"
```

Działa! Ale chwila moment... Czy właśnie nie odradzaliśmy używania `undefined`? Tutaj nie potrzebujemy tych wartości, ale przydałoby się umieć unikać takich sytuacji. Cały problem naszej funkcji `tcompare` jest taki, że przyjmuje argumenty, ale ich nie używa. A gdyby tak dało się tego uniknąć? Umiemy już wprowadzać zmienne typu: za pomocą `forall`! Napiszmy najpierw sygnaturę:

```
tcompare' :: forall a b c. (c ~ TCompare a b, Show c) => String
```

Ma to sens. Co z implementacją? Podobnie, jak powyżej, tylko bez niepotrzebnych argumentów:

```
tcompare' :: forall a b c. (c ~ TCompare a b, Show c) => String
tcompare' = show (undefined :: c)
```

Wszystko pięknie, tylko jak tego użyć? Czyżby istniał sposób, żeby explicite powiedzieć kompilatorowi "zmienna `a` to typ `Int`, a zmienna `b` to typ `Float`"? Istnieje pod postacią rozszerzenia Type Applications (w GHCi: `:set -XTypeApplications`). Teraz możemy bezpośrednio przekazać typy w ten oto sposób:

```
> tcompare' @Int @Float
"LT"
> tcompare' @Float @Float
"EQ"
```

Taka składnia jest przydatna nie tylko przy skomplikowanym kodzie z type families. Przykład bliższy sercu programisty to funkcja `malloc` z modułu `Foreign.Marshal.Alloc`. Służy do alokowania pamięci dla określonego typu. Podobnie jak jej odpowiednik z języka C zwraca pointer, ale nie przyjmuje ilości bajtów do zaalokowania: domyśla się tego z kontekstu. Jaki ma typ?

```
malloc :: forall a. Storable a => IO (Ptr a)
```

Rozumiemy już takie sygnatury: dla dowolnego typu, który ma instancję `Storable` możemy zaalokować pointer do tego typu. Bez type applications użycie do alokacji pointera do inta wyglądałoby mniej więcej tak:

```
ptr <- malloc :: IO (Ptr Int)
```

Z naszą piękną składnią możemy napisać dużo czytelniej i bardziej ogólnie:

```
ptr <- malloc @Int
```

Uwaga końcowa

Należy pamiętać, że na świecie nie ma nic za darmo. Stosując takie zaawansowane tricki dostajemy kod, który jest piękny i generyczny, ale ma dwie bardzo poważne wady:

1. jest trudny do zrozumienia dla innych programistów
2. bardzo wolno się kompiluje

"Zasadą kciuka" powinno być używanie zaawansowanych konstrukcji języka dopiero w przypadku, kiedy inne, bardziej podstawowe, zawiodły. Czym innym jest biblioteka, która będzie użyta w setkach różnych projektów, a czym innym kod produkcyjny, który najprawdopodobniej ma tylko jedno wcielenie. Za wielką generyczność płaci się wielką cenę w postaci skomplikowania i czasów kompilacji (które mają ogromny wpływ na produktywność programistów w projekcie).

Parser

Statementy

Skoro umiemy już ładnie dzielić wejście na tokeny, pozostaje nam konstruowanie struktur danych, które wcześniej sobie stworzyliśmy. Tutaj tak naprawdę dzieje się cała logika parsowania. To, co robiliśmy w lekserze było tak naprawdę (mało twórczą) techniczną koniecznością. Zacznijmy od parsowania sekwencji statementów:

```
stmt :: Parser Stmt
stmt = f <$> sepBy1 stmt' semi
    where f l = if length l == 1 then head l else Seq l
```

Funkcja `sepBy1` zwróci listę jednego lub więcej statementów, które będzie dzielić po podanym separatorze (u nas to średnik, parsowany przez `semi`). Puste wejście da nam błąd, wejście z tylko jednym statementem zwróci tenże statement (zajmuje się tym funkcja `f`). Lista dwóch lub więcej statementów musi jeszcze zostać zapakowana w nasze AST: `Seq l`. Pamiętajmy, że o ile `Stmt` i `Seq [Stmt]` są typami w naszym AST, o tyle samo `[Stmt]` nie jest! Umiemy parsować sekwencje, ale nie umiemy parsować pojedynczych statementów (do tego służy niezdefiniowana jeszcze funkcja `stmt'`, której użyliśmy powyżej). Możemy ją napisać patrząc na typ `Stmt`, który mamy zdefiniowany:

```
stmt' :: Parser Stmt
stmt' =  ifStmt
     <|> whileStmt
     <|> skipStmt
     <|> assignStmt
     <|> parens stmt
```

Nasz stary przyjaciel: `Alternative`! Statement to `ifStmt`, albo `whileStmt`, itd. Zauważmy, że parser piszemy top-down, więc używamy funkcji, które dopiero sobie zdefiniujemy. Jeśli chcemy tylko sprawdzić, czy się to dobrze otypowało, możemy zdefiniować sobie tylko nagłówki funkcji:

```
ifStmt :: Parser Stmt
ifStmt = undefined
```

Dyskusja na później to: jakiego typu jest `undefined`, że możemy go użyć w absolutnie dowolnym wyrażeniu, o dowolnym typie? Czyżby system typów Haskella jednak nie był taki spójny? Uwaga: taka funkcja skompiluje się i kompilator sprawdzi, czy typy się zgadzają. Podczas wykonania najprawdopodobniej rzuci jednak błędem (chyba, że z jakiegoś powodu nie zostanie wyewaluowana, jak to się często w Haskellu, chcący lub niechcący, zdarza).

Teraz nie pozostaje nic innego, jak mozolnie zdefiniować parsery dla całego drzewka. Zacznijmy od `if`:

```
ifStmt :: Parser Stmt
ifStmt = do
```

```
    rword "if"
    cond  <- bExpr
    rword "then"
    stmt1 <- stmt
    rword "else"
    stmt2 <- stmt
    return $ If cond stmt1 stmt2
```

Pytanie

Czy powyższy parser możemy napisać aplikatywnie, bez monad i "<-"?

Ćwiczenie 4.3

Parsowanie `if` jest stosunkowo podobne do `while`, gdzie musimy sparsować:

- słowo kluczowe `while`
- warunek, będący boolowskim wyrażeniem
- słowo kluczowe `do`
- blok wyrażeń (sekwencję -- jedno lub więcej)

Po sparsowaniu musimy zwrócić odpowiednią strukturę (`While`).

Teraz przejdźmy do pozostałych statementów: podstawienia i "skip".

```
assignStmt :: Parser Stmt
assignStmt = do
    var  <- identifier
    symbol ":="
    expr <- aExpr
    return $ Assign var expr
```

Co tu się dzieje? Spodziewamy się czegoś pokroju `x := y`, więc najpierw parsujemy identyfikator. Następnie oczekujemy konkretnego symbolu: `:=`. `symbol` zwraca konkretny rezultat, więc GHC bedzie nas ostrzegać o ignorowaniu rezultatu wyrażenia. Żeby temu zapobiec, możemy explicite użyć funkcji `void :: Monad m => m a -> m ()`, która wykonuje akcję, ale ignoruje jej wartość zwracaną. Zostaje nam jeszcze `skip`, które definiujemy stosunkowo prosto:

```
skipStmt :: Parser Stmt
skipStmt = Skip <$ rword "skip"
```

Czym jest zagadkowe `<$`? Sprawdźmy sygnaturę: `(<$) :: Functor f => a -> f b -> f a`. Bierzemy jakąś wartość, drugą wartość "w pudełku" i zastępujemy zawartość pudełka tą pierwszą wartością. `Skip` nie ma żadnych argumentów, więc możemy zwrócić je "w ciemno", ale musimy być pewni, że `rword "skip"` zostało wykonane.

Wyrażenia

Parsowanie wyrażeń (np. arytmetycznych) może być nieco mozolne. Operatory, które przyjmują dwa argumenty i są infixowe (czyli pisze się je pomiędzy ich argumentami), pociągają za sobą konieczność zdefiniowania dodatkowych reguł, przede wszystkim: jak nawiasujemy? Przykładowo: wiemy, że mnożenie wykonujemy przed dodawaniem, czyli 2 + 2 * 2 to 2 + (2 * 2), oraz że dodawanie jest łączne do lewej strony, czyli 2 + 2 + 2 to (2 + 2) + 2.

Pytanie

Wydaje się, że większość operacji nawiasuje się od lewej do prawej (jak dodawanie). Pytanie: czy potrafimy znaleźć jakiś prosty przykład operacji, którą nawiasuje się od prawej do lewej? Możemy pomóc sobie uruchamiając GHCi, które razem z typem wyświetla priorytet i łączność operatorów. Sprawdźmy:

```
> :i (+)
class Num a where
  (+) :: a -> a -> a
  ...
        -- Defined in ‘GHC.Num’
infixl 6 +

> :i (*)
class Num a where
  ...
  (*) :: a -> a -> a
  ...
        -- Defined in ‘GHC.Num’
infixl 7 *
```

Ostatnia linijka mówi nam, że `+` jest łączny do lewej i ma niższy priorytet, niż mnożenie: czyli tak, jak się spodziewaliśmy.

Pisanie wszystkich reguł nawiasowania i priorytetów operatorów jest zadaniem tyleż mozolnym, co łatwo automatyzowalnym. My nie będziemy bawić się w robienie tego samodzielnie, tylko użyjemy pakietu `parser-combinators`, który zrobi to za nas. Zdefiniowana jest tam funkcja `makeExprParser`, która przyjmuje, oprócz parsera do wyrażeń, tablicę pierwszeństw operatorów. Składnia wykorzystuje tą samą nomenklaturę, którą już widzieliśmy, dodając jeszcze `InfixN`, dla operatorów, które nie są łączne (jak np. porównanie) oraz `Prefix` i `Postfix`.

Swoją drogą, w Pythonie: `1 < x < 2` to odpowiednik `1 < x and x < 2`, a nie `(1 < x) < 2` czy `1 < (x < 2)`. W Haskellu takie wyrażenie daje błąd, ale wcale nie typów, jak moglibyśmy się spodziewać, a parsowania!

```
> 1 < x < 2
<interactive>:3:1: error:
    Precedence parsing error
        cannot mix ‘<’ [infix 4] and ‘<’ [infix 4] in the same infix
expression
```

Zachowanie funkcji `makeExprParser` najlepiej po prostu zobaczyć na przykładzie. Zacznijmy pisać top-down, parsery dla wyrażeń boolowskich (`bExpr`).

```
bExpr :: Parser BExpr
bExpr = makeExprParser bTerm bOperators
```

Świetnie! Teraz pozostaje tylko skonstruować tabelę priorytetów operatorów: operatory występujące wcześniej mają wyższy priorytet. Jest to lista list: każdy element jest listą operatorów o takim samym priorytecie. W naszym przypadku to wygląda tak: `[[not], [and, or]]`, a w prawdziwym kodzie tak:

```
bOperators :: [[Operator Parser BExpr]]
bOperators =
    [ [ Prefix (BNeg <$ rword "not") ]
    , [ InfixL (BBinary And <$ rword "and")
      , InfixL (BBinary Or  <$ rword "or")
      ]
    ]
```

Możemy tutaj zaobserwować popularną w Haskellowym świecie fascynację wyrównywaniem kodu.
Parę narzędzi jest w stanie nam w tym mocno pomóc, przede wszystkim: `stylish-haskell`.

Pozostaje nam jeszcze zdefiniować jak wyglądają operandy naszych `bOperators` (`bTerm`). Mogą być wszystkim, co w wyniku da wartość logiczną, czyli:

- stałą (true lub false)
- wyrażeniem boolowskim w nawiasach
- porównaniem

Napiszmy to w kodzie:

```
bTerm :: Parser BExpr
bTerm  =  parens bExpr
      <|> (BoolLit True  <$ rword "true")
      <|> (BoolLit False <$ rword "false")
      <|> rExpr
```

Brakuje nam już tylko porównań:

```
rExpr :: Parser BExpr
rExpr = do
    a1 <- aExpr
    op <- relation
    a2 <- aExpr
    return $ BRel op a1 a2
```

```
relation :: Parser ARelOp
relation  =  (Greater <$ symbol ">")
         <|> (Less    <$ symbol "<")
```

I gotowe. Prawie. Jeszcze wyrażenia arytmetyczne, które definiuje się podobnie do boolowskich, ale nieco prościej. Z tym związane są trzy ostatnie ćwiczenia w tej części.

Ćwiczenie 4.4

Zdefiniuj parser(y) dla wyrażeń arytmetycznych. Potrzebne będzie zdefiniowanie parsera `aExpr`, który wykorzysta tabelę operatorów arytmetycznych (`aOperators`) i zdefiniuje, co może być operandem wyrażenia arytmetycznego (`aTerm`): zmienna lub stała.

Ćwiczenie 4.5

Napisz testy parsera. Powinny sprawdzać, czy otrzymujemy odpowiednie struktury podczas parsowania kawałków kodu. Wymaga to jeszcze jednego dodatku: obecnie nasze AST nie daje się porównywać (typy nie mają instancji `Eq`. Można ją derive'owawać, a można napisać własnoręcznie: czasami jest to przydatne, jeśli równość jest z jakiegoś powodu niestandardowa, np. ignorujemy automatycznie generowane id).

Ćwiczenie 4.6 (*)

Ćwiczenie "z gwiazdką", wymagające nieco więcej pracy. Umiemy przerabiać tekst programu na drzewo syntaktyczne i najczęściej to jest kierunek, który nas interesuje. Ale okazuje się, że czasem przydatne jest zrobienie operacji odwrotnej: wygenerowaniu kodu na podstawie AST. Może się to przydać na przykład przy automatycznym formatowaniu kodu: parsując go, a następnie generując kod z AST, efektywnie sformatujemy kod tak, jak sobie tego życzymy. Taki proces zwykle nazywa się "pretty printing".

Wskazówka: prawdopodobnie najłatwiej osiągnąć to, co chcemy, tworząc typeclassę (np. `PrettyPrint`) z jedną metodą (np. `prettyPrint`). Sam pretty printing można zaimplementować rekurencyjnie.

Dalsze prace

Mamy już cały parser, ale ani razu go nie użyliśmy. Spróbujmy więc...

```
> Parsers.parse RealParser.stmt "x := 2; if x < 2 then y := 5 else y := 6"
Right (Seq [Assign "x" (IntLit 2),If (BRel Less (IntVar "x") (IntLit 2))
(Assign "y" (IntLit 5)) (Assign "y" (IntLit 6))])
```

Wydaje się, że działa! Pytanie teraz: co możemy zrobić z takim pięknym AST? Skoro piszemy język programowania, dobrze byłoby móc wykonywać programy w nim napisane. Skoro mamy parser, brakuje interpretera! To będzie tematem następnych odcinków.

[Opcjonalne] Ćwiczenie 4.7

Dopisz do naszego języka kolejny rodzaj statementu: `print`. Będzie to rodzaj interfejsu ze światem, kiedy już będziemy umieli interpretować programy.