Středoškolská odborná činnost

Líně, čistě, funkcionálně

Užití funkcionálního programovacího jazyka Haskell k řešení úloh Ústředních kol ČR Soutěže v programování

Jan Špaček

Ostrava 2013

Středoškolská odborná činnost

Obor SOČ: 18 Informatika

Líně, čistě, funkcionálně

Užití funkcionálního programovacího jazyka Haskell k řešení úloh Ústředních kol ČR Soutěže v programování

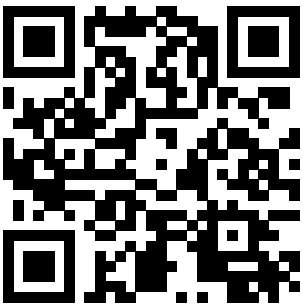
Lazily, purely, functionally

Using the functional programming language Haskell to solve tasks from the finals of the Czech Programming Contest

Autor: Jan Špaček

Škola: Wichterlovo gymnázium
Čs. exilu 669
Ostrava-Poruba

Ostrava 2013



Prohlášení

Prohlašuji, že jsem svou práci vypracoval samostatně, použil jsem pouze podklady (literaturu, SW atd.) uvedené v přiloženém seznamu a postup při zpracování a dalším nakládání s prací je v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) v platném znění.

V ..... dne ..... podpis: .....

Abstrakt

Funkcionální programování, vycházející z λ-kalkulu, tvoří alternativu k dnes široce využívanému imperativnímu přístupu k programování, reprezentovanému jazyky jako C, C++, Java, C#, Objective-C, JavaScript, Python, Ruby či PHP.

V této práci je představeno řešení dvou úloh Ústředního kola ČR Soutěže v programování z let 2010 a 2012 s využitím líně vyhodnocovaného funkcionálního jazyka Haskell. První úloha je implementace jednoduchého programovacího jazyka, podobného jazyku Logo, sloužícího k vykreslování želví grafiky. Druhá úloha se zabývá hledáním nejkratší cesty v bludišti s pohyblivými překážkami a omezenou schopností procházet zdmi.

Cílem práce je prezentovat funkcionální programování a techniky, které se v něm využívají, a ukázat, že tento programovací styl není pouze předmětem akademického výzkumu, ale že se dá využít i k řešení „skutečných“ úloh.

Klíčová slova Funkcionální programování; Haskell; želví grafika; programovací jazyky; hledání cest.

Abstract

Functional programming has its roots in λ-calculus and is an alternative to the mainstream imperative paradigm, represented by languages as C, C++, Java, C#, Objective-C, JavaScript, Python, Ruby or PHP.

In this paper we present solutions to two tasks from the Czech Programming Contest, a competition for secondary and high school students, from the final rounds held in 2010 and 2012. The first task is an implementation of a simple programming language drawing turtle graphics based on Logo. The second task is a variation of pathfinding in a maze with moving obstacles and a limited ability to get through the walls.

The goal of the paper is to present functional programming and its technique and to show that this style of programming is not just a subject of academic research, but might be used to solve “real-world” problems.

Keywords Functional programming; Haskell; turtle graphics; programming languages; pathfinding.

Obsah

Kapitola 1

Úvod

Funkcionální programování má původ v λ-kalkulu, matematickém formálním systému, jež popisuje výpočty jako vyhodnocování výrazů tvořených funkcemi. S funkcemi se v λ-kalkulu pracuje jako s hodnotami[1] a lze je *aplikovat* na argument a získat jejich výsledek nebo je vytvořit pomocí λ-*abstrakce*. Tento systém je svou výpočetní silou ekvivalentní Turingovu stroji, je v něm tedy možno vyjádřit jakýkoli strojem proveditelný výpočet.

V *imperativních* programovacích jazycích (C, C++, Java, Ruby, JavaScript, Befunge, PHP...) jsou algoritmy vyjádřeny jako sekvence kroků – příkazů, které mění stav programu, zvláště hodnoty proměnných. Postupným vykonáváním těchto kroků se provádí výpočet. Tento přístup přímo vychází ze strojového kódu počítačů a v dnešní době jde o dominantní programovací styl.

Ve funkcionálních jazycích se naproti tomu výpočty provádí *vyhodnocováním výrazů*. Aplikace („zavolání“) funkce má jediný účinek, a to je získání jejího výsledku. Jelikož neexistuje žádný stav, na němž by výsledek funkce mohl záviset, je garantováno, že aplikujeme-li funkci vícekrát na stejné argumenty, dostaneme vždy stejný výsledek. Tato vlastnost se nazývá *referenční transparentnost*.

Na funkce je tedy možno nahlížet podobně jako na funkce v matematice, takže pro programátora i pro kompilátor je snadné zjišťovat a *dokazovat* chování funkcí. Kompilátor může kód Haskellu snadno optimalizovat [16], takže programátor může v programech používat vysokoúrovňové konstrukce bez obav z negativního dopadu na rychlost programu.

Další vlastností funkcionálních jazyků je snadná paralelizace. Potřebujeme-li zjistit hodnotu dvou výrazů, můžeme bez obav každý výpočet provést na samostatném procesoru, jelikož máme zaručeno, že se výpočty nebudou navzájem ovlivňovat.

1.1 Řešené úlohy

V této práci je představeno řešení dvou úloh z programátorských soutěží s užitím funkcionálního jazyka Haskell. Úlohy pochází z Ústředních kol Soutěže v programování, vyhlašované Ministerstvem školství, mládeže a tělovýchovy a pořádané Národním institutem dětí a mládeže, z let 2010 (Krunimír – želví grafika) a 2012 (Bílá paní – hledání cesty v bludišti),

Tyto úlohy byly vybrány, protože nejsou ani příliš rozsáhlé, ani příliš jednoduché, a není je možno podezírat, že by byly vytvořeny funkcionálnímu jazyku „na míru“.

Zadání obou původních úloh je možno nalézt jak na Internetu na stránkách soutěže ([2], [1]), tak přetištěné v této práci na stranách 61 a 63.

1.2 Technická stránka práce

V této práci jsou vytvořeny dva programy, jež řeší tyto úlohy. Kód je zapsán ve stylu *literárního programování*, kdy se volně střídá kód srozumitelný pro počítač a komentář pro člověka, což

[1] Základní λ-kalkulus dokonce žádné jiné hodnoty než funkce nezná.

znamená, že zdrojový kód Haskellu obou programů je zároveň zdrojovým kódem systému LaTeX, kterým je tato práce vysázena.

Texty kapitol 2 a 3 jsou tedy kompletní programy, které je možno přeložit a spustit. Tímto je zajištěno, že zde prezentovaný kód je plně funkční.

Zdrojový kód práce je spravován pomocí distribuovaného systému správy verzí Git. Repozitář je volně přístupný na serveru github.com,[2] adresa práce je `https://github.com/honzasp/funsp`. Detailnější popis jednotlivých souborů a složek se nachází v souboru `README`.

Jako kompilátor byl využit GHC (Glasgow Haskell Compiler), který je dnes mezi překladači Haskellu dominantní.

1.3 Haskell

Haskell je *de-facto* standardní líný čistě funkcionální jazyk. Jeho vývoj začal v září roku 1987, přičemž první verze jazyka (Haskell 1.0) byla vydána 1. března 1990. Postupně následovaly verze 1.1 až 1.4. V roce 1999 byla vytvořena „stabilní“ verze Haskell 98, která byla později revidována [11]. [9]

Od roku 2006 probíhá proces nazývaný Haskell' (Haskell Prime), jehož cílem je vytvářet nové revize standardu každý rok. První a zatím poslední revizí je Haskell 2010 [3].

V následujících podsekcích budou představeny základní vlastnosti jazyka Haskell, v práci ovšem budeme předpokládat jistou základní znalost funkcionálního programování v tomto jazyku. Pokud se čtenář s tímto stylem programování ještě nesetkal, autor mu může doporučit knihu slovinského studena Mirana Lipovači *Learn You a Haskell for Great Good* [14], která je volně dostupná na Internetu jak v původní anglické verzi,[3] tak v částečném českém překladu.[4] Práce by nicméně měla být srozumitelná i se základními znalostmi „klasického“ programování.

1.3.1 Čistý funkcionální jazyk

Haskell je *čistý* funkcionální jazyk,[5] což znamená, že *všechny* funkce jsou referenčně transparentní a nemají žádný vnitřní stav, jsou tedy prosty vedlejších účinků.

Většina jazyků, které bývají označovány jako funkcionální, totiž funkce s vedlejšími účinky povoluje, například pro vykonávání vstupně-výstupních operací nebo používání měnitelných datových struktur, potřebných pro efektivní implementaci některých algoritmů. Haskell ovšem pro tyto účely používá jiné prostředky, především *monády*, které si představíme v podsekci 1.3.5 a které umožňují zachovat čistotu jazyka.

1.3.2 Líné vyhodnocování

Haskell je *líně vyhodnocovaný* jazyk, což znamená, že hodnoty výrazů se vyhodnocují až ve chvíli, kdy je to nezbytně nutné, přičemž hodnoty, které nejsou potřeba, se nevyhodnotí vůbec. Tím se nejen zvýší efektivita programu (neprovádí se zbytečné výpočty), ale hlavně se programátor zbaví nutnosti zabývat se pořadím vyhodnocování výrazů. Je tedy možné například snadno oddělit kód produkující a kód konzumující data, čímž se zvýší modularita. [10]

Ve funkcionálních programech je běžné používat nekonečné struktury, např. nekonečné seznamy či nekonečně se větvící stromy, a nechat vyhodnotit pouze tu část dat, která je potřeba k získání požadovaného výsledku.

1.3.3 Statické typování

Statické typování znamená, že typ každého výrazu je znám již v při kompilaci programu, takže případné chyby kompilátor odhalí velmi brzy. Není tedy možné, aby program za běhu zhavaroval

[2] `http://github.com`

[3] `http://learnyouahaskell.com/`

[4] `http://naucte-se.haskell.cz/`

[5] Někdy se používá rovněž označení „čistě funkcionální jazyk“, které ovšem není úplně přesné.

na typovou chybu, čímž se eliminuje celá škála potencionálních bugů. Většinou platí, že když se kód úspěšně zkompiluje, máme velkou šanci, že bude už napoprvé skutečně fungovat tak, jak si představujeme.

Narozdíl od jazyků jako Java či C++, jež jsou také staticky typované, je mnohem silnější typový systém Haskellu schopen naprostou většinu typů odvodit sám, takže typové anotace se obvykle používají jen jako druh dokumentace a způsob kontroly určené pro člověka (programátora).

1.3.4 Typové třídy

S typovým systémem úzce souvisí *typové třídy*. Typové třídy byly do Haskellu zavedeny, aby se vyřešil problém s „přetěžováním" funkcí jako je například porovnávání (`==`), které bychom potřebovali použít s větším množstvím typů (operace ekvivalence má smysl např. pro čísla, řetězce, seznamy, množiny...).

Ukažme si příklad typové třídy `Eq`, která slouží k implementaci porovnání dvou hodnot:

```
class Eq a where
  (==) :: a -> a -> Bool
```

Tímto deklarujeme, že typ `a` patří do třídy `Eq` právě tehdy, když implementuje *metodu* `==`.

Mějme dva datové typy:

```
data Color = Red | Green | Blue
data Bit = On | Off
```

Pro oba tyto typy určitě dává operace porovnání smysl, proto můžeme nadefinovat *instanci* třídy `Eq`:[6]

```
instance Eq Color where
  Red   == Red   = True
  Green == Green = True
  Blue  == Blue  = True
  _     == _     = False

instance Eq Bit where
  On  == On  = True
  Off == Off = True
  _   == _   = False
```

Touto deklarací specifikujeme, že typy `Color` a `Bit` jsou instancemi třídy `Eq`, a poskytneme implementaci metody `==`. Nyní můžeme používat funkci `==` s barvami i bity, např. `Red == Red` vrátí `True` a `On == Off` vrátí `False`. Zároveň ale nemůžeme omylem porovnávat barvy a bity – napíšeme-li `Red == On`, kompilátor ohlásí typovou chybu, jelikož funkce `==` akceptuje pouze hodnoty stejného typu.

Kdybychom chtěli funkci, která otestuje, jestli se daný prvek nachází v seznamu, mohli bychom ji nadefinovat takto:

```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) = if x == y then True else elem x ys
```

Typ této funkce, `Eq a => a -> [a] -> Bool`, odráží skutečnost, že tato funkce je definovaná pouze pro takové typy `a`, které náleží do třídy `Eq`. Můžeme ji tedy použít jak na seznam barev (`[Color]`), tak na seznam bitů (`[Bit]`).

Instancemi tříd samozřejmě nemusí být jen takovéto jednoduché typy. Můžeme si nadefinovat typ reprezentující binární strom:

```
data Tree a = Node (Tree a) (Tree a) | Leaf a
```

Tento typ bychom obratem mohli učinit instancí třídy `Eq`:

[6] Haskell umožňuje pro datové typy automaticky odvodit instance tříd ze standardní knihovny, jako je `Eq` nebo `Ord`, pomocí klauzule `deriving`, takže jen zřídka definujeme operaci `==` takto „ručně".

```
instance Eq a => Eq (Tree a) where
  Leaf x == Leaf y         = x == y
  Node l1 r1 == Node l2 r2 = l1 == l2 && r1 == r2
  _ == _ = False
```

Tato instance deklaruje, že pro všechny typy `a` náleží typ `Tree a` do třídy `Eq`, platí-li, že typ `a` náleží do třídy `Eq`.

1.3.5 Monadický vstup/výstup

Jak už bylo zmíněno výše, vyhodnocování výrazů v čistě funkcionálním jazyce nemůže mít žádné vedlejší efekty a musí být referenčně transparentní. Co když ale potřebujeme vykonat nějakou vstupně/výstupní operaci, např. přečíst znak, který uživatel napsal na klávesnici, nebo zapsat soubor na disk? Takové „funkce“ by určitě vedlejší efekt měly a referenčně transparentní jistě také nejsou.

Většina jiných funkcionálních jazyků tento problém řeší tak, že obětuje *čistotu* a takové „nefunkcionální funkce“ povoluje, přičemž starosti s porušením referenční transparentnosti nechává na programátorovi. Tento přístup ovšem není možný v líně vyhodnocovaném jazyku, jelikož nemáme žádnou kontrolu nad tím, v jakém pořadí a jestli vůbec se funkce „zavolají“.

Typ `IO`

Haskell proto používá jinou techniku, která umožňuje zachovat funkcionální čistotu i líné vyhodnocování. Standardní knihovna poskytuje typ `IO a`, který reprezentuje *vstupně/výstupní operaci*, jejíž výsledek je typu `a`.

Ukažme si příklad několika operací ze standardní knihovny:

```
getChar :: IO Char
getLine :: IO String
putStrLn :: String -> IO ()

readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

`getChar` je vstupně/výstupní operace, která přečte jeden znak zadaný uživatelem a jejím výsledkem je tento znak, tedy typ `Char`. Obdobně `getLine` přečte celý řádek a její výsledek je typu `String`. Chceme-li naopak řádek vypsat, můžeme použít `putStrLn`, což je funkce, které předáme řetězec, jež si přejeme vypsat, a dostaneme vstupně/výstupní operaci, která tento řetězec vypíše. Jelikož tato operace nemá žádný smyslupný výsledek, vrací tzv. *nulový typ* `()`, který má jedinou hodnotu (rovněž se zapisuje `()`) a používá se jako „výplň“.[7]

Abychom přečetli soubor, musíme znát jeho umístění, proto je `readFile` funkce, které předáme cestu k souboru[8] a dostaneme vstupně/výstupní operaci, jejíž výsledek bude obsah souboru jako řetězec (`String`). Chceme-li zapsat data do souboru, použijeme funkci `writeFile`, které předáme dva argumenty – cestu k souboru a řetězec znaků, které si přejeme zapsat – a dostaneme IO operaci. Výsledek této operace je opět `()`.

Existuje pouze jedna možnost, jak vykonat operaci reprezentovanou typem `IO` – definovat ji jako proměnnou `main` v modulu `Main`.[9] Tato hodnota má typ `IO` α. Spustit program v Haskellu tedy vlastně znamená vyhodnotit operaci, která je přiřazena do `main`, a výsledek typu α zahodit.

Pokud bychom tedy po programu chtěli, ať spočte jednoduchý příklad a výsledek vypíše, mohli bychom nadefinovat `main` takto (funkce `++` slouží ke spojení dvou řetězců a funkce `show` převede číslo na řetězec):

```
main = putStrLn ("Jedna plus jedna je " ++ show (1+1))
```

[7] Jde o jistou obdobu typu `void` z jazyka C nebo `nil` či `null` z dynamických jazyků jako Ruby či JavaScript, která je ovšem typově bezpečná.

[8] Typ `FilePath` je synonym k typu `String`, který se ve standardní knihovně používá pro označení cest k souboru.

[9] „Proměnná“ samozřejmě neznamená, že se její hodnota nějakým způsobem mění; tento pojem se používá podobně jako v matematice. V imperativním jazyku bychom řekli „konstanta“.

Spojování operací pomocí >>= a >>

Co kdybychom ale chtěli provést více operací, například se zeptat uživatele na jméno a pak ho pozdravit? Potřebujeme nějakým způsobem „slepit“ dvě IO operace, a to takovým způsobem, aby druhá mohla využít výsledek první. K tomu slouží funkce `>>=`, někdy též nazývaná „bind“. Její typ je takovýto:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Jako první argument jí předáme první vstupně/výstupní operaci, kterou si přejeme vykonat a její výsledek dát jako vstupní hodnotu funkci předané jako druhý argument. Výsledek této funkce „zabalený“ v typu `IO` je výsledkem celé `>>=`.

S její pomocí můžeme uživatele pozdravit takto: [10]

```
main = getStrLn >>= (\name -> putStrLn ("Ahoj " ++ name))
```

Jak se ale uživatel doví, že po něm chceme, aby napsal svoje jméno? Nejprve mu musíme sdělit, že po něm požadujeme zadat jméno, a až poté jej přečíst a pozdravit. Opět můžeme využít `putStrLn` a `>>=`:

```
main =
  putStrLn "Kdo tam?" >>= (\_ ->
    getStrLn >>= (\name ->
      putStrLn ("Ahoj " ++ name)))
```

Výsledek z prvního `putStrLn` nás nazajímá (koneckonců to je pouze nulový typ `()`), takže jsme jej přiřadili do speciální „proměnné“ `_`, která funguje jako „černá díra“ – můžeme do ní přiřazovat nepotřebné údaje. Tento způsob zacházení s výsledky vstupně/výstupních operací je poměrně častý, proto je definována funkce `>>` s typem `IO a -> IO b -> IO b`, která je podobná `>>=`, s tím rozdílem, že první operaci sice provede, ale její výsledek zahodí a následně provede druhou operaci (nepředáváme jí tedy funkci), jejíž výsledek vrátí.

```
main =
  putStrLn "Kdo tam?" >>
    getStrLn >>= (\name ->
      putStrLn ("Ahoj " ++ name))
```

Tento kód ale není příliš přehledný a pokud bychom jej chtěli ještě dále rozšířit, mohli bychom se v něm brzy ztratit. Protože zřetězené používání `>>=` a `>>` je v Haskellu velmi časté, obsahuje jazyk tzv. `do`-syntaxi, která nám umožňuje zapisovat série takovýchto operací o něco úhledněji:

```
main = do
  putStrLn "Kdo tam?"
  name <- getStrLn
  putStrLn ("Ahoj " ++ name)
```

Monády

Typ `IO` je jen jedním z příkladů *monád*. Monády, původně koncept z teorie kategorií, je v Haskellu možno považovat za *výpočty* nebo *akce*, které je možno *skládat*, tedy přesměrovat *výstup* z jedné monády do druhé.

Do Haskellu jsou monády zahrnuty pomocí typové třídy `Monad` ze standardní knihovny jazyka. Tato třída je definována takto:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  m >> k  = m >>= \_ -> k

  return :: a -> m a
```

[10] Syntaxe `\x -> E` znamená anonymní (lambda) funkci s parametrem `x` a tělem `E`.

Funkce `>>=` (nazývaná „bind“) a `>>`, které jsme si výše ukázali při použití s typem `IO`, jsou tedy ve skutečnosti definované pro všechny monády. Funkce `return` pak pro libovolnou hodnotu vytvoří monádu, jejímž výstupem je tato hodnota.

Kromě typu `IO` jsou ze standardní knihovny monádami například i typy `[]` (seznam), `Maybe`, `Either e` či `(->) a` (funkce, jejímž argumentem je typ `a`). V podsekci 3.6.4 na straně 51 si představíme monádu `ST s`, kterou je možno považovat za „odlehčený“ typ `IO`, umožňující v čistém kódu využívat měnitelné datové struktury a efektivně implementovat algoritmy, které takovéto struktury vyžadují.

1.4 Dokumentace

Veškeré knihovny použité v této práci se, včetně dokumentace, nachází na serveru Hackage.[11] Pro vyhledávání v dokumentaci je možno použít vyhledávač Hoogle.[12] Tento vyhledávač umožňuje vyhledávat funkce, datové typy či třídy podle jména, ale dokáže najít funkci i podle přibližného *typu*, což z něj činí velmi užitečný nástroj.

[11] `http://hackage.haskell.org/packages/hackage.html`
[12] `http://www.haskell.org/hoogle/`

Kapitola 2

Krunimír: želví grafika

> Želvák Krunimír je velký myslitel. Zjistil, ze když si za krunýř přiváže trochu křídy, kreslí za sebou při svém plazení cestičku. I pojal plán nakreslit svoji podobiznu, samozřejmě včetně přesných detailů krunýře. Hned se dal pln nadšení do díla a práce mu šla pěkně od ... tlapy.
>
> „Co to děláš, dědečku," zeptal se jednou Krunimíra jeho vnuk Krunoslav. „Krešlím tady švou poďobižnu," odpověděl Krunimír. „Žačal jsem š ní, když tvůj tatík ještě nebyl na švětě, a ještě nemám ani krunýř," dodal smutně. „To už ji aši dokrešlit neštihnu..." Vnuk Krunoslav, znalec moderní techniky, mu však poradil: „Tak si nech napsat program, který ji nakreslí za Tebe." Protože se ale s tlapami a zobákem moc dobře neprogramuje, najali si želváci vás. [2]

Takto začíná zadání finální úlohy Soutěže v programování z roku 2010 [2] (přetisknuto na straně 61). Popisuje jednoduchý procedurální jazyk na generování želví grafiky, inspirovaný jazykem Logo, a úkolem je vytvořit interpret tohoto jazyka, jehož vstupem je text programu a výstupem vykreslený obrázek.

2.1 Popis jazyka

Krunimírův jazyk umožňuje vykreslovat obrázky principem želví grafiky. Na začátku se želva nachází uprostřed obrázku o velikosti 701×701 pixelů vyplněného bílou barvou a je otočená směrem nahoru. Uživatel má k dispozici několik primitivních kreslících procedur, kterými může želvu ovládat:

`forward(d)` Želva se posune vpřed o `d` jednotek (pixelů). Pokud je tloušťka pera kladná, zanechá za sebou čáru vedoucí z původní pozice do nové. Takovýto pohyb se považuje za jeden *tah*.

`right(a)`, `left(a)` Želva se otočí doprava, resp. doleva o `a` stupňů.

`pen(s)` Nastaví tloušťku pera na `s` pixelů. Je-li tloušťka pera nulová, želva nic nekreslí.

`color(r,g,b)` Nastaví barvu pera, `r`, `g` a `b` jsou jednotlivé složky barevného modelu RGB v rozsahu 0 až 255.

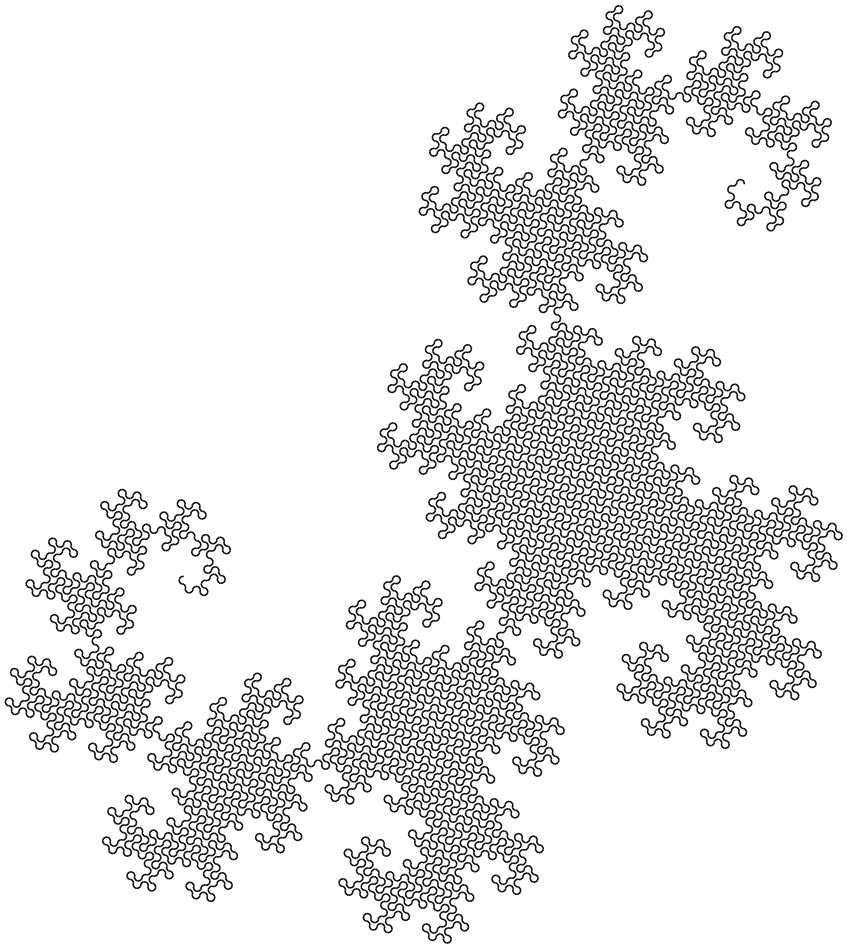
Jazyk dále umožňuje použít jednoduchou podmínku a cyklus:

`if(x) { ... }` Vykoná příkazy v těle podmínky právě když je `x` kladné.

`repeat(x) { ... }` Vykoná příkazy `x`-krát, je-li `x` kladné.

Uživatel může definovat vlastní procedury a volat je:

`define` *`procedura`*`(`*`p1`*`,`*`p2`*`,...) { ... }` Definuje proceduru *procedura*, která má libovolný počet parametrů (*p1*, *p2*, ...). Tyto parametry mohou být v těle procedury použity ve výrazech a nabývají hodnoty předané v místě volání.



Obrázek 2.1: Příklad obrázku vykresleného pomocí Krunimírova jazyka (Dračí křivka)

***procedura*`(`*arg1*`,`*arg2*`,...)` Zavolá proceduru *procedura* s argumenty (*arg1*, *arg2*, ...). Procedura musí být definována *před* svým voláním a může být rekurzivní.

Poslední a nejzajímavější struktura je rozdvojení:

`split { ... }` Vytvoří klon aktuální želvy, která vykoná příkazy v těle struktury `split`, přičemž původní želva pokračuje ve vykonávání dalších příkazů. Všechny želvy se pohybují paralelně, vždy všechny provedou jeden *tah*, poté druhý atd.

Jako argumenty při volání procedur lze používat výrazy vytvořené z celočíselných literálů, parametrů aktuální procedury, binárních operátorů `+`, `-`, `*` a `/` (dělení je celočíselné) a negace pomocí operátoru `-`. Ve výrazech je možno používat závorky `(` a `)`, priorita a asociativita operátorů je jako v matematice.

2.1.1 Příklady

Uvedeme si několik příkladů, které ilustrují využití veškerých příkazů Krunimírova jazyka. Vygenerované výstupy těchto příkladů jsou na obrázku 2.2.

Čtverec (2.2a)

Jednoduchý kód, který vykreslí čtverec.

```
pen(1)
forward(200) right(90)
forward(200) right(90)
forward(200) right(90)
forward(200) right(90)
```

Mřížka čtverců (2.2b)

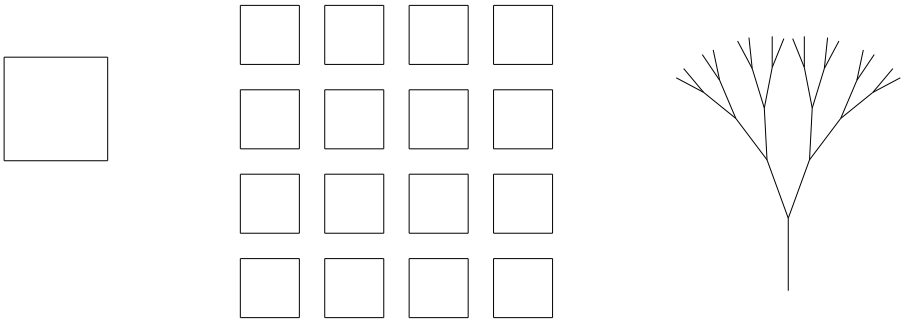
V této ukázce využijeme procedury, abychom kód rozdělili na menší a přehlednější části.

```
define square(side) {
  pen(1) repeat(4) { forward(side) right(90) } pen(0)
}

define row() {
  repeat(4) { square(114) forward(163) }
  forward(-652) right(90) forward(163) left(90)
}

forward(301) left(90) forward(301) right(180)
repeat(4) { row() }
```

Binární strom (2.2c)

Při vykreslování stromů prokáže svoji užitečnost příkaz `split`.

```
define tree(n) {
  if(n) {
    forward(40+20*n)
    split { left(5+3*n) tree(n-1) }
    right(5+3*n) tree(n-1)
  }
}

forward(-250) pen(1) tree(5)
```



(a) Čtverec (b) Mřížka čtverců (c) Binární strom

Obrázek 2.2: Výsledné obrázky z příkladů

2.2 Analýza

Problém si můžeme rozdělit na tři části:

1. *Syntaktická analýza* („parsování“) zpracuje vstupní řetězec na *abstraktní syntaktický strom*, který zachycuje strukturu programu ve formě, která je jednoduše zpracovatelná v dalších fázích.
2. Následuje *vyhodnocení*, kdy ze syntaktického stromu vypočteme výslednou stopu (ve vektorové podobě jako seznam úseček).
3. Poslední částí je *vykreslení*, které vykreslí vyhodnocenou stopu do obrázku. Budeme exportovat do rastrových obrázků formátu PNG a vektorových formátu SVG.

Pomocí tohoto jednoduchého rozdělení můžeme naše řešení rozvrhnout do sedmi modulů:

`Krunimir.Main` exportuje `main`, která slouží jako rozhraní s uživatelem.[1]

`Krunimir.Parser` exportuje funkci `parse`, která z textového zápisu programu vytvoří syntaktický strom (nebo syntaktickou chybu, pokud program není korektní).

`Krunimir.Ast` definuje datové typy, které reprezentují syntaktický strom.

`Krunimir.Evaluator` poskytuje funkci `eval`, která ze syntaktického stromu vypočte výslednou stopu.

`Krunimir.Trace` definuje datové typy a funkce spojené se stopou želvy.

`Krunimir.PngRenderer` exportuje funkci `renderPng`, která vykreslí stopu jako PNG obrázek.

`Krunimir.SvgRenderer` poskytuje funkci `renderSvg`, jenž uloží stopu ve vektorovém formátu SVG.

Nyní můžeme přejít na popis jednotlivých modulů, ze kterých je složen výsledný program.

2.3 `Krunimir.Main`

Začneme modulem `Krunimir.Main`, ve kterém definujeme hodnotu `main`, která tvoří „tělo“ programu.

```
module Krunimir.Main (main) where
```

[1] Podobně jako funkce `main()` v jazyku C

Potřebujeme několik funkcí z knihoven

```
import System.Environment (getArgs,getProgName)
import System.IO (stderr,hPutStrLn)
import System.Exit (exitFailure)
import System.FilePath (replaceExtension)
import Data.Time.Clock (getCurrentTime,diffUTCTime)
```

a také námi definované funkce z ostatních modulů

```
import Krunimir.Parser(parse)
import Krunimir.Evaluator(eval)
import Krunimir.PngRenderer(renderPng)
import Krunimir.SvgRenderer(renderSvg)
import Krunimir.Trace(prune)

main :: IO ()
main = do
```

Uložíme si čas na začátku běhu programu, bude se nám hodit, až budeme chtít zjistit, jak dlouho výpočet trval.

```
  startTime <- getCurrentTime
```

Nejprve se podíváme, jaké argumenty jsme dostali na příkazové řádce, a podle toho nastavíme proměnnou `inputFile` obsahující jméno vstupního souboru, a `steps`, což je `Just` *`početKroků`*, pokud máme zadaný počet kroků, nebo `Nothing`, pokud jej zadaný nemáme (takže předpokládáme, že uživatel chce vykreslit celý obrázek).[2] Pokud uživatel zadal jiný počet argumentů než jeden nebo dva, vypíšeme chybu a pomocí IO operace `exitFailure` ukončíme program s návratovým kódem, který signalizuje selhání.

```
  args <- getArgs
  (inputFile,steps) <- case args of
    [file] -> return (file,Nothing)
    [file,steps] -> return (file,Just $ read steps)
    _ -> do
      progname <- getProgName
      hPutStrLn stderr $ "Use: " ++ progname ++ " input-file [steps]"
      exitFailure
```

Nyní můžeme přečíst požadovaný soubor a jeho obsah předat funkci `parse` z modulu `Krunimir.Parser`. Pokud dostaneme chybu, zobrazíme ji na chybový výstup a program přerušíme.

```
  txt <- readFile inputFile
  ast <- case parse inputFile txt of
    Right ast -> return ast
    Left err -> do
      hPutStrLn stderr $ show err
      exitFailure
```

Úspěšně přečtený syntaktický strom můžeme předat funkci `eval` a dostaneme výslednou stopu v písku (`fullTrace`). Pokud uživatel zadal omezení počtu kroků, pomocí funkce `prune` stopu omezíme, pokud ne, necháme ji celou (`prunedTrace`).

```
  let fullTrace = eval ast
      prunedTrace = case steps of
        Nothing -> fullTrace
        Just count -> prune count fullTrace
```

Díky línému vyhodnocování se v případě, kdy je počet kroků omezen, vypočítá jen ta část stopy, která nás zajímá. Máme tedy zajištěno, že i když bude mít stopa desetitisíce kroků a

[2] V zadání je specifikováno, že nula zadaná jako počet kroků znamená vykreslit celý obrázek, a chování našeho programu je odlišné – nevykreslí nic.

uživatel bude chtít zobrazit jen prvních několik, nebudeme počítat celou stopu, ale jen zobrazenou část. Naše implementace dokonce umožňuje spouštět nekonečné programy, samozřejmě pouze pokud uživatel specifikuje počet kroků, jež si přeje vykonat.

```
  let outputPng = replaceExtension inputFile ".test.png"
      outputSvg = replaceExtension inputFile ".test.svg"
```

Jména výstupních souborů (jak PNG, tak SVG) odvodíme ze jména souboru vstupního, jen změníme příponu. Zbývá jen vykreslit

```
  renderPng prunedTrace outputPng
  renderSvg prunedTrace outputSvg
```

a vypsat řádek, který nás informuje o délce výpočtu:

```
  endTime <- getCurrentTime
  putStrLn $ show (diffUTCTime endTime startTime) ++ " : " ++ inputFile
```

2.4 `Krunimir.Ast`

V tomto modulu definujeme datové typy popisující syntaktický strom, které využijeme v modulech `Krunimir.Parser` a `Krunimir.Evaluator`.

```
module Krunimir.Ast where
```

2.4.1 Definice typů

Příkazy

Příkaz je reprezentován datovým typem `Stmt`, který obsahuje konstruktor pro každý z příkazů želvího jazyka.

```
data Stmt =
    ForwardStmt Expr
  | LeftStmt Expr
  | RightStmt Expr
  | PenStmt Expr
  | ColorStmt Expr Expr Expr
  | RepeatStmt Expr [Stmt]
  | IfStmt Expr [Stmt]
  | SplitStmt [Stmt]
  | CallStmt String [Expr]
  deriving Show
```

Výrazy

Reprezentace výrazů je podobně přímočará:

```
data Expr =
    LiteralExpr Integer
  | VariableExpr String
  | BinopExpr Op Expr Expr
  | NegateExpr Expr
  deriving Show

data Op = AddOp | SubOp | MulOp | DivOp
  deriving Show
```

Definice

Poslední strukturou je definice uživatelské procedury, pro kterou použijeme datový typ s pojmenovanými prvky (záznam neboli *record*).

```
data Define = Define
  { defineName :: String
  , defineParams :: [String]
  , defineStmts :: [Stmt]
  } deriving Show
```

Programy

Na nejvyšší úrovni v programu se mohou nacházet jak definice funkcí, tak příkazy, což odráží typ `TopStmt`. Těmto strukturám budeme říkat *top-příkazy*. Želví program je pak jen seznam těchto „top-příkazů“.

```
type Program = [TopStmt]

data TopStmt = TopDefine Define | TopStmt Stmt
  deriving Show
```

Identifikátor `TopStmt` může označovat dvě odlišné entity – *typový* konstruktor `TopStmt` a *datový* konstruktor `TopStmt` příslušející stejnojmennému typu, podle kontextu je ale vždy jasné, který z těchto dvou konstruktorů myslíme. V Haskellu se s takovýmito případy, kdy definujeme datový typ se stejnojmenným konstruktorem, setkáváme poměrně často.

2.4.2 Příklad

Využijeme program, který nakreslí binární strom a který jsme již jednou uvedli (jeho výstup je na obrázku 2.2c). Pro přehlednost ještě jednou zopakujeme jeho kód:

```
define tree(n) {
  if(n) {
    forward(40+20*n)
    split { left(5+3*n) tree(n-1) }
    right(5+3*n) tree(n-1)
  }
}

forward(-250) pen(1) tree(5)
```

Tento program je reprezentován pomocí výše uvedených typů následovně (text následující za `--` jsou komentáře):

```
[
  -- nejprve definice procedury 'tree'
  TopDefine (Define {
    defineName = "tree", -- jméno definované procedury
    defineParams = ["n"], -- seznam parametrů
    defineStmts = [ -- seznam příkazů v těle procedury
      IfStmt (VariableExpr "n") [
        -- příkaz 'forward(40+20*n)'
        ForwardStmt (BinopExpr AddOp
          (LiteralExpr 40)
          -- podvýraz '20*n'
          (BinopExpr MulOp (LiteralExpr 20) (VariableExpr "n"))),
        -- příkaz 'split { left(5+3*n) tree(n-1) }'
        SplitStmt [
          LeftStmt (BinopExpr AddOp
            (LiteralExpr 5)
            (BinopExpr MulOp (LiteralExpr 3) (VariableExpr "n"))),
```

```
          CallStmt "tree" [BinopExpr SubOp (VariableExpr "n") (LiteralExpr 1)]
        ],
        -- příkaz 'right(5+3*n)'
        RightStmt (BinopExpr AddOp
          (LiteralExpr 5)
          (BinopExpr MulOp (LiteralExpr 3) (VariableExpr "n"))),
        -- příkaz 'tree(n-1)'
        CallStmt "tree" [BinopExpr SubOp (VariableExpr "n") (LiteralExpr 1)]
      ]
    ]
  }),
  -- trojice příkazů následujících za definicí procedury
  TopStmt (ForwardStmt (NegateExpr (LiteralExpr 250))),
  TopStmt (PenStmt (LiteralExpr 1)),
  TopStmt (CallStmt "tree" [LiteralExpr 5])
]
```

2.5 `Krunimir.Parser`

Pro syntaktickou analýzu („parsování“) použijeme knihovnu `parsec` [13]. Jedná se o jeden z nejpoužívanějších nástrojů na tvorbu parserů v Haskellu.

```
module Krunimir.Parser (Krunimir.Parser.parse) where
import Text.Parsec
import Control.Applicative ((<$>), (<$), (<*), (*>), (<*>))
import Krunimir.Ast
```

Narozdíl od generátorů jako GNU Bison [8], které vyžadují speciální soubor s definicí gramatiky a strojově jej překládají do cílového jazyka, se `parsec` používá v normálním kódu Haskellu a parsery se konstruují pomocí *vysokoúrovňových kombinátorů*, které umožňují kombinovat malé parsery do větších celků.

Jako formální základ pro náš parser použijeme gramatiku PEG [6]. Výhodou PEG gramatik je jejich snadná implementace, jelikož popisují *rozpoznávání* jazyka, narozdíl od tradičních bezkontextových gramatik, které byly vytvořeny pro popis lidských jazyků a definují jejich *generování*.

PEG gramatika pro Krunimírův jazyk se nachází na straně 27.

2.5.1 Základní definice

Parsery v knihovně `parsec` mají typ `Parsec s u a`, kde:

- `s` je typ vstupu, v našem případě `String`.
- `u` je typ uživatelského stavu, tj. data, která uživatel (programátor parserů) může ukládat během parsování. My tuto vlastnost využívat nebudeme, proto použijeme „prázdný“ typ `()`.
- `a` je výsledek parseru, tedy typ který parser vrátí. Naše parsery budou vracet více typů, např. příkaz (`Stmt`) nebo číslo (`Integer`).

Abychom nemuseli neustále opakovat `Parsec String () ...`, nadefinujeme *typový synonym*, který využijeme i při prezentaci jednotlivých kombinátorů.

```
type Parser a = Parsec String () a
```

2.5.2 Představení základních kombinátorů

Některé kombinátory definuje přímo `parsec`:

`char :: Char -> Parser Char`
`char c` vytvoří parser, který akceptuje znak `c` a v případě úspěchu jej vrátí.

`string :: [Char] -> Parser [Char]`
`string cs` je parser, jenž akceptuje sekvenci znaků (řetězec) `cs`.

`(<|>) :: Parser a -> Parser a -> Parser a`
`p <|> q` představuje volbu – nejprve aplikuje parser `p`, a pokud selže, *aniž zkonzumoval nějaký vstup*, aplikuje parser `q`.

`(<?>) :: Parser a -> String -> Parser a`
`p <?> msg` aplikuje parser `p`, a pokud selže, *aniž zkonzumoval část vstupu*, nahradí část `"Expected ..."` chybové zprávy řetězcem `msg`.

`try :: Parser a -> Parser a`
`try p` funguje jako `p`, ale s tím rozdílem, že pokud `p` selže, předstírá, že nic nezkonzumoval. Použijeme ho nejčastěji ve spojení s `<|>`.

`many :: Parser a -> Parser [a]`
`many p` aplikuje parser `p` *nula* či vícekrát a vrátí seznam výsledků z `p` (což znamená, že pokud `p` poprvé skončí neúspěchem, `many` vrátí prázdný seznam).

`many1 :: Parser a -> Parser [a]`
`many1 p` funguje obdobně jako `many p`, s tím rozdílem, že `p` aplikuje *vždy alespoň jednou* (pokud `p` napoprvé selže, skončí neúspěchem i `many1`).

`sepBy :: Parser a -> Parser sep -> Parser [a]`
`p` ‘`sepBy`‘ `s` zparsuje *nula* či více výskytů `p` oddělených `s`.[3] Obdobně jako u `many` existuje varianta `sepBy1`, která aplikuje `p` alespoň jednou.

Každý parser je samozřejmě *monáda*, proto můžeme použít základní monadické operace:

`(>>=) :: Parser a -> (a -> Parser b) -> Parser b`
`p >>= f` aplikuje parser `p` a jeho výsledek předá funkci `f`.

`(>>) :: Parser a -> Parser b -> Parser a`
`p >> q` nejprve aplikuje parser `p`, jeho výsledek zahodí a aplikuje `q`.

`return :: a -> Parser a`
`return x` vytvoří parser, který vždy uspěje a vrátí `x`.

Každá monáda je *aplikativní funktor*, tudíž můžeme použít i následující funkce:

`(<$>) :: (a -> b) -> Parser a -> Parser b`
`f <$> p` aplikuje parser `p` a v případě úspěchu předá jeho výsledek funkci `f`, jejíž výstup se stane výsledkem `<$>`.

`(<*>) :: Parser (a -> b) -> Parser a -> Parser b`
`p <*> q` nejprve aplikuje `p`, poté `q` a výsledek z `q` předá funkci získané z `p`, jejíž výstup se stane výsledkem `<*>`.

`(<$) :: a -> Parser b -> Parser a`
`x <$ p` aplikuje parser `p`, ale jeho výsledek zahodí a namísto toho vrátí `x`.

[3] Zápis pomocí ‘ je pouze syntaktický cukr, kterým můžeme zapsat infixově volání jakékoli funkce; jinak je ekvivalentí klasickému `sepBy p s`.

`(<*) :: Parser a -> Parser b -> Parser a`
: `p <* q` aplikuje nejprve parser `p`, poté parser `q`, jehož výsledek zahodí a vrátí výsledek `p`.

`(*>) :: Parser a -> Parser b -> Parser b`
: `p *> q` aplikuje parser `p`, poté `q`, jehož výsledek vrátí. Tato funkce je ekvivalentní s `>>`, ale použití spolu s `<*` dáme přednost této variantě.

2.5.3 Funkce `parse`

Funkce `parse` představuje „rozhraní“ modulu `Krunimir.Parser`. Vstupem je jméno parsovaného souboru (použije se v případných chybových hláškách) a samotný text programu. Výstupem je buď chyba (`ParseError`) nebo želví program (`Program`).

Využijeme stejně pojmenovanou funkci, kterou nám `parsec` nabízí, a předáme jí nejprve parser celého programu (`program`) a pak oba zbývající argumenty.

```
parse :: String -> String -> Either ParseError Program
parse filename txt =
  Text.Parsec.parse program filename txt
```

2.5.4 Programy

Na začátku programu může být libovolné množství prázdných znaků, následuje nula a více top-příkazů a konec souboru.

```
program :: Parser Program
program = spaces *> many topStmt <* eof
```

Operátory `*>` a `<*` mají stejnou prioritu a jsou asociativní zleva, což znamená že tento kód je ekvivalentní `(spaces *> many topStmt) <* eof`. Nejprve se tedy aplikuje `spaces`,[4] jehož výsledek se zahodí, poté `many topStmt`, kterým získáme seznam top-příkazů, a nakonec `eof`. Pokud `eof` uspěje, dostaneme výsledek z `many topStmt`, pokud ne, parser vrátí chybu.

Top-příkazy

Top-příkaz je buď definice procedury (parser `define`) nebo příkaz (parser `stmt`), ze kterých pomocí příslušných datových konstruktorů (`TopDefine`, resp. `TopStmt`) vytvoříme typ `TopStmt`.

```
topStmt :: Parser TopStmt
topStmt =
  TopDefine <$> try define <|>
  TopStmt <$> stmt
```

Definice procedur

Definice procedur v Krunimírově jazyku začínají klíčovým slovem `define` následovaným jménem procedury, za kterým je v závorkách nula a více parametrů. Tělo procedury je uzavřeno ve složených závorkách.

```
define :: Parser Define
define = do
  string "define" >> skipMany1 space
  name <- identifier
  params <- parens $ identifier `sepBy` comma
  stmts <- braces $ many stmt
  return $ Define name params stmts
```

Použili jsme pomocné funkce `parens` a `braces`, které slouží k „obalování závorkami“ a které si nadefinujeme později.

[4] Parser `spaces` definuje samotná knihovna `parsec`, má typ `Parser ()` a zahodí nula a více prázdných znaků.

2.5.5 Příkazy

K parsování *příkazů* slouží `stmt`, která jen aplikuje další pomocné parsery a pojmenuje případnou chybu.

```
stmt :: Parser Stmt
stmt =
  try repeatStmt <|>
  try ifStmt <|>
  try splitStmt <|>
  try procStmt <?>
  "statement"
```

Volání procedur

Začneme syntaxí užitou při volání procedur. Jak zabudované primitivní (`forward`, `color`...), tak programátorem definované procedury se volají syntaktivky stejně, proto je musíme rozlišit podle jména a podle toho vytvořit příslušný uzel syntaktického stromu.

Volání začíná jménem volané procedury a následuje v závorkách seznam argumentů, který může být prázdný, závorky ale vynechat nelze.

```
procStmt :: Parser Stmt
procStmt = do
  name <- identifier
  args <- parens $ expr `sepBy` comma
  case name of
    "forward" -> primitive ForwardStmt "forward" args
    "left"    -> primitive LeftStmt "left" args
    "right"   -> primitive RightStmt "right" args
    "pen"     -> primitive PenStmt "pen" args
    "color"   -> case args of
        [r,g,b] -> return $ ColorStmt r g b
        _       -> parserFail $
            "color takes 3 arguments, got " ++ show (length args)
    _ -> return $ CallStmt name args
  where
    primitive con _ [arg] = return $ con arg
    primitive _ name args = parserFail $
        name ++ " takes 1 argument, got " ++ show (length args)
```

Využili jsme parsery `identifier` a `parens`, které si nadefinujeme později, a pomocnou funkci `primitive`, kterou si ušetříme opakování při zpracování příkazů `forward`, `left`, `right` a `pen`, které všechny vyžadují jeden argument.

Příkazy `if` a `repeat`

Syntaxe pro `if` a `repeat` je velmi podobná – nejprve klíčové slovo, poté v závorkách výraz a nakonec seznam příkazů ve složených závorkách.

```
repeatStmt :: Parser Stmt
repeatStmt = do
  keyword "repeat"
  times <- parens expr
  stmts <- braces $ many stmt
  return $ RepeatStmt times stmts

ifStmt :: Parser Stmt
ifStmt = do
  keyword "if"
  cond <- parens expr
  stmts <- braces $ many stmt
  return $ IfStmt cond stmts
```

Pomocný parser `keyword` nadefinujeme později; kdybychom místo něj použili jednoduše `string`, například `string "if"`, a programátor by nadefinoval třeba proceduru `iffy` a pokusil by se ji zavolat (`iffy(42)`), parser by přečetl pouze `"if"`, domníval by se, že jde o příkaz `if`, a pak nevěděl co s `"fy(42)"`, protože očekává otevírací závorku. Naproti tomu `keyword "if"` se aplikuje pouze na sekvenci znaků `"if"`, za kterou *nenásleduje písmeno*, čímž zajistíme, že jsme opravdu narazili na klíčové slovo `if`.

Konstrukce `split`

Syntaxe pro `split` je přímočará, za klíčovým slovem následují rovnou složené závorky se seznamem příkazů.

```
splitStmt :: Parser Stmt
splitStmt = do
  keyword "split"
  stmts <- braces $ many stmt
  return $ SplitStmt stmts
```

2.5.6 Výrazy

Parsování *výrazů* je o něco složitější, jelikož se musíme vypořádat s prioritami a asociativitami jednotlivých operátorů.

Gramatiku matematických výrazů můžeme vyjádřit v bezkontextové gramatice takto:

⟨*expr*⟩ ::= ⟨*add-expr*⟩

⟨*add-expr*⟩ ::= ⟨*add-expr*⟩ `+` ⟨*neg-expr*⟩
| ⟨*add-expr*⟩ `-` ⟨*neg-expr*⟩
| ⟨*neg-expr*⟩

⟨*neg-expr*⟩ ::= `-` ⟨*mul-expr*⟩
| ⟨*mul-expr*⟩

⟨*mul-expr*⟩ ::= ⟨*mul-expr*⟩ `*` ⟨*a-expr*⟩
| ⟨*mul-expr*⟩ `/` ⟨*a-expr*⟩
| ⟨*a-expr*⟩

⟨*a-expr*⟩ ::= variable
| literal
| `(` ⟨*expr*⟩ `)`

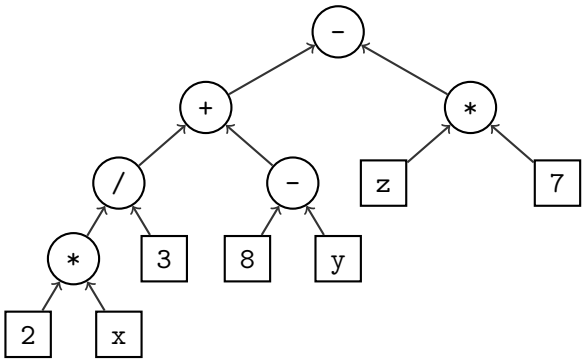
Problém je, že pravidla pro sčítání/odčítání a násobení/dělení jsou rekurzivní zleva, takže je nelze v této podobě zpracovávat pomocí gramatiky PEG. Proto je musíme přeformulovat (ošetření mezer jsme pro přehlednost vynechali):

```
expr        <- add-expr
add-expr    <- neg-expr (add-op neg-expr)*
neg-expr    <- "-"? mul-expr
mul-expr    <- a-expr (mul-op a-expr)*
a-expr      <- variable / literal / "(" expr ")" "blaah"

add-op      <- "+" / "-"
mul-op      <- "*" / "/"
```

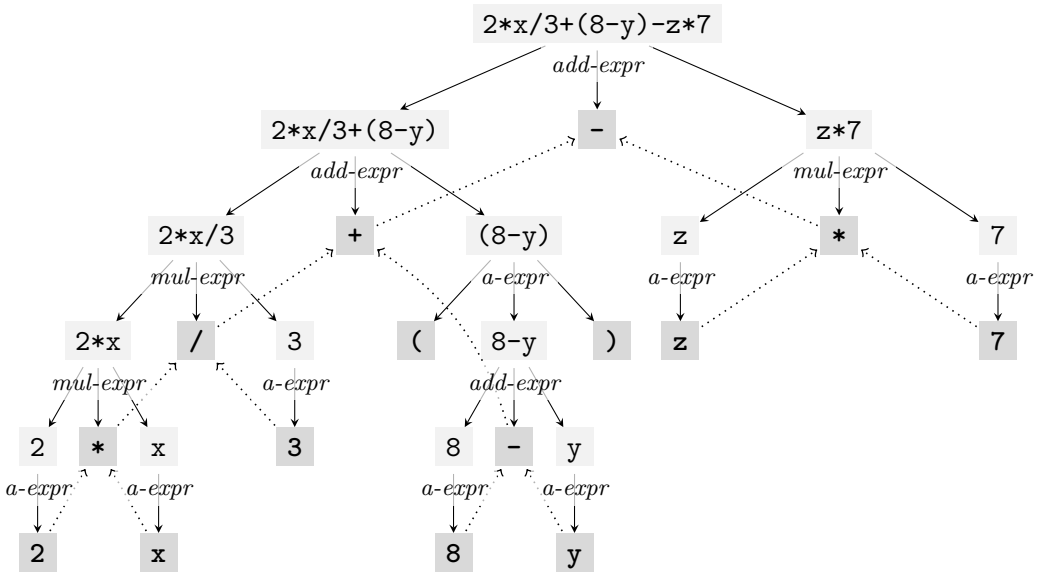
Tuto PEG gramatiku již můžeme použít, ale struktura gramatiky již neodpovídá struktuře syntaktického stromu. `parsec` naštěstí obsahuje pomocné funkce, které nám úkol značně ulehčí.

Použijeme funkci `chainl1`, jejíž typ je `chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a`. `chainl p op` zparsuje jeden a více výskytů `p` oddělených `op`. Výsledky z `p` postupně odleva „spojí“ pomocí funkcí vrácených z `op`.

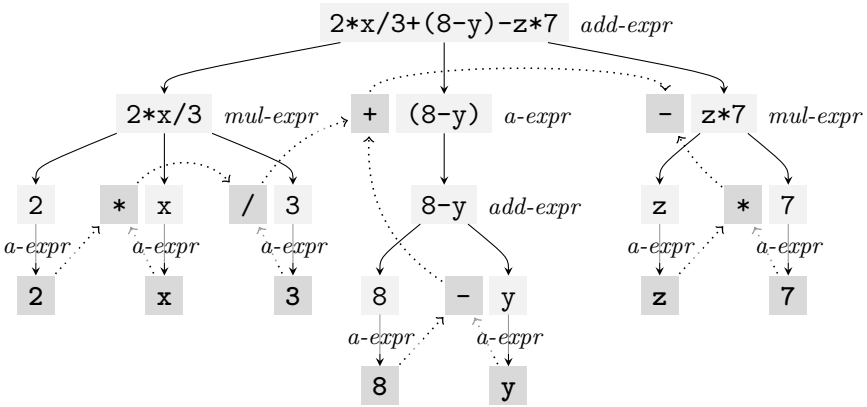
Obrázek 2.3: Příklady parsování výrazu `2*x/3+(8-y)-z*7`



(a) Syntaktický strom, reprezentující tento výraz. Binární operace (zaznačené kolečky) mají vždy dva operandy, které jsou mohou být číslo, proměnná nebo další binární operace.



(b) Ilustrace způsobu, jakým tento výraz zpracuje bezkontextová gramatika. Tečkovanými čarami je zaznačena struktura výsledného syntaktického stromu, která přímo vyplývá z postupného dělení výrazu na menší části.



(c) Tentýž výraz zparsovaný gramatikou PEG se znázorněným výsledným syntaktickým stromem. Postup parsování již jeho struktuře přímo nedpovídá.

```
expr :: Parser Expr
expr = addExpr <?> "expression"

addExpr,mulExpr,negExpr,aExpr,varExpr,litExpr :: Parser Expr

addExpr = chainl1 mulExpr (addOp <* spaces)
mulExpr = chainl1 negExpr (mulOp <* spaces)

addOp,mulOp :: Parser (Expr -> Expr -> Expr)
addOp =
  BinopExpr AddOp <$ char '+' <|>
  BinopExpr SubOp <$ char '-'

mulOp =
  BinopExpr MulOp <$ char '*' <|>
  BinopExpr DivOp <$ char '/'

negExpr = (negOp <* spaces) <*> aExpr <|> aExpr

negOp :: Parser (Expr -> Expr)
negOp = NegateExpr <$ char '-'

aExpr = litExpr <|> varExpr <|> parens expr
varExpr = VariableExpr <$> identifier
litExpr = LiteralExpr <$> integer
```

2.5.7 Pomocné parsery

Nakonec si nadefinujeme drobné parsery, které jsme použili. Každý z nich zkonzumuje i všechny prázdné znaky, které se za ním nachází, takže se s jejich ošetřením nemusíme zabývat ve „vyšších“ parserech.

V identifikátorech povolíme i velká písmena a číslice, pokud se nenachází na začátku.

```
integer :: Parser Integer
integer = read <$> many1 digit <* spaces
identifier :: Parser String
identifier = (:) <$> letter <*> many alphaNum <* spaces

keyword :: String -> Parser ()
keyword s = string s >> notFollowedBy alphaNum >> spaces

lparen,rparen,lbrace,rbrace,comma :: Parser ()
lparen = char '(' >> spaces
rparen = char ')' >> spaces
lbrace = char '{' >> spaces
rbrace = char '}' >> spaces
comma  = char ',' >> spaces

parens,braces :: Parser a -> Parser a
parens = between lparen rparen
braces = between lbrace rbrace
```

2.5.8 PEG gramatika

Na závěr uvedeme kompletní „referenční“ PEG gramatiku Krunimírova jazyka.

```
program      <- space* top-stmt* eof
top-stmt     <- define / stmt

define       <- "define" space+ identifier
                lparen (identifier (comma identifier)*)? rparen
                lbrace stmt* rbrace

stmt         <- repeat-stmt / if-stmt / split-stmt / proc-stmt
repeat-stmt  <- "repeat" lparen expr rparen lbrace stmt* rbrace
if-stmt      <- "if" space* lparen expr rparen lbrace stmt* rbrace
split-stmt   <- "split" space* lbrace stmt* rbrace
proc-stmt    <- "forward" space* lparen expr rparen
              / "left" space* lparen expr rparen
              / "right" space* lparen expr rparen
              / "pen" space* lparen expr rparen
              / "color" space* lparen expr comma expr comma expr rparen
              / identifier space* lparen (expr (comma expr)*)? rparen

expr         <- add-expr
add-expr     <- mul-expr (add-op space* mul-expr)*
mul-expr     <- neg-expr (mul-op space* neg-expr)*
neg-expr     <- (neg-op space*)? a-expr

add-op       <- "+" / "-"
mul-op       <- "*" / "/"
neg-op       <- "-"

a-expr       <- lit-expr / var-expr / lparen expr rparen
lit-expr     <- integer
var-expr     <- identifier

integer      <- digit+ space*
identifier   <- letter alpha-num space*

lparen       <- "(" space*
rparen       <- ")" space*
lbrace       <- "{" space*
rbrace       <- "}" space*
comma        <- "," space*

digit        <- [0-9]
letter       <- [a-zA-Z]
alpha-num    <- [a-zA-Z0-9]
space        <- [ \t\r\n\v\f]
eof          <- !.
```

2.6 Krunimir.Trace

Než představíme vyhodnocování programu reprezentovaného syntaktickým stromem, musíme ukázat modul `Krunimir.Trace`, který poskytuje datové typy pro práci se stopami složenými z čar, které za sebou zanechává želva pochodující po písku. Tyto stopy jsou výstupem funkce `Krunimir.Evaluator.eval`.

```
module Krunimir.Trace
( Trace(..)
, Segment(..)
, prune
, traceToSegss
) where
```

2.6.1 Typy

Nejdůležitějším typem je `Trace`, reprezentující stopu želvy. `Trace` má tři konstruktory:

`EmptyTrace` je prázdná stopa, tedy nic.

`SplitTrace` reprezentuje rozdělení stopy na dvě v místě příkazu `split`.

`SegmentTrace` je stopa tvořená úsečkou, za kterou následuje další stopa.

```
data Trace
  = EmptyTrace
  | SplitTrace Trace Trace
  | SegmentTrace Segment Trace
  deriving Show
```

Typ `Segment` představuje úsečku mezi dvěma body, která má barvu a tloušťku.

```
data Segment = Segment Point Point Color Int
  deriving Show

type Point = (Float,Float)
type Color = (Int,Int,Int)
```

2.6.2 Funkce

V ostatních modulech budeme potřebovat pomocné funkce `prune` a `traceToSegss`.

Funkce `prune`

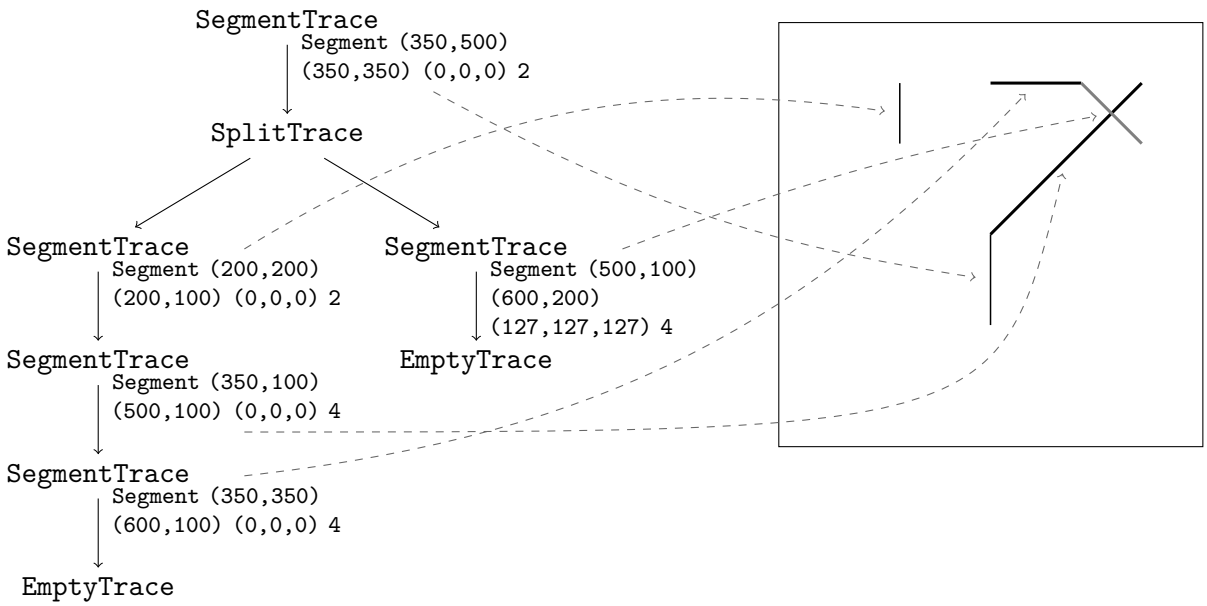
Funkce `prune` omezí počet *tahů*, které stopa zahrnuje.

```
prune :: Integer -> Trace -> Trace
prune n img
  | n <= 0 = EmptyTrace
  | otherwise = case img of
    EmptyTrace -> EmptyTrace
    SplitTrace l r -> SplitTrace (prune n l) (prune n r)
    SegmentTrace seg x -> SegmentTrace seg $ prune (n-1) x
```

Funkce `traceToSegss`

Funkce `traceToSegss` transformuje stopu do seznamu seznamů segmentů. „Vnější“ seznam reprezentuje jednotlivé segmenty jednoho tahu. Tato funkce se využívá při vykreslování a seřazení jednotlivých segmentů podle tahů zajišťuje korektní překrytí čar.[5]

[5] Kdybychom se rozhodli vzájemné překrytí čar zanedbat, celý program by byl *výrazně* jednodušší.




```
SegmentTrace (Segment (350,500) (350,350) (0,0,0) 2)
  (SplitTrace
    (SegmentTrace (Segment (200,200) (200,100) (0,0,0) 2)
      (SegmentTrace (Segment (350,100) (500,100) (0,0,0) 4)
        (SegmentTrace (Segment (500,100) (600,200) (127,127,127) 4)
          EmptyTrace)))
    (SegmentTrace (Segment (350,350) (600,100) (0,0,0) 4)
      EmptyTrace))
```

Obrázek 2.4: Grafické znázornění stopy. Obrázek nalevo odpovídá vykreslené stopě, šipky od jednotlivých segmentů ukazují na příslušné úsečky na obrázku. Důležitý je způsob, jakým se čáry překrývají – segment, jenž je blíž kořeni stromu, je překryt segmentem nacházejícím se ve stromu „hlouběji“.

```
traceToSegss :: Trace -> [[Segment]]
traceToSegss EmptyTrace = []
traceToSegss (SegmentTrace seg img) = [seg]:traceToSegss img
traceToSegss (SplitTrace limg rimg) = zipSegs (traceToSegss limg) (traceToSegss rimg)
  where
  zipSegs :: [[Segment]] -> [[Segment]] -> [[Segment]]
  zipSegs [] [] = []
  zipSegs lss [] = lss
  zipSegs [] rss = rss
  zipSegs (ls:lss) (rs:rss) = (ls ++ rs):zipSegs lss rss
```

V případě `SplitTrace` nejprve vyhodnotíme seznamy seznamů segmentů pro každou stranu zvlášť a pak je pomocnou funkcí `zipSegs` „slijeme“ dohromady.

2.7 `Krunimir.Evaluator`

Nyní se dostáváme k jádru problému, totiž samotnému *vyhodnocování* Krunimírova programu, implementovanému funkcí `eval`. Vstupem této funkce je syntaktický strom v podobě typu `Program` (což je jen synonym pro `[TopStmt]`), výstupem stopa želvy jako typ `Trace`.

```
module Krunimir.Evaluator (eval) where
import Krunimir.Trace
import Krunimir.Ast

import qualified Data.Map as M
import Data.List (genericReplicate)
```

2.7.1 Pomocné typy

Definujeme si datový typ `Turtle`, který zahrnuje celý stav želvy – její pozici, natočení a barvu a tloušťku pera.[6]

```
data Turtle = Turtle
  { getPos :: (Float,Float)
  , getAngle :: Integer
  , getColor :: (Int,Int,Int)
  , getPen :: Int
  }
```

V průběhu vyhodnocování musíme mít uloženy informace o definovaných procedurách a aktuálních hodnotách argumentů (proměnných). K tomu slouží typ `Env`. Definujeme si synonyma `ProcMap` a `VarMap`, což jsou *mapy* mapující jména procedur na jejich definice a jména proměnných na hodnoty.

```
data Env = Env ProcMap VarMap
type ProcMap = M.Map String Define
type VarMap = M.Map String Integer
```

2.7.2 Představení `DiffTrace`

Nyní se musíme rozhodnout, jak přesně budeme vyhodnocování stopy v syntaktickém stromu implementovat. Určitě bude vhodné vytvořit funkci na vyhodnocení jednoho příkazu a jednoho výrazu.

Jaké informace potřebujeme k tomu, abychom mohli spočítat hodnotu výrazu? Vzhledem k tomu, že se ve výrazu můžou vyskytovat argumenty aktuální procedury, budeme potřebovat `Env`,

[6] Typ `Float` jsme použili namísto obvyklého `Double` pro ušetření paměti, měl by totiž vyžadovat pouze 4 bajty namísto 8. Jelikož Krunimírovy programy pracující s desetitisíci želvami nejsou žádnou výjimkou, ušetřené bajty se na výkonu programu pozitivně projeví.

z něhož získáme hodnoty aktuálních proměnných (argumentů). Z toho nám plyne typ pro funkci `evalExpr`:

```
evalExpr :: Env -> Expr -> Integer
```

Dále bude třeba vytvořit funkci, která vyhodnotí příkaz. Argumenty této funkce bude učitě znovu `Env` (potřebujeme znát, jaké procedury existují) a `Turtle` (musíme vědět, jaký je aktuální stav želvy).

Jakou hodnotu bychom měli vrátit? Každý příkaz změní stav želvy, proto bychom novou želvu měli vrátit jako část výsledku. Hlavní je ale to, jestli příkaz nezmění stopu, kterou za sebou želva zanechá. Jakým způsobem ale tuto změnu reprezentovat? Nemůžeme použít přímo typ `Trace`, jelikož ten reprezentuje celou želvinu trasu, kdežto my spočteme jen její začátek, neboť za tímto jedním příkazem mohou následovat další, které trasu rovněž prodlouží.

Nejlepší bude, když vrátíme *funkci*, která jako argument dostane `Trace` získaný z následujících příkazů a vrátí novou `Trace`.

Tím získáváme typ:

```
evalStmt :: Env -> Stmt -> Turtle -> (Turtle,Trace -> Trace)
```

S funkcemi typu `Trace -> Trace` budeme pracovat často, proto si vytvoříme *nový typ*.

```
newtype DiffTrace = DiffTrace { applyDT :: Trace -> Trace }
```

Jaký typ bude mít funkce `applyDT`? Z hodnoty typu `DiffTrace` extrahuje hodnotu typu `Trace -> Trace`, tudíž její typ bude `DiffTrace -> (Trace -> Trace)`, neboli `DiffTrace -> Trace -> Trace`. To znamená, že na `applyDT` můžeme nahlížet jako na funkci se dvěmi argumenty, která *aplikuje* změnu – `DiffTrace` – na `Trace`, čímž získáme novou `Trace`.

Nadefinujeme si také operaci `identity`, tj. žádná změna se neprovede.

```
identityDT :: DiffTrace
identityDT = DiffTrace { applyDT = id }
```

S tímto novým typem, který reprezentuje *rozdíl* nebo *změnu* stopy `Trace`, bude typ funkce `evalStmt` vypadat takto:

```
evalStmt :: Env -> Stmt -> Turtle -> (Turtle,DiffTrace)
```

Tuto deklaraci typu můžeme chápat takto: „`evalStmt` je funkce vyžadující mapu procedur a proměnných uložených v typu `Env`, dále příkaz k vyhodnocení a stav želvy; vrátí změněný stav želvy a *změnu*, kterou tento příkaz vyvolá na stopě želvy.“

I když toto typové kung-fu může vypadat na první pohled zbytečně komplikovaně a složitě, opak je pravdou – umožní nám vyhodnocování příkazů implementovat velmi elegantně a jednoduše.

2.7.3 Funkce `eval`

Nejprve představíme funkce `eval`, která vyhodnotí celý program:

```
eval :: Program -> Trace
eval prog = applyDT (snd $ evalStmts env stmts startTurtle) EmptyTrace
  where

  (defs,stmts) = foldl go ([],[]) (reverse prog)
    where go (ds,ss) topstmt = case topstmt of
            TopDefine def -> (def:ds,ss)
            TopStmt stmt -> (ds,stmt:ss)

  env = Env procMap varMap
  procMap = M.fromList [(defineName def,def) | def <- defs]
  varMap = M.empty

  startTurtle = Turtle
    { getPos = (350.5,350.5)
    , getAngle = 0
```

```
    , getColor = (0,0,0)
    , getPen = 0
    }
```

Nejdříve si rozeberme klauzuli `where`. Nejprve průchodem seznamu `prog` funkcí `foldl` získáme seznam definic `defs` a seznam příkazů `stmts`, které extrahujeme z top-příkazů.

Ze seznamu definic vytvoříme mapu procedur `procMap`. Na nejvyšší úrovni se v programu nenachází žádné proměnné, proto je mapa proměnných prázdná. `startTurtle` je počáteční stav želvy – nachází se uprostřed obrázku s vypnutým černým perem a je otočená směrem nahoru.

V samotném těle funkce `eval` nejprve vyhodnotíme pomocí funkce `evalStmts` seznam příkazů, čímž získáme dvojici `(Turtle,DiffTrace)`. První prvek, želva, nás nezajímá, ale funkcí `snd` získáme hodnotu `DiffTrace`, která reprezentuje změnu, jenž program vykoná na celkové stopě želvy. Tuto změnu aplikujeme na prázdnou stopu, takže získáme kýženou hodnotu `Trace`.

2.7.4 Vyhodnocování příkazů

Funkce `evalStmts`, která vyhodnotí seznam příkazů, vždy vyhodnotí jeden příkaz, poté seznam následujících příkazů a vrátí výslednou želvu a složený `DiffTrace`.

```
evalStmts :: Env -> [Stmt] -> Turtle -> (Turtle,DiffTrace)
evalStmts _ [] turtle = (turtle,identityDT)
evalStmts env (stmt:stmts) turtle =
  let (turtle',dt) = evalStmt env stmt turtle
      (turtle'',dt') = evalStmts env stmts turtle'
  in (turtle'',DiffTrace { applyDT = applyDT dt . applyDT dt' })
```

V `evalStmt` použijeme velký `case`, v němž patřičně reagujeme na každý druh příkazu.

```
evalStmt :: Env -> Stmt -> Turtle -> (Turtle,DiffTrace)
evalStmt env stmt = case stmt of
  ForwardStmt e -> forward (ee e)
  LeftStmt e    -> rotate (negate $ ee e)
  RightStmt e   -> rotate (ee e)
  PenStmt e     -> pen (ee e)
  ColorStmt r g b -> color (ee r) (ee g) (ee b)
  RepeatStmt e stmts -> evalStmts env $ concat $ genericReplicate (ee e) stmts
  IfStmt e stmts -> if ee e > 0 then evalStmts env stmts else noop
  SplitStmt stmts -> split $ evalStmts env stmts
  CallStmt name args -> let
      def = lookupDef env name
      binds = zip (defineParams def) (map ee args)
      newenv = makeEnv env binds
    in evalStmts newenv (defineStmts def)
  where ee = evalExpr env
```

Primitivní operace s želvou jsme ošetřili pomocnými funkcemi, které implementujeme později. Podmínka a cyklus v podobě `RepeatStmt` a `IfStmt` jsou implementovány pomocí `evalStmts`, stejně jako volání procedury, kde ale musíme vytvořit novou mapu proměnných z předaných argumentů.

S výhodou jsme využili *currying*u – ač `evalStmt` vyžaduje tři argumenty, na levé straně rovnice jsme uvedli pouze první dva (`env` a `stmt`), tudíž na pravé straně musí být funkce typu `Turtle -> (Turtle,DiffTrace)` (podle typové deklarace funkce `evalStmt`). Tímto se zbavíme neustálého opakování a předávání argumentu `Turtle` jednotlivým specializovaným funkcím.

2.7.5 Jednotlivé příkazy

Nyní se dostáváme k implementaci jednotlivých funkcí použitých v `evalStmt`.

Prázdná operace

Funkci `noop` jsme využili v příkazu `IfStmt` na ošetření situace, když podmínka neplatí, a její chování je jednoduché – nedělá nic, takže vrátí nezměněnou želvu a *identitu*.

```
noop :: Turtle -> (Turtle,DiffTrace)
noop turtle = (turtle,identityDT)
```

Posun vpřed

Při pohybu vpřed musíme želvu posunout na nové místo a zkontrolovat, jestli za sebou nezanechala čáru. Pokud ano, vrátíme `DiffImage`, který tuto změnu zachycuje, pokud ne, dostaneme identitu.

```
forward :: Integer -> Turtle -> (Turtle,DiffTrace)
forward len turtle = (turtle',DiffTrace diff) where
  (x,y) = getPos turtle
  ang   = getAngle turtle
  p     = getPen turtle
  x'    = x + sinDeg ang * fromIntegral len
  y'    = y - cosDeg ang * fromIntegral len
  turtle' = turtle { getPos = (x',y') }
  segment = Segment (x,y) (x',y') (getColor turtle) p
  diff = if p > 0 then SegmentTrace segment else id
```

Funkce `sinDeg` a `cosDeg`, které počítají sinus a kosinus úhlu ve stupních, si definujeme později.

Otáčení a změny pera

Tyto operace jednoduše změní jednotlivé vlastnosti želvy.

```
rotate :: Integer -> Turtle -> (Turtle,DiffTrace)
rotate ang turtle = (turtle',identityDT) where
  turtle' = turtle { getAngle = getAngle turtle + ang }

pen :: Integer -> Turtle -> (Turtle,DiffTrace)
pen p turtle = (turtle',identityDT) where
  turtle' = turtle { getPen = fromIntegral p }
```

Při změně barvy musíme dbát na to, ať se nějaká ze složek RGB modelu nedostane mimo povolený rozsah 0 až 255.

```
color :: Integer -> Integer -> Integer -> Turtle -> (Turtle,DiffTrace)
color r g b turtle = (turtle',identityDT) where
  turtle' = turtle { getColor = (crop r,crop g,crop b) }
  crop x
    | x < 0     = 0
    | x > 255   = 255
    | otherwise = fromIntegral x
```

Rozdvojení želvy

Zbývá nám funkce `split`, která implementuje rozdělení želvy. Prvním parametrem je funkce reprezentující „vedlejší větev“, tedy tělo příkazu `split { ... }`. Této funkci předáme aktuální želvu, získáme z ní `DiffTrace`, který následně aplikujeme na `EmptyTrace`, čímž získáme hodnotu `Trace` reprezentující stopu, kterou „naklonovaná“ želva za sebou zanechala. Vrátíme stav původní želvy a ve výsledné hodnotě `DiffTrace` uložíme částečně aplikovaný konstruktor `SplitTrace`.

```
split :: (Turtle -> (Turtle,DiffTrace)) -> Turtle -> (Turtle,DiffTrace)
split f turtle =
  let (_,dt) = f turtle
      branch = applyDT dt EmptyTrace
  in (turtle,DiffTrace { applyDT = SplitTrace branch })
```

2.7.6 Vyhodnocení výrazů

Typ funkce `evalExpr` jsme si představili již dříve, její implementace je přímočará:

```
evalExpr :: Env -> Expr -> Integer
evalExpr _   (LiteralExpr n) = n
evalExpr env (VariableExpr name) = lookupVar env name
evalExpr env (BinopExpr op left right) =
  let a = evalExpr env left
      b = evalExpr env right
  in case op of
    AddOp -> a + b
    SubOp -> a - b
    MulOp -> a * b
    DivOp -> a `div` b
evalExpr env (NegateExpr expr) = negate $ evalExpr env expr
```

2.7.7 Pomocné funkce

Zbývá nám definovat jen pomocné funkce pro vyhledávání proměnných a procedur v `Env`:

```
lookupDef :: Env -> String -> Define
lookupDef (Env procmap _) name =
  case M.lookup name procmap of
    Just def -> def
    Nothing -> error $ "Undefined procedure " ++ name

lookupVar :: Env -> String -> Integer
lookupVar (Env _ varmap) name =
  case M.lookup name varmap of
    Just num -> num
    Nothing -> error $ "Undefined variable " ++ name
```

Funkce `makeEnv` vytvoří nový `Env` s hodnotami proměnných z *asociativního seznamu* `binds`:

```
makeEnv :: Env -> [(String,Integer)] -> Env
makeEnv (Env procmap _) binds = Env procmap $ M.fromList binds
```

A nakonec funkce sinus a kosinus na stupních:

```
sinDeg, cosDeg :: Integer -> Float
sinDeg n = sin $ fromIntegral n * pi / 180.0
cosDeg n = cos $ fromIntegral n * pi / 180.0
```

2.8 `Krunimir.PngRenderer`

K renderování stop ve formátu PNG použijeme knihovnu *GD* [4]. Její výhodou je, že je velmi jednoduchá na použití. Bohužel neumožňuje vykreslovat čáry jiné tloušťky než 1 px, takže informaci o tloušťce pera nemůžeme využít.

```
module Krunimir.PngRenderer(renderPng) where
import Krunimir.Trace
import qualified Graphics.GD as GD
```

Veškeré operace s obrázky jsou v knihovně GD implementovány jako operace v monádě IO.

```
renderPng :: Trace -> FilePath -> IO ()
renderPng trace fpath = do
  gimg <- GD.newImage (701,701)
  GD.fillImage (GD.rgb 255 255 255) gimg
  mapM_ (mapM_ $ drawSegment gimg) (traceToSegss trace)
  GD.savePngFile fpath gimg
  where
```

```
drawSegment :: GD.Image -> Segment -> IO ()
drawSegment gimg (Segment (x1,y1) (x2,y2) (r,g,b) _pen) =
  GD.drawLine (floor x1,floor y1) (floor x2,floor y2) (GD.rgb r g b) gimg
```

2.9 Krunimir.SvgRenderer

Na exportování do SVG nebudeme potřebovat žádnou speciální knihovnu, jelikož se jedná o formát založený na XML.

```
module Krunimir.SvgRenderer(renderSvg) where
import Krunimir.Trace
```

Podobně jako v modulu `Krunimir.PngRenderer` si nejprve stopu převedeme funkcí `Krunimir.Trace.trac` na seznam seznamů segmentů. Každému segmentu poté pouze vytvoříme jeden element úsečky ve tvaru `<line x1="`*x1*`" y1="`*y1*`" x2="`*x2*`" y2="`*y2*`" stroke="rgb(`*červená*`,`*zelená*`,`*modrá*`)" stroke-wid` *pera*`"/>`.

Tyto elementy stačí obalit do kořenového elementu `<svg> ... </svg>`, ve kterém specifikujeme velikost obrázku, přidat hlavičku a máme hotovo.

```
renderSvg :: Trace -> FilePath -> IO ()
renderSvg trace fpath =
  writeFile fpath $
       svgHeader
    ++ (unlines . map svgSegment . concat . traceToSegss $ trace)
    ++ svgFooter
  where
  svgHeader = "<svg version=\"1.1\" width=\"701\" height=\"701\"\
    \ xmlns=\"http://www.w3.org/2000/svg\">\n"
  svgFooter = "</svg>\n"

  svgSegment (Segment (x1,y1) (x2,y2) (r,g,b) pen) =
    "<line x1=\"" ++ show x1 ++ "\"\
         \ y1=\"" ++ show y1 ++ "\"\
         \ x2=\"" ++ show x2 ++ "\"\
         \ y2=\"" ++ show y2 ++ "\"\
         \ stroke=\"rgb(" ++ show r ++ "," ++ show g ++ "," ++ show b ++  ")\"\
         \ stroke-width=\"" ++ show pen ++ "\"/>"
```

2.10 Příklady

Závěrem uvedeme několik rozsáhlejších příkladů, kdy využijeme želví grafiku k vykreslení několika známých fraktálů.

2.10.1 Hilbertova křivka

Hilbertova křivka je plochu vyplňující fraktál popsaný roku 1891 německým matematikem Davidem Hilbertem. [19] První čtyři iterace této křivky jsou zakresleny na obrázku 2.5a.

Hlavní část programu je procedura `hilbert(n,side)`, která nakreslí Hilbertovu křivku `n`-té iterace. Parametr `side` nabývá hodnot `-1` a `1` a určuje, na kterou stranu se křivka nakreslí. Výsledek programu je na obrázku 2.5b.

```
define hilbert(n,side) {
  left(90*side)
  if(n) {
    hilbert(n-1,-side)
    left(90*side) forward(10)
    hilbert(n-1,side)
    right(90*side) forward(10) right(90*side)
```

```
    hilbert(n-1,side)
    forward(10) left(90*side)
    hilbert(n-1,-side)
  }
  left(90*side)
}

forward(310) right(90)
forward(310) right(90)
pen(1) hilbert(6,-1)
```

2.10.2 Kochova vločka

Kochova vločka je známý fraktál založený na Kochově křivce, kterou v roce 1904 vytvořil švédský matematik Helge von Koch. [20]

Obrázek 2.6a zachycuje první čtyři iterace Kochovy křivky. O kreslení se stará procedura `koch(n)`, jež nakreslí `n`-tou iteraci Kochovy křivky. Tuto proceduru posléze zavoláme třikrát po sobě, čímž vytvoříme Kochovu vločku. Výsledek programu je na obrázku 2.6b.

```
define koch(n) {
  if(n) {
    koch(n-1)
    left(60) koch(n-1)
    right(120) koch(n-1)
    left(60) koch(n-1)
  }
  if(1-n) {
    forward(2)
  }
}

pen(0) forward(-175) left(90) forward(250) right(120)
pen(1) koch(5) right(120) koch(5) right(120) koch(5)
```

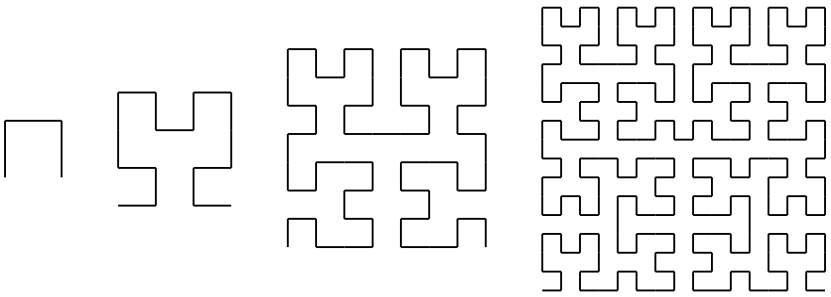
2.10.3 Gosperova křivka

Gosperova křivka, pojmenovaná po svém objeviteli, americkém programátorovi a matematikovi Billu Gosperovi, je plochu vyplňující fraktál. [18]

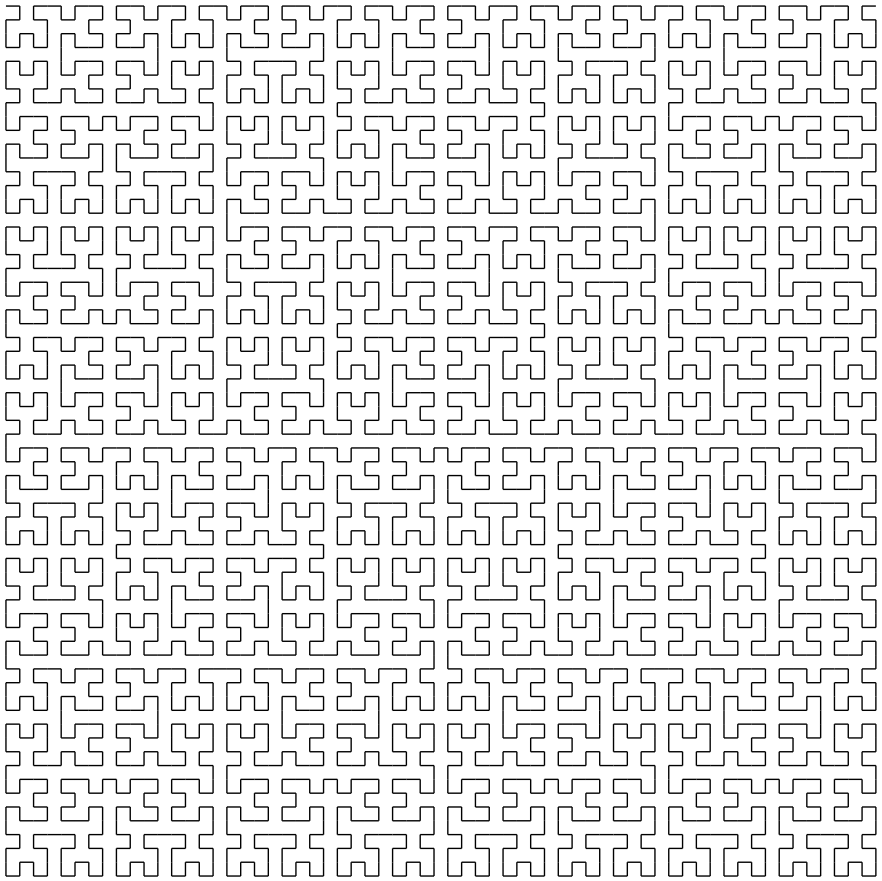
Na vykreslení této křivky musíme použít dvojici procedur, `gosperA` a `gosperB`, přičemž každá kreslí křivku z jiné strany (odpředu a odzadu). Způsob generování je zobrazen na obrázku 2.7a, výstup programu na obrázku 2.7b.

```
define gosperA(n) {
  if(n) {
    right(19)  gosperA(n-1)
    left(60)   gosperB(n-1)
    left(120)  gosperB(n-1)
    right(60)  gosperA(n-1)
    right(120) gosperA(n-1) gosperA(n-1)
    right(60)  gosperB(n-1)
    left(79)
  }
  if(1-n) { forward(4) }
}

define gosperB(n) {
  if(n) {
    right(79)  gosperA(n-1)
    left(60)   gosperB(n-1) gosperB(n-1)
```

Obrázek 2.5: Hilbertova křivka

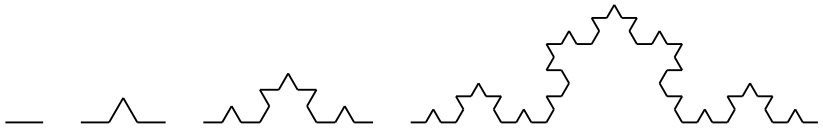


(a) Postupné generování Hilbertovy křivky (zleva doprava iterace 1 až 4).

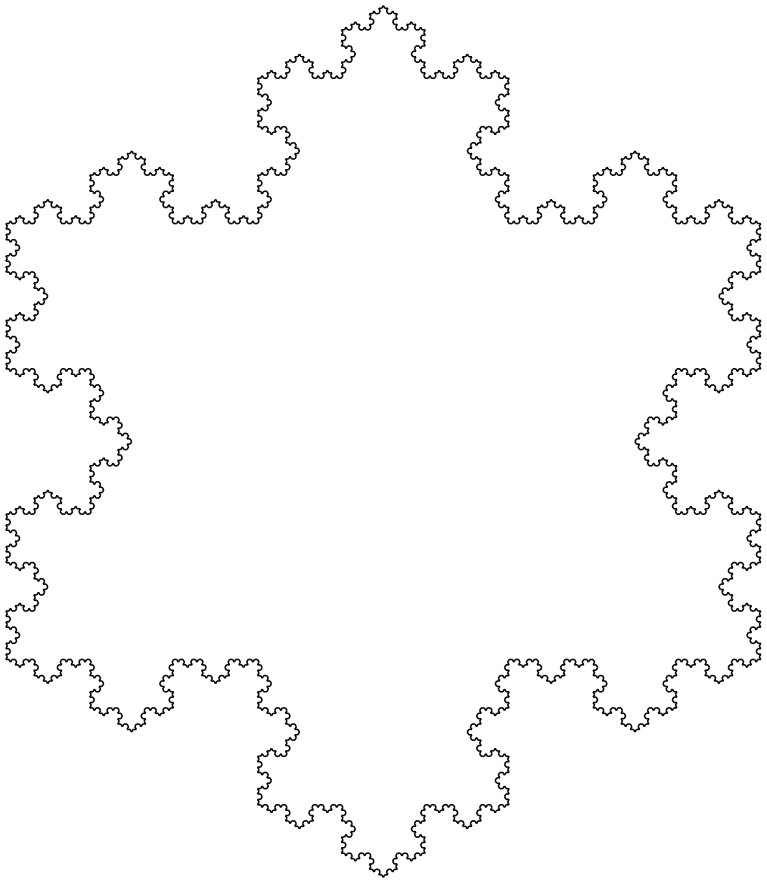


(b) Hilbertova křivka vykreslená Krunimírem

Obrázek 2.6: Kochova vločka



(a) Iterace Kochovy křivky. Kochovu vločku dostaneme spojením tří Kochových křivek.



(b) Kochova vločka vykreslená Krunimírem

```
    left(120)  gosperB(n-1)
    left(60)   gosperA(n-1)
    right(120) gosperA(n-1)
    right(60)  gosperB(n-1)
    left(19)
  }
  if(1-n) { forward(4) }
}

left(19) forward(-250) right(30)
pen(1) gosperA(5)
```

2.10.4 Křivka arrowhead

Křivka arrowhead je podobná Sierpiǹského trojúhelníku, fraktálu polského matematika Wacława Sierpiǹského, který jej popsal v roce 1915. [17]

Podobně jako u Hilbertovy křivky i zde musíme počítat s tím, že křivku musíme vykreslovat ve dvou zrcadlových variantách, k čemuž využijeme parametr `side` procedury `arrowhead(n,side)`. Postup generování ukazuje obrázek 2.8a, výsledek programu obrázek 2.8b.

```
define arrowhead(n,side) {
  if(n) {
    left(60*side)
    arrowhead(n-1,-side)
    right(60*side)
    arrowhead(n-1,side)
    right(60*side)
    arrowhead(n-1,-side)
    left(60*side)
  }
  if(1-n) { forward(2) }
}

right(90) forward(128) left(60) forward(256) left(120)
pen(1) arrowhead(8,1)
```

2.11 Závěr

Vytvořili jsme interpret zadaného programovacího jazyka Krunimír, podporující veškerá rozšíření, vykreslování do dvou grafických formátů, a s velmi přijatelným výkonem.

Představený program by byl podle zadání ze soutěže nejspíše ohodnocen přibližně 300 body z 333. Chybějících 33 bodů je za „televizní přenos“, neboli grafické uživatelské rozhraní (GUI). Tvorba GUI není nijak zvlášť programátorsky zajímavá, proto ji náš program neimplementuje.

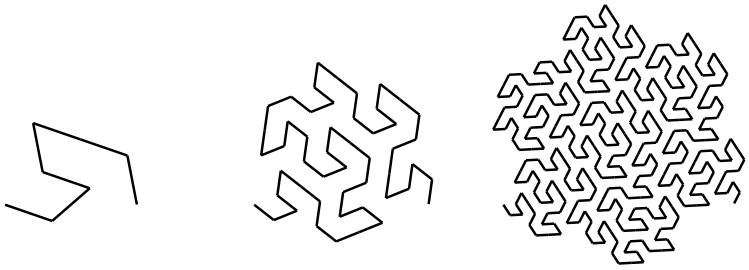
Všechny zdrojové soubory mají dohromady asi 350 řádků kódu. Vezmeme-li v úvahu, že se jedná o kompletní implementaci netriviálního programovacího jazyka, je toto číslo poměrně nízké.[7] Kód je rozdělen do modulů s minimálními vzájemnými závislostmi, může tedy být snadno udržován, upravován a rozšiřován.

Pokud bychom namísto Haskellu použili nějaký imperativní programovací jazyk, například C++, Ruby[8] nebo dokonce Javu, náš program by byl nejspíš delší a jeho modularita by byla nižší. Program by pravděpodobně sestával z parseru vytvořeného pomocí nějakého externího nástroje, který by vytvořil syntaktický strom sestávající z objektů. Vyhodnocení by bylo sloučeno s vykreslováním a probíhalo by voláním metod objektů ze syntaktického stromu, jejichž definice by

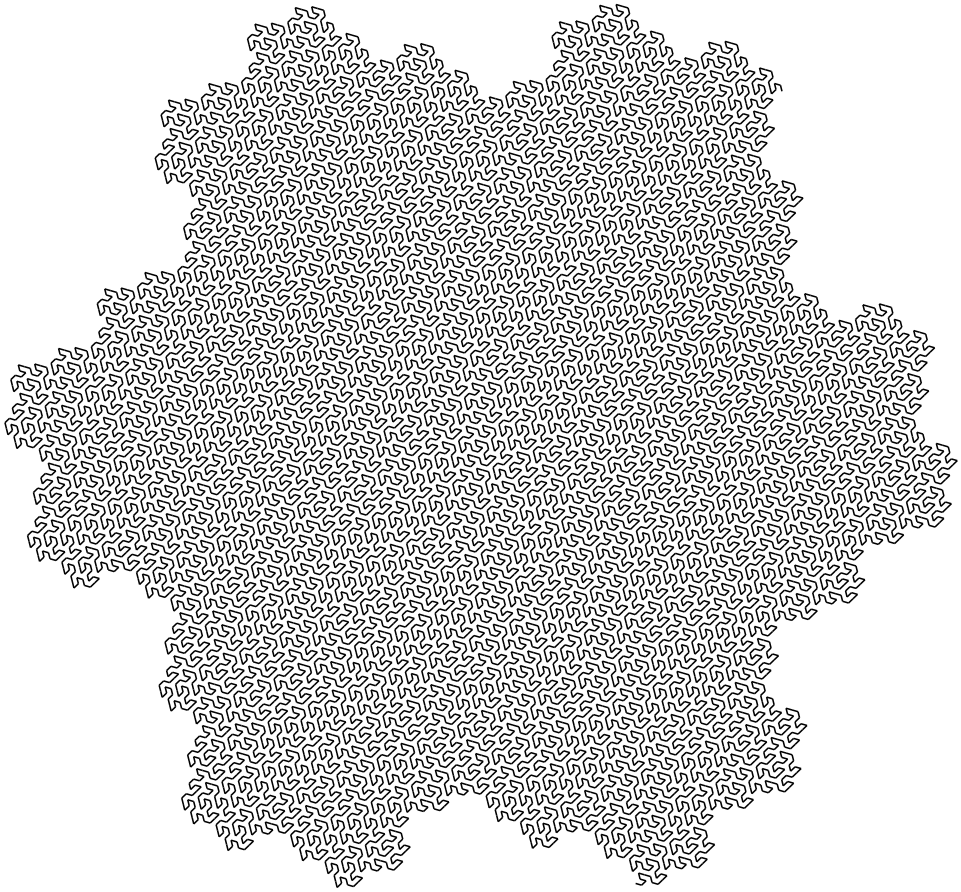
[7] Je nutno podotknout, že program nebyl psán tak, aby řádků bylo co nejméně, ale aby byl co nejpřehlednější.

[8] Pokud bychom využili dynamický jazyk jako Ruby, mohli bychom si ušetřit práci a několika jednoduchými textovými úpravami převést program pro Krunimíra na program v použitém programovacím jazyku, který bychom vyhodnotili pomocí funkce `eval`. Tento postup svého času autor na soutěži úspěšně využil. Jedinou nevýhodou je nesnadnost korektní implementace příkazu `split`, jinak jde o řešení téměř dokonalé :-)

Obrázek 2.7: Gosperova křivka

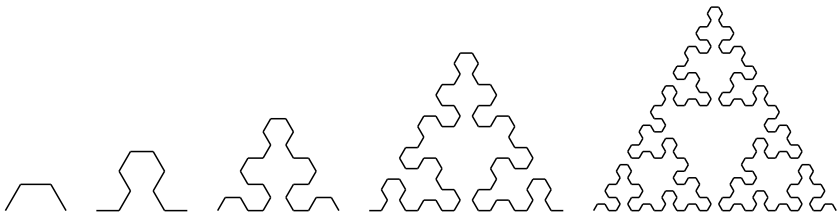


(a) První tři iterace Gosperovy křivky.

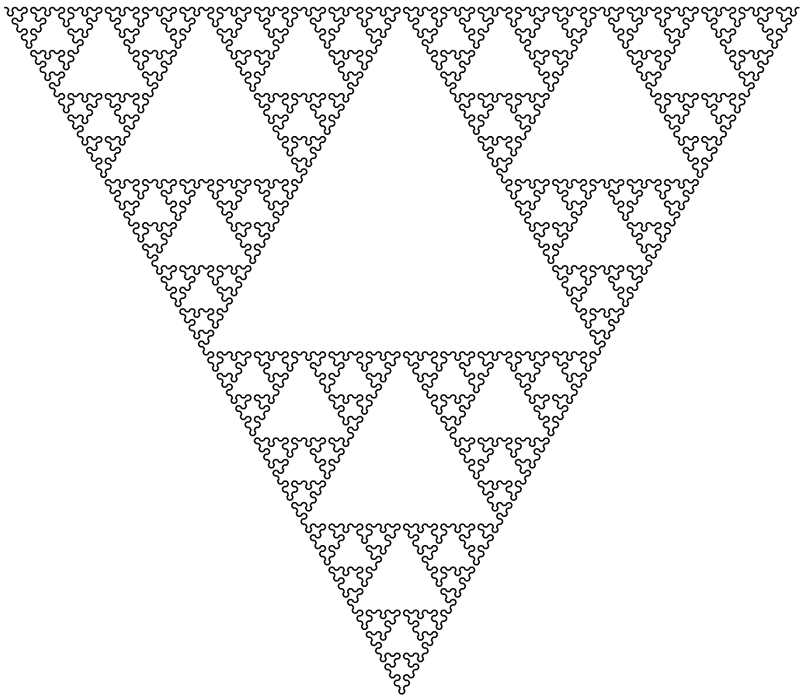


(b) Gosperova křivka vykreslená Krunimírem

Obrázek 2.8: Křivka arrowhead



(a) Prvních pět iterací křivky arrowhead.



(b) Křivka arrowhead vykreslená Krunmírem

byly roztroušeny u definic jednotlivých tříd.[9] Implementovat příkaz `split` by bylo přinejmenším obtížné.

2.11.1 Zdrojové kódy

Veškeré soubory související s Krunimírem jsou v repozitáři s prací uloženy ve složce `krunimir/`. Zdrojové kódy všech uvedených modulů jsou ve složce `krunimir/Krunimir`, testovací soubory ve složce `krunimir/test`. Část testovacích souborů pochází ze soutěže, část byla vytvořena v rámci této práce.

Pro testování je možné použít skript `runtest.sh`, který spustí program pro všechny soubory ze složky `krunimir/test`. Výsledné soubory (s příponou `.test.png` a `.test.svg`) je pak možno vizuálně porovnat s očekávanými výsledky (přípona pouze `.png`).

Složka `krunimir/examples` obsahuje zdrojové kódy příkladů, ze kterých se generují obrázky, jež jsou v práci vloženy.

[9] V Javě bychom nejspíše vytvořili třídu `Statement`, která by měla potomky `ForwardStatement`, `LeftStatement`, `RepeatStatement` apod. přičemž každý by byl ve vlastním souboru...

Kapitola 3

Bílá paní: cesta hradem

> Bílá paní s bezhlavým rytířem společně sledovali bezhlavě prchající turisty. „A nenudíte se sama dole v podhradí," ptal se rytíř, „když všechny návštěvníky vyděsíte tak, že nestačí utíkat?" „Ale ne, jedni mi tam nechali televizor a já na něm sleduji ty, jak se to jmenuje ... telenovely. Jenom teď musím ten televizor zase najít a v mém věku už není procházení zdí tak jednoduché jako zamlada." [1]

Soutěžní úloha z roku 2010 [1] (přetisknuta na straně 63) je variací na klasické téma hledání cesty v bludišti. Bílá paní se nachází na hradě se zdmi a chodbami a snaží se dojít k televizoru, aby stihla začátek své oblíbené telenovely. Není ovšem sama – v hradě se pohybují zvědaví zvědové, kteří se snaží bílou paní odhalit, proto se jim musí vyhnout. Bílá paní sice může procházet zdmi, ale snaží se procházení omezit na minimum, dá přednost delší cestě s menším počtem zdí než kratší cestě s větším počtem zdí.

3.1 Popis úlohy

Bludiště představuje hrad sestávající z volných políček a zdí, ve kterém se po předem daných cyklických drahách pohybují zvědové. Je dána počáteční pozice bílé paní a televizor, který představuje kýžený cíl. Bílá paní může procházet zdmi, ale nesmí být objevena zvědem.

Cesta s počtem kroků k_1, která vede skrz z_1 zdí, je *lepší* než cesta s k_2 kroky a z_2 zdmi právě tehdy, je-li $z_1 < z_2$ nebo $z_1 = z_2 \wedge k_1 < k_2$. Platí-li $z_1 = z_2 \wedge k_1 = k_2$, považujeme obě cesty za rovnocenné. Úkolem je najít *nejlepší* cestu z počáteční pozice k televizoru, aniž by byla objevena zvědem.

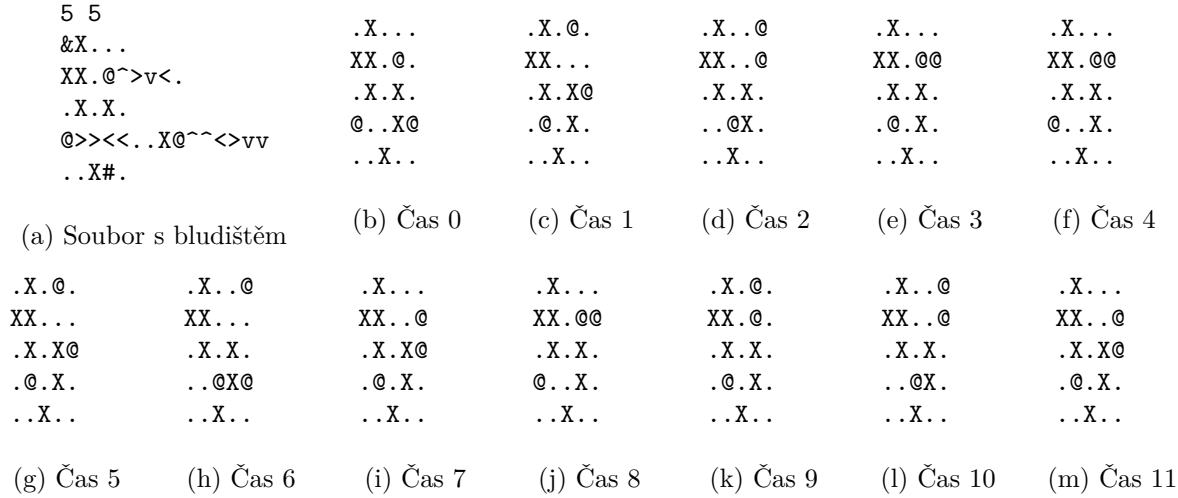
Zvěd bílou paní objeví tehdy, když (a) se oba ve stejném čase nachází na stejném políčku, nebo (b) si v jednom kroku vymění pozice, tj. v čase t stojí zvěd na políčku A a bílá paní na políčku B a v čase $t + 1$ bílá paní na políčku A a *stejný* zvěd na políčku B.

Vstupem programu je soubor s bludištěm uloženým v klasickém formátu programovacích soutěží, tj. na prvním řádku dvě čísla určující rozměry bludiště a na dalších řádcích po znacích jednotlivá políčka – '`.`' volné, '`X`' zeď, '`&`' počáteční pozice bílé paní, '`#`' televizor. Zvěd je uložen jako znak '`@`', za kterým následuje jeho trasa jako sada pohybů '`v`', '`>`', '`^`' a '`<`' (na jih, východ, sever a západ).

Příklad mapy i s pohyby zvědů je na obrázku 3.1.

3.2 Banshee

I když bychom mohli do angličtiny přeložit „bílou paní" jako "white lady", budeme o ní v programu hovořit jako o "banshee", což je v irské mytologii duch ženy přicházející z podsvětí a předznamenávající blížící se smrt. Samozřejmě nejde o ekvivalent klasické české hradní bílé paní, ale pro naše účely je název `Banshee` vhodnější než `WhiteLady`.



(a) Soubor s bludištěm (b) Čas 0 (c) Čas 1 (d) Čas 2 (e) Čas 3 (f) Čas 4

(g) Čas 5 (h) Čas 6 (i) Čas 7 (j) Čas 8 (k) Čas 9 (l) Čas 10 (m) Čas 11

Obrázek 3.1: Příklad bludiště (převzatý z ukázkových souborů ze soutěže) s rozfázovanými pohyby zvědů. Zvědové mají pohyby s periodou 4 a 6, což znamená, že po 12 krocích jsou všichni znovu na stejných pozicích.

3.3 Analýza

Stejně jako u Krunimíra i nyní bude vhodné program rozčlenit do modulů:

`Banshee.Castle` definuje datové typy reprezentující hrad, které budeme používat v celém programu.

`Banshee.CastleParser` poskytuje funkci `parseCastle`, která přečte popis hradu ze souboru ve výše uvedeném formátu.

`Banshee.Navigate` obsahuje jádro programu – funkci `navigate`, která najde cestu v předaném hradu.

`Banshee.Main` exportuje `main`, která slouží jako uživatelské rozhraní programu.

3.4 `Banshee.Castle`

V tomto modulu si nadefinujeme datové typy, které využijeme v ostatních modulech.

```
module Banshee.Castle where
import Data.Array
```

3.4.1 Hrad

Typ `Castle` reprezentuje hrad v podobě, v jaké jsme ho načetli ze souboru. Všechna políčka jsou uložena v poli `castleFields` jako typ `Field`, který může nabývat hodnot `Free` a `Wall`, tedy prázdné políčko či zeď.

```
data Castle = Castle
  { castleFields :: Array Loc Field
  , castleTV :: Loc
  , castleStart :: Loc
  , castleScouts :: [Scout]
  } deriving Show

type Scout = [Loc]
type Loc = (Int,Int)
data Field = Free | Wall deriving (Eq,Show)
```

Pole prvků typu `e` indexovaná typem `i` mají v Haskellu typ `Array i e`. Index pole musí být instancí typové třídy `Ix`. Kromě celočíselných typů jako `Int` či `Integer` tak můžeme použít i n-tice (např. `(Int,Int,Int)`) a vytvořit tak vícerozměrné pole.

Indexem našeho pole `castleFields` je `Loc`, což je pouze synonym pro `(Int,Int)` (dvojice celých čísel), pole je tedy dvojrozměrné. Typ `Loc` budeme používat pro reprezentaci pozice na mapě jako dvojice souřadnic `(x,y)`. Levé horní políčko mapy má souřadnice `(1,1)`, pravé dolní (*šířka mapy*,*výška mapy*).

V `castleTV` je uložena pozice televizoru, v `castleStart` počáteční pozice bílé paní. `castleScouts` obsahuje seznam zvědů (typ `Scout`), kteří jsou reprezentování jako seznam umístění, která postupně navštěvují.

Příklad

Zde je bludiště z obrázku 3.1 uložené jako typ `Castle`.

```
Castle
  { castleFields = array ((1,1),(5,5))
      [((1,1),Free),((1,2),Wall),((1,3),Free),((1,4),Free),((1,5),Free)
      ,((2,1),Wall),((2,2),Wall),((2,3),Wall),((2,4),Free),((2,5),Free)
      ,((3,1),Free),((3,2),Free),((3,3),Free),((3,4),Free),((3,5),Wall)
      ,((4,1),Free),((4,2),Free),((4,3),Wall),((4,4),Wall),((4,5),Free)
      ,((5,1),Free),((5,2),Free),((5,3),Free),((5,4),Free),((5,5),Free)]
  , castleTV = (4,5)
  , castleStart = (1,1)
  , castleScouts =
    [ [(5,4),(5,3),(5,2),(4,2),(5,2),(5,3)]
    , [(1,4),(2,4),(3,4),(2,4)]
    , [(4,2),(4,1),(5,1),(5,2)]]
  }
```

3.4.2 Řezy hradu

V průběhu hledání cesty budeme muset být schopni rychle určit, zda se v daném čase nachází na daném políčku zvěd nebo ne. K tomu využijeme typ `Slice`, který reprezentuje „řez časoprostorem hradu“, tedy umístění zdí, volných políček a zvědů v daném čase (pohybují se samozřejmě pouze zvědové, zdi a volná políčka zůstávají na svém místě, pouze mohou být „překryta“ zvědem).

Zvědové se na mapě pohybují po konstantní uzavřené dráze, což znamená, že pokud je zvěd v čase t na daném políčku, bude na tomto políčku i v čase $t + kp$, kde p je délka jeho trasy, pro všechna $k \in \mathbb{Z}$.

Z toho plyne, že po konečném počtu kroků se všichni zvědové z hradu dostanou zpátky na své startovní pozice. Tomuto počtu kroků budeme říkat *perioda* a je roven nejmenšímu společnému násobku délek tras všech zvědů v hradu.[1]

Pro hrad s periodou p tedy stačí vygenerovat prvních p „řezů“. Pro čas t získáme číslo příslušného řezu (*offset*) ze zbytku po dělení $t \div p$ (označíme-li počáteční stav časem $t = 0$).

Jednotlivé „řezy“ pro jednoduchou mapu jsou zobrazeny na obrázku 3.1.

```
newtype Slice = Slice (Array Loc SliceField) deriving Show
data SliceField = FreeSF | WallSF | ScoutSF [Loc] deriving (Eq,Show)
```

Typ `Slice` je jen „obalený“ typ `Array Loc SliceField` (tzv. *newtype*). Od typového synonymu (uvozeného klíčovým slovem `type`) se liší tím, že jej musíme explicitně „rozbalovat“ pomocí konstruktoru `Slice` a že typ `Slice` není ekvivalentní typu `Array Loc SliceField`, můžeme pro něj tedy definovat např. jiné typové třídy.[2] Je tedy velmi podobný klasickému datovému typu (klíčové slovo `data`) s jediným datovým konstruktorem s jediným prvkem.

[1] Z původního zadání plyne, že délka tras všech zvědů musí být sudá a že nepřesáhne 8, tudíž nejvyšší možná perioda je 24. Náš program akceptuje libovolně dlouhé trasy zvědů, periodu tedy budeme počítat pro každý hrad zvlášť.

[2] To je důvod, proč Haskell takovéto typy zavádí.

`SliceField` je obdobou typu `Field`, rozdíl je v tom, že ve `SliceField` je možno uložit i zvěda. Parametr typu `[Loc]` konstruktoru `ScoutSF` je následující pozice zvědů nacházejících se na tomto políčku; využijeme ji při zjišťování, zda si bílá paní a zvěd v jednom kroku nevyměnili pozice, což je považováno za objevení bílé paní zvědem.

3.4.3 Řezání hradu

Pro samotné rozřezání časoprostoru hradu na tenké plátky definujeme funkci `sliceCastle`.

```
sliceCastle :: Castle -> [Slice]
sliceCastle castle = map slice [0..period-1] where
  fields = castleFields castle
  period = foldl lcm 1 . map length . castleScouts $ castle
  cycledScouts = map cycle $ castleScouts castle
  sfFields = fmap fieldToSF fields

  slice s = Slice $ accum acc sfFields
    [((x,y),(nextx,nexty))
    | scout <- cycledScouts
    , let (x,y):(nextx,nexty):_ = drop s scout
    , x >= 1 && x <= width
    , y >= 1 && y <= height
    ]

  acc :: SliceField -> Loc -> SliceField
  acc old scoutNext = case old of
    FreeSF -> ScoutSF [scoutNext]
    WallSF -> ScoutSF [scoutNext]
    ScoutSF scoutNexts' -> ScoutSF $ scoutNext:scoutNexts'

  fieldToSF Free = FreeSF
  fieldToSF Wall = WallSF

  ((1,1),(width,height)) = bounds $ castleFields castle
```

Pro přehlednost bude vhodné rozebrat si jednotlivé proměnné:

`fields` je pole políček (`Array Loc Field`) získané z předaného hradu.

`period` je délka periody, s jakou se pohybují všichni zvědové. Jedná se o nejmenší společný násobek délek tras všech zvědů z hradu. (Funkce `lcm a b` spočte nejvyšší společný násobek dvou čísel `a` a `b`.)

`cycledScouts` je seznam tras všech zvědů (`[[Loc]]`) „zacyklených“ do nekonečně se opakujících sekvencí pomocí funkce `cycle`.

`sfFields` je pole `field` převedené z hodnot typu `Field` na `FieldSF`. Obsahuje tedy stejnou informaci, jen jiného typu.

`slice s` je funkce, která pro offset `s` vrátí příslušný řez. Získáme jej tak, že do „čistého“ pole `sfFields`, obsahujícího pouze `FreeSF` a `WallSF`, přidáme na příslušné pozice i hodnoty `ScoutSF`, k čemuž použijeme funkci `accum`. Nesmíme zapomenout zkontrolovat, že se přidávaný zvěd nachází v hradu, jinak bychom mohli dostat za běhu chybu (přistupovali bychom k indexu nacházejícímu se mimo rozsah pole).

Typ funkce `accum` je `(e -> a -> e) -> Array i e -> [(i, a)] -> Array i e`.[3] První argument je akumulační funkce f, druhým výchozí pole ary a třetí seznam asociací xs. Výsledné pole obsahuje stejné prvky jako ary, s tím rozdílem, že pro každou asociaci

[3] Pro jednoduchost jsme vynechali omezení třídy `Ix i => ...`

index-hodnota *(i,x)* ze seznamu *xs* zavolá funkci *f*, která na základě předchozí hodnoty z pole a hodnoty *x* vrátí novou hodnotu, která se uloží ve výsledném poli na indexu *i*.

Důležité je, že stejný index se v seznamu *xs* může objevit i vícekrát – v tom případě se funkci *f* předá hodnota získaná z předchozí asociace. Takto je tedy možno stejný prvek „upravit“ vícekrát.

Je důležité si uvědomit, že pole *ary*, předané funkci `accum`, se nijak nezmění, výsledek se předá v novém poli (jinak to ve funkcionálním jazyku ani nejde).

V našem případě funkci `accum` předáme pomocnou akumulační funkci `acc`, pole `sfFields` a seznam vyjádřený pomocí generátoru seznamu, který obsahuje dvojice `((x,y),(nextx,nexty))`, kde `(x,y)` je pozice zvěda v čase `s` a `(nextx,nexty)` je pozice stejného zvěda v čase `s+1`. Funkce `accum` tedy pomocí funkce `acc` na pozici *(x,y)* zapíše informaci o tom, že se zde nachází zvěd, jehož další destinací je pozice `(nextx,nexty)`.

`acc old scoutNext` tedy udělá jednu ze dvou věcí: (a) je-li políčko prázdné (`FreeSF`) nebo zeď (`WallSF`), nahradí jej hodnotou `ScoutSF` s jediným zvědem, jehož následující pozice je `scoutNext`; nebo (b) pokud na políčku již byli zapsáni nějací zvědové, přidá k seznamu jejich následujících pozic i `scoutNext`.

`fieldToSF` je pomocná funkce, kterou jsme využili v definici pole `sfFields`.

3.5 Banshee.CastleParser

Na parsování vstupního souboru s hradem použijeme opět knihovnu `parsec`, kterou jsme si popsali v sekci 2.5.

```
module Banshee.CastleParser(parseCastle) where
import Banshee.Castle

import Text.Parsec
import Control.Applicative ((<$>),(<*),(*>))
import Control.Monad
import Data.Array
```

Funkce `parseCastle` a typ `Parser` jsou definovány obdobně jako v Krunimírovi:

```
parseCastle :: String -> String -> Either ParseError Castle
parseCastle = parse castle

type Parser a = Parsec String () a
```

3.5.1 SemiCastle

Nejprve nadefinujeme typ `SemiCastle`, který budeme používat v průběhu parsování. Tento typ představuje „napůl známý“ hrad. Pozice startu a televize se dozvíme až v průběhu čtení souboru, proto na jejich uložení použijeme typ `Maybe Loc`. Podobně na uložení políček nepoužijeme pole, ale seznam, jelikož se informace o jednotlivých políčkách budeme dovídat postupně.

```
data SemiCastle = SemiCastle
  { scFields :: [(Loc,Field)]
  , scTV :: Maybe Loc
  , scStart :: Maybe Loc
  , scScouts :: [Scout]
  }
```

`zeroCastle` je „prázdný“ (nebo nulový) hrad, o kterém zatím nevíme vůbec nic, a `scAdd` je pomocná funkce, která do předaného `SemiCastle` přidá jedno políčko.

```
zeroCastle = SemiCastle
  { scFields = [] , scTV = Nothing
  , scStart = Nothing , scScouts = [] }

scAdd :: Loc -> Field -> SemiCastle -> SemiCastle
scAdd loc fld sc = sc { scFields = (loc,fld):scFields sc }
```

3.5.2 Jednotlivé parsery

Celý hrad

Parser `castle` se stará o parsování celého hradu:

```
castle :: Parser Castle
castle = do
  width <- read <$> (spaces *> many digit)
  height <- read <$> (spaces *> many digit)
  sc <- spaces >> foldM (row width) zeroCastle [1..height]
  eof
  tv <- case scTV sc of
    Just t -> return t
    Nothing -> parserFail "There was no television in the castle"
  start <- case scStart sc of
    Just s -> return s
    Nothing -> parserFail "There was no starting position in the castle"
  return $ Castle
    { castleFields = array ((1,1),(width,height)) $ scFields sc
    , castleTV = tv, castleStart = start
    , castleScouts = scScouts sc }
```

Nejprve přečteme šířku (`width`) a výšku (`height`) hradu, která je zapsána jako dvojice čísel na začátku souboru. Následně pomocí monadického kombinátoru `foldM` a parseru `row`, který si brzy nadefinujeme, přečteme všechny řádky ze souboru a získáme tak hodnotu `cs` typu `SemiCastle`, která reprezentuje přečtený hrad.

Funkce `foldM` má typ `Monad m => (a -> b -> m a) -> a -> [b] -> m a` a je analogická funkci `foldl`, ovšem pracuje s monádami.

Jakmile máme všechny řádky přečteny, ujistíme se, jestli jsme opravdu přečetli pozici televizoru (`tv`) a startu (`start`). Pokud ne, pomocí `parserFail` parsování ukončíme s chybou. Pokud ano, nic nám už nebrání vytvořit hodnotu `Castle` a vrátit ji.

Řádek

Na parsování jednoho řádku použijeme opět `foldM` a parser `field`, který přečte jedno políčko z mapy. Jako argumenty funkce `row` musíme předat šířku mapy, `SemiCastle` který upravujeme a číslo řádku, který čteme.

Na konci řádku by měl být znak konce řádku, náš parser ovšem akceptuje jakékoli prázdné znaky.

```
row :: Int -> SemiCastle -> Int -> Parser SemiCastle
row width sc y = foldM (field y) sc [1..width] <* spaces
```

Políčko

Nyní už se dostáváme k nejdůležitější části našeho parseru – čtení jednoho políčka:

```
field :: Int -> SemiCastle -> Int -> Parser SemiCastle
field y sc x = sc `seq` free <|> wall <|> start <|> tv <|> scout
  where
    free  = char '.' >> return (scAdd (x,y) Free sc)
    wall  = char 'X' >> return (scAdd (x,y) Wall sc)
```

```
    start = char '&' >> case scStart sc of
      Nothing -> return $ scAdd (x,y) Free sc { scStart = Just (x,y) }
      Just (x',y') -> parserFail $
        "There is already starting position at " ++ show (x',y')
    tv    = char '#' >> case scTV sc of
      Nothing -> return $ scAdd (x,y) Free sc { scTV = Just (x,y) }
      Just (x',y') -> parserFail $
        "There is already television at " ++ show (x',y')
    scout = char '@' >> do
      moves <- many move
      let locs = if null moves
            then [(x,y)]
            else applyMoves (x,y) $ init moves
      return $ scAdd (x,y) Free sc { scScouts = locs:scScouts sc }
```

Před samotným parsováním nejprve pomocí funkce `seq` vynutíme vyhodnocení proměnné `sc`. Pokud bychom to neudělali, postupně by se během parsování vytvořil řetěz nevyhodnocených hodnot `SemiCastle`. K jeho vyhodnocení by došlo až na konci parsování a u velkých hradů by mohlo dojít k přetečení zásobníku.[4]

Pro každý typ políčka jsme si nadefinovali vlastní pomocný parser. Prázdná pole (parser `free`) a zdi (`wall`) jsou jednoduché, pouze do hradu přidáme jedno pole.

Narazíme-li na políčko s televizorem (`tv`) nebo startovní pozicí (`start`), nejprve zkontrolujeme, jestli už jsme televizor či start jednou nepřečetli, a pokud ano, skončíme parsování s chybou; v opačném případě upravíme odpovídající část `SemiCastle`.

Posledním typem políčka je zvěd (parser `scout`). Za znakem '`@`' následuje několik pohybů, ze kterých musíme spočítat zvědovu trasu. Pokud za zvědem není zadán ani jeden pohyb, jeho trasa je prostá – stojí stále na své původní pozici.

Pokud je zadáno pohybů více, pomocí funkce `applyMoves`, kterou si vzápětí nadefinujeme, pohyby převedeme na seznam pozic. Ze seznamu pohybů ovšem předáme všechny pohyby kromě posledního (funkcí `init`), protože v posledním kroku svého cyklu se zvěd vždy přesune na svou počáteční pozici (tuto vlastnost nekontrolujeme, ale tímto si ji vynutíme).

3.5.3 Pohyby

Pro reprezentaci pohybů využijeme jednoduchý algebraický datový typ:

```
data Move = North | West | South | East deriving (Eq,Show)
```

Parser `move`, který jsme využili při čtení zvěda, vrací hodnoty tohoto typu podle přečteného znaku:

```
move :: Parser Move
move = char '^' *> return North
   <|> char '<' *> return West
   <|> char 'v' *> return South
   <|> char '>' *> return East
```

Funkce `applyMoves` „aplikuje“ seznam pohybů na počáteční pozici, čímž dostaneme seznam pozic:

```
applyMoves :: Loc -> [Move] -> [Loc]
applyMoves (x,y) []     = (x,y):[]
applyMoves (x,y) (m:ms) = (x,y):case m of
  North -> applyMoves (x,y-1) ms
  West  -> applyMoves (x-1,y) ms
  South -> applyMoves (x,y+1) ms
  East  -> applyMoves (x+1,y) ms
```

[4] Velkými hrady myslíme hrady s milióny políček – pokud bychom se drželi omezení na 80×80 polí, jak je uvedeno v zadání, žádný problém by nenastal.

3.6 Banshee.Navigate

```
module Banshee.Navigate(navigate) where
import Data.Array
import Data.Array.ST
import Data.Maybe
import Control.Monad
import Control.Applicative
import Control.Monad.ST

import Banshee.Castle
```

3.6.1 Popis algoritmu

Náš algoritmus hledání cesty je založen na prohledávání všech možných cest do šířky s počátkem na startovní pozici bílé paní. O cestě hradem můžeme uvažovat jako o sérii změny „časopozic“, kde časopozicí budeme nazývat dvojici *pozice* v hradu a *času*, potřebného k cestě na tuto pozici modulo perioda hradu (offset). V hradu je tedy celkem $w \times h \times p$ časopozic, kde w je šířka hradu, h je výška a p je perioda. Nejkratší cesta nikdy neprochází stejnou časopozici dvakrát, jelikož by obsahovala cyklus.

V průběhu výpočtu si budeme udržovat pole, jež pro každou časopozici obsahuje nejkratší cestu, jak této časopozice dosáhnout. Pokud v každém kroku algoritmu přiřadíme jedné časopozici nejkratší cestu, dostaneme nejhorší časovou složitost $O(whp)$, tedy přímo úměrnou počtu políček v hradu a periodě.

Ve skutečnosti vždy provedeme řádově méně kroků než whp, jelikož v hradech bez větších překážek nalezneme cestu rychle, v hradech s více překážkami sice cestu nalezneme po delší době, ale velká část časopozic bude nedostupných.

3.6.2 Typ `Path`

Během výpočtu budeme často manipulovat s cestami. Abychom nemuseli stále dokola počítat délky cest pomocí `length`, budeme používat typ `Path`, který obsahuje seznam pozic, kterými cesta vede, spolu s její délkou (tedy počtem těchto pozic).

```
data Path = Path Int [Loc] deriving Show
pathLength (Path len _) = len
```

3.6.3 Určení možných pohybů z políčka

Funkce `moves` slouží k určení všech možných sousedních pozic, na které se bílá paní může dostat z dané pozice. Tato funkce má typ `moves :: Bool -> Slice -> Slice -> (Loc,Path) -> [(Loc,Path)]`:

- První argument (typu `Bool`) určuje, jestli můžeme procházet zdmi.
- Druhý argument (`Slice`) je řez hradu v čase, ze kterého vycházíme.
- Třetí argument (`Slice`) je řez hradu v dalším kroku (v čase, ve kterém dorazíme na další políčko).
- Poslední argument (`(Loc,Path)`) je dvojice – pozice, ze které vycházíme, a cesta, kterou jsme se na tuto pozici dostali.
- Výsledkem (`[(Loc,Path)]`) je seznam pozic, kam se můžeme dostat, spolu s příslušnými cestami.

```
moves :: Bool -> Slice -> Slice -> (Loc,Path) -> [(Loc,Path)]
moves thruWalls (Slice foreslice) (Slice afterslice) ((x,y),Path len ps) =
  [((tox,toy),Path (len+1) $ (x,y):ps)
    | (tox,toy) <- [(x+1,y),(x-1,y),(x,y+1),(x,y-1)]
    , tox >= 1 && tox <= width
    , toy >= 1 && toy <= height
    , case afterslice ! (tox,toy) of
        FreeSF -> True
        WallSF -> thruWalls
        ScoutSF _ -> False
    , case foreslice ! (tox,toy) of
        ScoutSF scouts | (x,y) `elem` scouts -> False
        _ -> True
    ]
  where
    ((1,1),(width,height)) = bounds foreslice
```

Funkci jsme implementovali pomocí generátoru seznamu (*list comprehension*). Nejprve si vygenerujeme dvojice `(tox,toy)` všech čtyř pozic sousedících s výchozí pozicí `(x,y)`. Následně zkontrolujeme, jestli se tyto pozice nachází v hradu, a podíváme se, co nás na této pozici v hradu čeká.

Je-li to volné políčko, je vše v pořádku. Pokud narazíme na zeď, tak pokračujeme pouze pokud má argument `thruWalls` hodnotu `True`. Pokud by se bílá paní dostala na stejné políčko se zvědem, musíme tuto pozici přeskočit.

Nakonec zkontrolujeme, jestli se v čase, ve kterém jsme vyšli, na políčku, na které jdeme, nevyskytují zvědi. Pokud ano, a některý z těchto zvědů se v dalším tahu přesune na pole, ze kterého jsme vyšli, znamená to, že si bílá paní se zvědem prohodí místo, což znamená, že by ji zvěd objevil a do této pozice tedy nemůžeme.

3.6.4 Monáda `ST`

V našem algoritmu budeme pracovat s polem, ve kterém si budeme ukládat nejlepší cesty, které jsme nalezli do každé časopozice v hradě. Mohli bychom použít klasické pole `Array`, ale protože do tohoto pole budeme zapisovat po jednom prvku, byl by program velmi neefektivní, jelikož při každém zápisu se celé pole musí zkopírovat.

Proto využijeme monádu `ST s` [12]. Tento typ je podobný typu `IO` v tom, že umožňuje akcím využívat měnitelné datové struktury, narozdíl od `IO` jsou však akce v monádě `ST s` omezeny pouze na „vnitřní“ stav, nemohou tedy např. přistupovat k disku nebo vypisovat znaky na obrazovku, pouze pracovat s měnitelnou pamětí. To ale znamená, že každá taková akce je *deterministická* a monádu `ST s` tedy můžeme spustit a získat její výsledek v čistém kódu.

Na akci typu `ST s a` je tedy možno pohlížet jako na výpočet, jenž využívá interní stav a jehož výstup má typ `a`. Proměnná `s` slouží k zajištění, že tento interní stav „neunikne“ z monády `ST`. Funkce, která nám umožní provést výpočet v monádě `ST` a získat jeho výstup se jmenuje `runST` a má typ `(forall s. ST s a) -> a`.[5]

Měnitelné pole `STArray`

Měnitelné pole, které můžeme použít v monádě `ST s`, má typ `STArray s i e`, kde `s` je stejná typová proměnná, kterou předáme do `ST s`, `i` je typ indexu a `e` typ prvků.

S takovýmto polem můžeme provádět následující operace:

`getBounds :: STArray s i e -> ST s (i,i)`
`getBounds ary` získá rozsah indexů pole `ary`.

[5] Kvantifikátor `forall` je součástí typové magie zajišťující bezpečnost použití `ST` a je součástí rozšíření `ExistentialQuantification` jazyka Haskell.

`newArray :: (i,i) -> e -> ST s (STArray s i e)`
: `newArray (a,b) x` vytvoří nové pole s rozsahem indexů od `a` do `b`, jehož prvky jsou inicializované na hodnotu `x`.

`readArray :: STArray s i e -> i -> ST s e`
: `readArray ary i` přečte hodnotu prvku na indexu `i` v poli `ary`.

`writeArray :: STArray s i e -> i -> e -> ST s ()`
: `writeArray ary i x` zapíše hodnotu `x` do indexu `i` v poli `ary`.

3.6.5 Hledání cest v souvislých oblastech bez průchodu zdí

Nyní si implementujeme funkci `flood`, která pro daný seznam výchozích pozic s příslušnými cestami nalezne nejlepší cestu do každé časopozice hradu, kam se můžeme dostat bez průchodu zdí. Tyto nejlepší cesty zapíše do předaného měnitelného pole `STArray` a vrátí buď cestu k televizoru, pokud byla takováto nalezena, nebo seznam dosažených pozic s příslušnými cestami.

Pro předaný seznam výchozích pozic a cest musí platit, že všechny cesty prochází stejným počtem zdí a jsou seřazeny vzestupně podle délky, což znamená, že lepší cesta vždy v tomto seznamu předchází cestu horší.

Jednotlivé argumenty této funkce jsou následující:

- První argument (typ `Castle`) je hrad, ve kterém hledáme cestu.
- Druhý argument (`[Slice]`) je seznam řezů, na které je tento hrad „nakrájen“.
- Třetí argument (`STArray s (Int,Loc) (Maybe Path)`) je měnitelné pole `bests`, které každé časopozici přiřazuje nejlepší cestu, pokud taková existuje.
- Poslední argument (`[(Loc,Path)]`) je seznam startovních pozic seřazených podle délky (tyto ještě nejsou zapsány v poli `bests`).
- Výsledek (`ST s (Either Path [(Loc,Path)])`) je výpočet v monádě `ST s`, jehož výsledek je buď cesta k televizoru (hodnota `Left`), nebo seznam pozic a cest, jež byly přidány do pole `bests`, seřazených podle délky.

```
flood :: Castle -> [Slice] -> STArray s (Int,Loc) (Maybe Path) ->
  [(Loc,Path)] -> ST s (Either Path [(Loc,Path)])
flood castle slices bests = step 0 (cycle slices)
  where
  step _ _ [] = return $ Right []
  step t (slice1:slice2:slices') locpaths = do
    let (starts,rest) = span ((<= t) . pathLength . snd) locpaths
        offset = t `mod` period

    (starts',nextss) <- fmap (unzip . catMaybes) $ forM starts $ \((x,y),path) -> do
      uncov <- isNothing <$> readArray bests (offset,(x,y))
      if uncov then do
          writeArray bests (offset,(x,y)) $ Just path
          return . Just . (,) ((x,y),path) $ moves False slice1 slice2 ((x,y),path)
        else return Nothing

    tv <- readArray bests (offset,castleTV castle)
    case tv of
      Nothing -> fmap (starts'++) <$>
        step (t+1) (slice2:slices') (concat nextss ++ rest)
      Just path -> return $ Left path

  period = length slices
```

Téměř všechnu práci v této funkci vykoná pomocná funkce `step`. Jejím prvním argumentem je čas `t`, ve kterém má začít, druhým nekonečný seznam řezů hradu (prvním prvkem je řez v čase `t` – `slice1`, druhým řez v čase `t+1` – `slice2`) a posledním seznam dvojic pozic a cest do těchto pozic, ze kterých hledáme cesty (`locpaths`).

Pokud jsme dostali tento seznam prázdný, funkce `step` jednoduše vrátí prázdný seznam značící, že televizor nebyl nalezen a do pole `bests` nebylo přidáno nic.

V opačném případě si nejprve seznam `locpaths` rozdělíme na dvě části – seznam `starts` obsahuje všechny páry pozice-cesta ze kterých vyjdeme v tomto kroku, v seznamu `rest` je zbytek, který zpracujeme v dalších krocích.

Následně pro každou dvojici ze seznamu `starts` zkontrolujeme, jestli v poli `bests` není náhodou již nalezena nějaká jiná cesta, která je tedy lepší. Pokud ne, do pole zapíšeme tuto cestu a vrátíme jak tuto dvojici, tak seznam sousedních políček, „zabalené“ v konstruktoru `Just`. Pokud již byla nalezena lepší cesta, vrátíme `Nothing`.

Tímto získáme seznam `[Maybe ((Loc,Path),[(Loc,Path)])]`, který následně pomocí složené funkce `unzip . catMaybes` převedeme na `([(Loc,Path)],[[(Loc,Path)]])`, tedy dvojici, jejíž první prvek je seznam pozic a cest, které byly uznány za nejlepší (tento seznam označíme jako `starts'`), a seznam seznamů políček a cest k nim, na které se z těchto nejlepších cest můžeme dostat (v proměnné `nextss`).

Pak zkontrolujeme, jestli jsme neobjevili nejkratší cestu k televizoru. Pokud ano, tak tuto cestu vrátíme jako `Left`, jinak rekurzivně zavoláme `step` znovu a pomocí funkce `fmap` k jejímu výsledku přidáme ještě seznam `starts'`.

3.6.6 Hledání cest včetně procházení zdí

Jako poslední implementujeme vlastní funkci `navigate`. Budeme jí předávat hrad, seznam řezů tohoto hradu a hodnotu `Bool` značící, jestli povolíme procházení zdí.

```
navigate :: Castle -> [Slice] -> Bool -> Maybe [Loc]
navigate castle slices thruWalls = runST $ do
  let ((1,1),(width,height)) = bounds $ castleFields castle
      start = [(castleStart castle,Path 0 [])]
  bests <- newArray ((0,(1,1)),(period-1,(width,height))) Nothing
  result <- if thruWalls
    then wallStep bests start
    else flood castle slices bests start
  case result of
    Left (Path _ locs) -> return . Just . reverse $ castleTV castle : locs
    Right _ -> return Nothing

  where
  wallStep :: STArray s (Int,Loc) (Maybe Path) -> [(Loc,Path)] ->
    ST s (Either Path [(Loc,Path)])
  wallStep bests locpaths = do
    result <- flood castle slices bests locpaths
    case result of
      Left tvPath -> return $ Left tvPath
      Right [] -> return $ Right []
      Right locpaths' -> do
        let nexts = concat . (`map` locpaths') $ \((x,y),path@(Path len _)) ->
              let offset = len `mod` period
                  offset' = (len+1) `mod` period
              in  moves True (slices !! offset) (slices !! offset') ((x,y),path)
        wallStep bests nexts

  period = length slices
```

Veškerou práci ve funkci `navigate` provedeme v monádě `ST s`, kterou pomocí `runST` „spustíme“ a získáme výsledek. Poté, co zjistíme šířku a výšku hradu a nadefinujeme seznam startovních

pozic (`start`), vytvoříme pole `bests`. Veškeré prvky tohoto pole budou nastaveny na hodnotu `Nothing`.

Pokud nemáme povoleno procházet skrz zdi (argument `thruWalls` má hodnotu `False`), použijeme funkci `flood`, jelikož ta nikdy negeneruje cesty, které prochází zdmi. V opačném případě zavoláme pomocnou funkci `wallStep`, jež najde i cesty skrz zdi a kterou si vzápětí popíšeme.

Ať už jsme použili `flood` nebo `wallStep`, získali jsme výsledek typu `Either Path [(Loc,Path)]` v proměnné `result`. Pokud jsme dostali `Left`, byla nalezena cesta k televizi. K seznamu pozic z této cesty ještě přidáme pozici televizoru a cestu otočíme, aby byla pozice startu na prvním místě a televize na místě posledním. Hodnota `Right` značí, že cesta k televizoru nebyla nalezena; v tom případě vrátíme `Nothing`.

Pomocná funkce `wallStep`

Funkci `wallStep` předáme jako argument pole `bests` a seznam startovních pozic a cest. Nejprve zavoláme funkci `flood` a její výsledek přiřadíme proměnné `result`. Pokud nalezla cestu k televizoru, jednoduše ji vrátíme. Pokud takovou cestu nenalezla a zároveň nenašla cestu ani do jedné nové pozice v hradě, znamená to, že už žádnou novou cestu nalézt nemůžeme a proto vrátíme hodnotu `Right`, značící neúspěch.

Pokud ale funkce `flood` nalezla cestu do nových pozic v hradě, seznam dvojic těchto pozic a cest do nich uložíme do proměnné `locpaths'`. Následně každou z těchto pozic projdeme a pomocí funkce `moves` (jíž jako první argument předáme `True`) získáme seznam pozic, do kterých se z nich můžeme jedním pohybem i skrz zeď dostat. Proměnná `nexts` tedy obsahuje seznam pozic a cest, které můžou mít o jednu zeď v cestě více. S tímto seznamem poté znovu rekurzivně zavoláme `wallStep`, která nyní najde i cesty za právě prošlými zdmi.

3.7 `Banshee.Main`

Modul `Banshee.Main` obsahuje uživatelské rozhraní programu. Podobně jako u Krunimíra budeme náš program spouštět v terminálu, ale umožníme uživateli pomocí argumentů zadaných na příkazovém řádku měnit chování programu.

3.7.1 Popis použití programu

Programu na příkazové řádce předáme jméno vstupního souboru s hradem a navíc můžeme předat některé z následujících přepínačů:

`-h`, `-?` nebo `--help` zobrazí návod na použití programu a skončí.

`-n` nebo `--not-through` zakáže procházení zdmi (ve výchozím nastavení je procházení zdmi povoleno).

`-s` nebo `--show-castle` vypíše předaný hrad se zvýrazněnou nalezenou cestou (výchozí chování).

`-q` nebo `--quiet` způsobí, že se vypíše pouze jeden řádek s počtem kroků cesty a počtem zdí, kterými cesta vede (hrad ani zvýrazněná cesta se nevypisuje).

`-i` nebo `--interactive` po nalezení cesty zobrazí textové rozhraní (založené na unixové knihovně *ncurses* [7]), které umožní interaktivně zobrazit průchod bílé paní hradem po nalezené nekratší cestě, včetně pohybů zvědů.

`-j` nebo `--json` zobrazí podobný výstup jako `--quiet`, pouze ve stojově čitelném formátu JSON [5] (využívá se při automatickém testování programu).

3.7.2 Zpracování argumentů z příkazové řádky

Na zpracování argumentů předaných na příkazové řádce použijeme modul `System.Console.GetOpt`, který je součástí standardní knihovny.

Začneme hlavičkou modulu:

```
module Banshee.Main(main) where
import System.IO
import System.Environment
import System.Exit
import System.Console.GetOpt
import Control.Applicative
import Control.Monad
import Data.Array
import Data.List (intersect)

import Banshee.Castle
import Banshee.CastleParser (parseCastle)
import Banshee.Navigate (navigate)
import Banshee.Interactive (showInteractive)
```

Definice přepínačů

Nyní si deklarujeme jednoduchý datový typ `Flag`, který reprezentuje jednotlivé přepínače, které můžeme dostat na příkazové řádce:

```
data Flag
  = HelpFlag
  | NotThroughFlag
  | ShowCastleFlag
  | QuietFlag
  | InteractiveFlag
  | JsonFlag
  deriving (Eq,Show)
```

V seznamu `options` uvedeme seznam všech možných nastavení s krátkými popisy:

```
options =
  [ Option ['h','?'] ["help"] (NoArg HelpFlag)
    "show the help"
  , Option ['n'] ["not-through"] (NoArg NotThroughFlag)
    "disable going through the walls"
  , Option ['s'] ["show-castle"] (NoArg ShowCastleFlag)
    "show the castle with highlighted path (default)"
  , Option ['q'] ["quiet"] (NoArg QuietFlag)
    "show just the minimal information about the found path"
  , Option ['i'] ["interactive"] (NoArg InteractiveFlag)
    "display a minimal interactive UI to show the found path"
  , Option ['j'] ["json"] (NoArg JsonFlag)
    "show the result as JSON"
  ]
```

Funkce `header` vytvoří hlavičku návodu na použití. Její argument `progname` je jméno programu (ve většině případů to bude `"banshee"`).

```
header progname = "Usage: " ++ progname ++ " [flags] castle-file"
```

Čtení předaných přepínačů

Na začátku `main` získáme seznam předaných argumentů z příkazové řádky pomocí `getArgs` a předáme jej funkci `getOpt` z knihovny `GetOpt`. První argument této funkce je hodnota `Permute`, která značí, že přepínače a jména souborů mohou být na příkazové řádce uvedeny v libovolném pořadí, druhým argumentem je seznam `options`, třetím je výsledek funkce `getArgs`.

```
main :: IO ()
main = do
  (flags,files,errs) <- getOpt Permute options <$> getArgs
```

Výsledkem funkce `getOpt` je trojice:

1. `flags` je seznam předaných přepínačů (`[Flag]`).
2. `files` je seznam předaných jmen souborů (`[String]`).
3. Pokud uživatel udělal chybu při zadávání přepínačů, v seznamu `errs` budou chybové hlášky (`[String]`).

Kontrola přepínačů

Ještě než se začneme zabývat předanými přepínači, vygenerujeme si návod k použití, který posléze můžeme zobrazit uživateli v případě, že udělal chybu. K tomu nám poslouží funkce `usageInfo` z knihovny `GetOpt`.

```
  progname <- getProgName
  let usage = usageInfo (header progname) options
```

Nejprve zkontrolujeme, jestli seznam `errs` neobsahuje nějakou chybu. Pokud ano, vypíšeme návod k použití a všechny chyby, načež program ukončíme.

```
  if not (null errs) then do
      hPutStrLn stderr usage
      hPutStrLn stderr $ concat errs
      exitFailure
    else return ()
```

Pokud si uživatel vyžádal nápovědu, zobrazíme ji a skončíme.

```
  if HelpFlag `elem` flags then do
      putStrLn usage
      exitSuccess
    else return ()
```

Přepínače `--quiet`, `--interactive`, `--json` a `--show-castle` se vzájemně vylučují, vždy lze použít nejvýše jeden.

```
  let exclusiveFlags = [QuietFlag,InteractiveFlag,JsonFlag,ShowCastleFlag]
  if (> 1) . length . intersect flags $ exclusiveFlags then do
      hPutStrLn stderr usage
      hPutStrLn stderr "The flags --quiet, --interactive, --json\
        \ and --show-castle are mutually exclusive (use at most one)"
      exitFailure
    else return ()
```

Nakonec zkontrolujeme, jestli jsme dostali právě jeden soubor; pokud ne, opět vypíšeme návod a skončíme.

```
  inputFile <- case files of
    [file] -> return file
    _ -> do
      hPutStrLn stderr usage
      hPutStrLn stderr "Expected one input file with castle"
      exitFailure
```

3.7.3 Výpočet trasy

Vstupní soubor přečteme a zparsujeme pomocí funkce `parseCastle`. Ta vrátí `Right` s výsledným hradem v případě úspěchu a `Left` s chybou v případě neúspěchu.

Přečtený hrad pak funkcí `sliceCastle` rozřežeme do `slices` a výsledek hledání cesty uložíme do `result`.

```
  txt <- readFile inputFile
  castle <- case parseCastle inputFile txt of
      Right c -> return c
      Left err -> do
        hPutStrLn stderr (show err)
        exitFailure

  let slices = sliceCastle castle
      result =  navigate castle slices thruWalls
      thruWalls = NotThroughFlag `notElem` flags
```

3.7.4 Zobrazení výsledku

Do proměnné `ui` na základě předaných přepínačů přiřadíme příslušnou zobrazovací funkci a vzápětí ji zavoláme s výsledkem hledání cesty, čímž končí akce `main`. Jednotlivé zobrazovací funkce popíšeme v následujících podsekcích.

```
  let ui :: Maybe [Loc] -> IO ()
      ui | QuietFlag `elem` flags       = showQuiet castle
         | InteractiveFlag `elem` flags = showInteractive castle slices
         | JsonFlag `elem` flags        = showJson castle
         | otherwise                    = showCastle castle
  ui result
```

3.7.5 Tiché zobrazení

Funkce `showQuiet`, kterou použijeme u přepínače `--quiet`, jednoduše vypíše jeden řádek v závislosti na tom, jesli byla cesta nalezena nebo ne.

```
showQuiet :: Castle -> Maybe [Loc] -> IO ()
showQuiet _ Nothing =
  putStrLn "No path found"
showQuiet castle (Just locs) =
  putStrLn . concat $ ["Found a path with ",show $ length locs," steps",
    " (",show $ countWalls castle locs," through walls)"]
```

Zde si také nadefinujeme funkci `countWalls`, která spočítá zdi nacházející se na dané cestě a kterou použijeme i v dalších funkcích.

```
countWalls :: Castle -> [Loc] -> Int
countWalls castle locs =
  length $ filter ((==Wall) . (castleFields castle !)) locs
```

3.7.6 Zobrazení ve formátu JSON

Výsledek ve formátu JSON zobrazíme podobně jako ve funkci `showQuiet`, pouze vypíšeme objekt ve tvaru `{ "steps": `*`počet_kroků`*`, "walls": `*`počet_zdí`*` }` pokud byla cesta nalezena nebo `{}` pokud nebyla.

```
showJson :: Castle -> Maybe [Loc] -> IO ()
showJson _ Nothing =
  putStrLn "{}"
showJson castle (Just locs) =
  putStrLn . concat $ ["{ \"steps\": ",show $ length locs,
    ", \"walls\": ",show $ countWalls castle locs," }"]
```

3.7.7 Zobrazení cesty v hradu

Funkce `showCastle` vypíše celý hrad a vyznačí v něm nalezenou trasu, samozřejmě pouze pokud byla nějaká cesta nalezena. Zdi jsou zaznačeny znaky '`X`', volná políčka znaky '`.`'. Cestu značíme znaky '`+`', cestu skrz zeď znakem '`~`'.

```
showCastle :: Castle -> Maybe [Loc] -> IO ()
showCastle _ Nothing =
  putStrLn "No path found"
showCastle castle (Just locs) = do
  forM_ [1..height] $ \y -> do
    forM_ [1..width] $ \x -> do
      putChar $ ary ! (x,y)
    putChar '\n'
  putStrLn . concat $ ["\n",show $ length locs," steps",
    " (",show $ countWalls castle locs," through walls)"]
  where

  ((1,1),(width,height)) = bounds $ castleFields castle

  fieldAry = fieldChar `fmap` castleFields castle
  fieldChar Free = '.'
  fieldChar Wall = 'X'

  ary = fieldAry // [((x,y),pathChar (x,y)) | (x,y) <- locs]
  pathChar (x,y) = case castleFields castle ! (x,y) of
      Free -> '+'
      Wall -> '~'
```

Vykreslení provádíme tak, že si vytvoříme pole `ary` o rozměrech hradu, zaplníme jej znaky a pak jej po řádcích vypíšeme.

Nejprve pole políček hradu, získané funkcí `castleFields`, převedeme pomocí funkce `fmap` na pole znaků `fieldAry`. Následně funkcí `//` změníme ta políčka, která se nachází na cestě (v seznamu pozic `locs`), na znak '`+`' nebo '`~`' v závislosti na tom, jestli na tomto políčku je zeď nebo ne.

3.7.8 Interaktivní zobrazení

Interaktivní zobrazení zajišťuje funkce `showInteractive` z modulu `Banshee.Interactive`. Její typ je:

```
showInteractive :: Castle -> [Slice] -> Maybe [Loc] -> IO ()
```

Tato funkce umožňuje krok po kroku procházet nalezenou cestu, včetně pohybujících se zvědů. Modul `Banshee.Interactive` zde nebudeme přetiskovat, jelikož je poměrně rozsáhlý (přibližně 180 řádků) a využívá knihovnu `ncurses` [15], jejíž popis je mimo rozsah této práce.

3.8 Závěr

Zde prezentované řešení soutěžní úlohy je téměř kompletní, podobně jako u Krunimíra jsme ale vynechali část s grafickým uživatelským rozhraním (editorem hradu). Na soutěži by nejspíše obdrželo přibližně 60 bodů z 90 (zbývajících 30 je za editor hradu) a možná nějaký bonus za interaktivní režim.

Nejdůležitější vlastností u podobných úloh je vždy rychlost. V tomto ohledu je naše řešení velmi dobré – všechny testovací soubory ze složek `test/data_ukazkova/` a `test/data_testovaci/` se na vývojovém počítači[6] vyřeší za necelé dvě sekundy (z toho asi 0.4 s zabere nejnáročnější soubor `data_testovaci/vstupy/4se_zvedy_skrz_zdi/vstup8.in`), spotřeba paměti se pohybuje okolo 10 MB.

[6] AMD Athlon™ 64 X2 Dual Core, frekvence procesoru 1000 MHz.

Program je zároveň poměrně krátký, obsahuje méně než 300 řádků kódu (bez modulu `Banshee.Interactive`), včetně přívětivého uživatelského rozhraní.

3.8.1 Zdrojové kódy

Soubory související s úlohou Bílá paní se nachází ve složce `banshee/` v repozitáři s prací. Zdrojové kódy všech modulů jsou uloženy ve složce `banshee/Banshee/`, testovací soubory ve složce `banshee/test/`. Část těchto souborů pochází ze soutěže a část byla vytvořena v rámci této práce. Konkrétně ve složce `test/extra/huge/` se nachází velmi velké hrady a ve složce `test/extra/cellular/` jsou hrady vytvořené pomocí celulárních automatů.

K automatickému testování korektnosti řešení je možné využít program `runtests.hs`, který čte názvy souborů s hrady a očekávané výsledky ze souboru `test/results.json` a kontroluje, jestli nalezená řešení těmto očekávaným výsledkům odpovídají.

Příloha A

Původní zadání soutěžních úloh

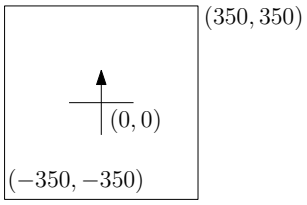
Na následujících stranách přetiskujeme původní zadání Ústředního kola Soutěže v programování z let 2010 a 2012.[1]

[1] Z technických důvodů musí vložený soubor PDF začínat na nové straně.

Zadání soutěžní úlohy: Želváci

Želvák Krunimír je velký myslitel. Zjistil, ze když si za krunýř přiváže trochu křídy, kreslí za sebou při svém plazení cestičku. I pojal plán nakreslit svoji podobiznu, samozřejmě včetně přesných detailů krunýře. Hned se dal pln nadšení do díla a práce mu šla pěkně od … tlapy.

„Co to děláš, dědečku," zeptal se jednou Krunimíra jeho vnuk Krunoslav. „Krešlím tady švou poďobižnu," odpověděl Krunimír. „Žačal jsem š ní, když tvůj tatík ještě nebyl na švětě, a ještě nemám ani krunýř," dodal smutně. „To už ji aši dokrešlit neštihnu…" Vnuk Krunoslav, znalec moderní techniky, mu však poradil: „Tak si nech napsat program, který ji nakreslí za Tebe." Protože se ale s tlapami a zobákem moc dobře neprogramuje, najali si želváci vás.



Napište aplikaci, která bude simulovat Krunimírovo chování. Krunimír se nalézá v ohradě, což je bílý čtverec s rohy v souřadnicích $(-350, -350)$ a $(350, 350)$. Na začátku je Krunimír ve středu ohrady (tj. na souřadnicích $(0, 0)$) a kouká na sever (tj. v kladném směru osy y), jak je nakresleno na obrázku vpravo.

Vaše aplikace dostane *program* pro Krunimíra. Tento program je posloupnost následujících příkazů:

- `left(úhel)` – Krunimír se natočí o daný úhel proti směru hodinových ručiček, přičemž úhel musí být násobek 90 (Krunimír tedy kouká na sever, východ, jih nebo západ).
- `right(úhel)` – Stejné jako `left`, jenom se Krunimír točí po směru hodinových ručiček.
- `pen(hodnota)` – Pokud je $hodnota \leq 0$, Krunimír odloží křídu. Pokud je $hodnota > 0$, Krunimír si křídu nasadí. Na začátku má Krunimír křídu odloženou.
- `forward(kolik)` – Krunimír popoleze vpřed ve směru svého natočení o *kolik* jednotek. Hodnota *kolik* může být kladná, nulová, nebo záporná (Krunimír leze vzad).

Všechna čísla v programu jsou celočíselná. Mezerové znaky (mezera, tabulátor, znaky konce řádky CR a LF) se v programu mohou vyskytovat kdekoliv (kromě uvnitř názvů funkcí a uvnitř čísel), ignorujte je.

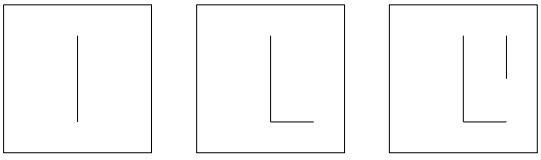
Po spuštění umožní vaše aplikace uživateli zvolit si soubor s programem pro Krunimíra a zadat počet tahů h. Můžete předpokládat, že program je bezchybný a že Krunimír nikdy nevyleze mimo svou ohradu. Vaším cílem je vytvořit obrázek, na kterém je černě nakresleno prvních h tahů, které Krunimír provedl. Tahem se rozumí každý příkaz `forward`, který Krunimír vykoná s nasazenou křídou. Pokud je zadaný počet tahů 0, vykreslete všechny Krunimírovy tahy.

Obrázek uložte v jednom z formátů PNG, BMP nebo GIF. To, že vykreslujete želví grafiku, neznamená, že váš program musí být stejně pomalý jako Krunimír :–)

Příklad: *Krunimírův program*

```
forward(200) pen ( 1 ) forward(
-400  )    right (90)
  forward(100)  left(90) pen (0)
forward(200)pen(1)forward(200)
```

Tahů: 1 *Tahů: 2* *Tahů: 3*



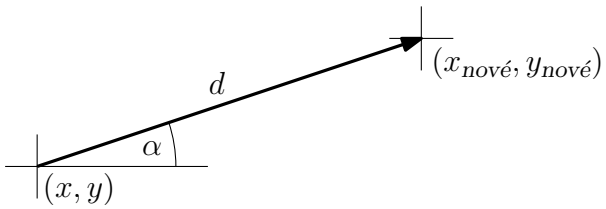
Za tuto základní část programu můžete získat 100 bodů. Kromě samotné funkčnosti se hodnotí i přehlednost zdrojového kódu a elegance řešení.

Dále můžete implementovat řadu rozšíření. Každé rozšíření se skládá z několika kroků, které musíte implementovat postupně jeden po druhém. Různá rozšíření můžete implementovat v libovolném pořadí. U každého rozšíření je uveden počet bodů, který získáte, pokud splníte všechny kroky v něm uvedené.

- **Televizní přenos [33 bodů]**
  1. Program bude interaktivně zobrazovat postup kreslení. Na začátku zobrazí obrázek s jedním tahem a počká na stisk klávesy nebo tlačítka. Poté zobrazí obrázek s dvěma tahy, počká na stisk klávesy nebo tlačítka, zobrazí obrázek s třemi tahy atd. Navíc zvýrazněte poslední provedený tah a vykreslete polohu a orientaci želváka.
  2. Předchozí funkci rozšiřte tak, aby bylo možno procházet i o tah zpět. Pokud je tedy vykreslen obrázek s t tahy, umožněte kromě zobrazení obrázku s $t+1$ tahy také zobrazení obrázku s $t-1$ tahy.

- **Lépe vybavený želvák [50 bodů]**
  1. Nový příkaz `color(č,z,m)` nastaví barvu křídy. Funkce má tři parametry, intenzitu červené, zelené a modré barvy v rozsahu 0 až 255. Hodnotu menší než nula považujte za nulu, hodnotu větší než 255 za 255.
  2. Rozšiřte příkaz `pen(šířka)` tak, že si po jeho provedení Krunimír nasadí křídu dané šířky. V případě, že je *šířka* ≤ 0, dojde k odložení křídy.
  3. Rozšiřte příkazy `left` a `right` tak, aby mohly jako parametr dostat libovolný celočíselný počet stupňů. Pokud stojí želvák na (x, y) a je otočený α stupňů od východního směru, tak pohybem vpřed o d jednotek se dostane na souřadnice

$$x_{nové} = x + d \cdot \cos(\alpha)$$
$$y_{nové} = y + d \cdot \sin(\alpha)$$



  Pozor na to, že ve většině jazyků je třeba zadat úhel v radiánech. Přepočet je

$$radiány = stupně * \pi/180 \approx stupně/57.295779513082320877 21$$

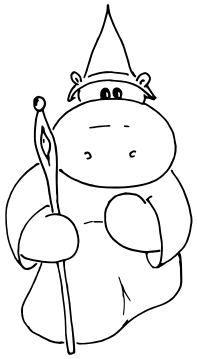
  Pozici želváka si musíte pamatovat jako desetinné číslo, ale při vykreslování můžete zaokrouhlovat.
- **Chytřejší želvák [100 bodů]**
  1. Program může obsahovat příkaz `repeat (počet) { nula a více příkazů }`. Při provedení příkazu `repeat` dojde k provedení příkazů v jeho těle *počet*-krát (pro počet ≤ 0 se příkaz neprovede vůbec).
  2. Program může obsahovat příkaz `if (číslo) { nula a více příkazů }`. Při provedení příkazu `if` dojde k provedení příkazů v jeho těle pouze v případě, že *číslo* je větší než nula.
  3. Program může obsahovat nové funkce. Ty se definují pomocí `define novafunkce() { nula a více příkazů }`. Jméno nové funkce se skládá jenom z malých písmen anglické abecedy. Novou funkci lze použít kdekoliv *po* její definici a funkce je možné volat rekurzivně, tj. funkce smí volat sama sebe. Definice funkcí nemohou být vnořené, tj. uvnitř definice funkce není možné definovat další funkci.
  4. Nově definované funkce mohou mít libovolný počet parametrů. Taková funkce se definuje jako `define f(parametry) { nula a více příkazů }`. Parametrů je libovolný pevný počet, jsou celočíselné, jejich jména se skládají z malých písmen anglické abecedy a jsou oddělená čárkou. Uvnitř funkce s parametry můžete jako argument libovolné funkce (a jako argument příkazů `if` a `repeat`) použít `a`, `a+b`, `a-b`, `a*b`, `a/b`, kde `a` a `b` jsou buď čísla nebo parametry aktuální funkce. Dělení je celočíselné.
     Příklad:
```
define spirala(delka) {
  if (delka) { forward(delka) right(90) spirala(delka-4) }
} pen(1) spirala(100)
```

  5. Uvnitř funkce s parametry můžete jako argument libovolné funkce (a jako argument příkazů `if` a `repeat`) použít jakýkoliv výraz, který obsahuje celá čísla, parametry aktuální funkce, kulaté závorky a operátory `+ - * /`. Priority operátorů jsou stejné jako v matematice.
- **Želváčí farma [50 bodů]**

  Toto rozšíření je možné implementovat až po splnění prvních 4 kroků z rozšíření Chytřejší želvák.
  1. Příkaz `split { nula a více příkazů }` vytvoří klon želváka na stejném místě a se stejnou orientací. Původní želvák pokračuje v provádění dalšího příkazu za příkazem `split`, klon provede příkazy uvedené v těle příkazu `split`.
     Pokud je želváků více, nijak se navzájem neovlivňují a provádí své tahy paralelně. Tedy všichni želváci provedou svůj první tah, teprve potom provedou všichni svůj druhý tah a tak dál. V rámci jednoho tahu není pořadí želváků definované.

Soutěžní úloha Bílá paní

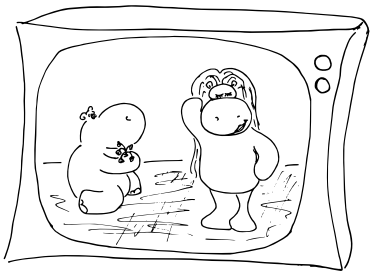
„Neboj se, žádní duchové přece nejsou.“
„Já se přece nebojím. Jenom se mi zdálo, že jsem támhle něco zahlédl. . . “
„. . . “
„Pomóc, bílá paní, utíkej kdo můžeš!“



Bílá paní s bezhlavým rytířem společně sledovali bezhlavě prchající turisty. „A nenudíte se sama dole v podhradí,“ ptal se rytíř, „když všechny návštěvníky vyděsíte tak, že nestačí utíkat?“ „Ale ne, jedni mi tam nechali televizor a já na něm sleduji ty, jak se to jmenuje . . . telenovely. Jenom teď musím ten televizor zase najít a v mém věku už není procházení zdí tak jednoduché jako zamlada.“

Pomozte bílé paní najít cestu k televizoru. Musíte najít cestu nejkratší, protože její oblíbená telenovela právě začíná. Podhradí, ve kterém se televizor nachází, má podobu bludiště o rozměru až 80×80 políček. Kromě jedné bílé paní a jednoho televizoru se v bludišti nachází prázdná políčka, zdi a zvědové. Zvědové jsou zvědavci, kteří se snaží objevit bílou paní. Kdyby se to nějakému povedlo, musela by ho bílá paní vyhnat a nestihla by kvůli tomu svůj oblíbený pořad, proto se bílá paní musí všem zvědům vyhnout.

Popis formátu bludiště



Bludiště je uloženo v textovém souboru, jehož řádky jsou odděleny dvojicí znaků CR a LF. Na prvním řádku se nachází mezerou oddělená dvě přirozená čísla $S \leq 80$ a $\check{\mathrm{R}} \leq 80$ – počet sloupců a řádků bludiště. Na každém z následujících $\check{R}$ řádků se nachází popis S políček bludiště, která mohou být následující:

- `.`: volné políčko
- `X`: zeď
- `&`: bílá paní, v celém bludišti je přesně jedna
- `#`: televizor, v celém bludišti je přesně jeden
- `@`: zvěd. Každý zvěd má vlastní trasu, po které se pohybuje. Trasa je složena z nejvýš osmi pohybů a po jejím provedení je zvěd vždy zpátky na políčku, ze kterého vyšel. Trasa je v souboru uvedena ihned za zvědem a skládá se z pohybů:
  - `<`: vlevo
  - `>`: vpravo
  - `^`: nahoru
  - `v`: dolů

Příklad bludiště: *soubor s bludištěm*

```
5 5
&X...
.X.@^>v<.
.X.X.
@>><<..X.
..X#.
```

rozfázované pohyby zvědů

```
&X...   &X.@.   &X..@   &X...
.X.@.   .X...   .X...   .X..@
.X.X.   .X.X.   .X.X.   .X.X.
@..X.   .@.X.   ..@X.   .@.X.
..X#.   ..X#.   ..X#.   ..X#.
```

Pohyb v bludišti

Bílá paní se v bludišti v každém kroku pohne v jednom ze čtyř směrů – vlevo, vpravo, nahoru či dolů. Pohyb je v každém kroku povinný, není možné zůstat stát. Cílem bílé paní je dostat se v co nejmenším počtu kroků na políčko s televizorem. Bílá paní nesmí být objevena zvědem: k tomu dojde buď tak, že se bílá paní a zvěd nachází najednou na jednom políčku, nebo si v jednom kroku bílá paní a zvěd vymění pozice („projdou skrz sebe“, tj. jedním krokem se z pozice `&@` dostanou do pozice `@&`, ať už ve vodorovném nebo svislém směru).

Zadání úlohy

Vaším cílem je naprogramovat hledání nejkratší cesty v bludišti, nebo editor bludiště, nebo obojí. Pokud budete pracovat na obou úkolech, naprogramujte je do jediného programu.

1) Hledání nejkratší cesty v bludišti [70% bodů]

Váš program načte bludiště z uživatelem zvoleného souboru. Můžete přitom předpokládat, že soubor obsahuje synatkticky korektní bludiště zapsané ve výše popsaném formátu.

Postupně přidávejte funkce v pořadí, v jakém jsou uvedeny:

- V bludišti, ve kterém nejsou zvědové, najděte libovolnou nejkratší cestu od bílé paní k televizoru. Bílá paní nesmí nikdy vstoupit na políčko se zdí. Zobrazte tuto cestu a její délku, případně vypište, že taková cesta neexistuje.
  - Nejkratší cestu zobrazte interaktivně – zobrazujte postupně jednotlivé kroky bílé paní a umožněte uživateli přejít o krok vpřed a o krok vzad.
- V bludišti, ve kterém nejsou zvědové, najděte libovolnou nejkratší cestu k televizoru, na které musí bílá paní co nejméně krát projít zdí. Důležitější je projít co nejmenším počtem zdí, takže libovolně dlouhá cesta bez průchodu zdí je lepší než jakkoliv krátká cesta, na které musí bílá paní jednou projít zdí. Jako průchod zdí se počítá každý krok, který končí na políčku se zdí. Nalezenou cestu včetně délky zobrazte, případně vypište, že taková cesta neexistuje.
  - Nejkratší cestu zobrazte interaktivně.
- V bludišti se zvědy najděte libovolnou nejkratší cestu od bílé paní k televizoru. Na cestě nesmí bílou paní objevit žádný zvěd a bílá paní nemůže procházet zdí. Nalezenou cestu včetně délky zobrazte, případně vypište, že taková cesta neexistuje.
  - Nejkratší cestu zobrazte interaktivně, včetně pohybu zvědů.
- V bludišti se zvědy najděte libovolnou nejkratší cestu k televizoru, na které musí bílá paní co nejméně krát projít zdí. Stejně jako v předchozím případě je důležitější projít co nejmenším počtem zdí. Na cestě nesmí bílou paní samozřejmě objevit žádný zvěd. Nalezenou cestu včetně délky zobrazte, případně vypište, že taková cesta neexistuje.
  - Nejkratší cestu zobrazte interaktivně, včetně pohybu zvědů.

2) Editor bludiště [30% bodů]

Postupně přidávejte funkce v pořadí, v jakém jsou uvedeny:

- Úprava bludišť bez zvědů:
  - Načtěte a zobrazte bludiště.
  - Umožněte vytvořit nové prázdné bludiště zadaných rozměrů. Bílá paní je vlevo nahoře, televizor vpravo dole.
  - Dovolte editaci bludiště – přesun televizoru na vybrané políčko, přesun bílé paní na vybrané políčko, vytváření a rušení zdí.
  - Umožněte uložení bludiště.
- Úprava bludišť obsahujících zvědy. Kromě právě uvedených funkcí naprogramujte následující:
  - Umožněte editaci zvědů – vytvoření nového zvěda, odstranění existujícího zvěda, přesun zvěda (včetně jeho trasy) na vybrané políčko, úprava trasy zvěda. Při úpravě trasy zkontrolujte, že se po jejím provedení vrátí zvěd na počáteční pozici.
  - Dovolte uživateli zvolit režim, ve kterém se interaktivně zobrazují pohyby zvědů v bludišti. Uživatel může přejít o krok vpřed a o krok zpět.

Rejstřík

Literatura

[1] *Soutěžní úloha Bílá paní.* Zadání Ústředního kola ČR Soutěže v programování 2012. http://stv.cz/sp/mcr12zp.pdf.

[2] *Zadání soutěžní úlohy: Želváci.* Zadání Ústředního kola ČR Soutěže v programování 2010. http://stv.cz/sp/mcr2010z.pdf.

[3] *Haskell 2010 Language Report.* Technická zpráva, Haskell Comittee, 2010. http://www.haskell.org/onlinereport/haskell2010/.

[4] Bringert, Bjorn: *HackageDB: The gd package.* http://hackage.haskell.org/package/gd, [15.2.2013].

[5] Crockford, Douglas: *The application/json media type for javascript object notation (json).* 2006. http://tools.ietf.org/html/rfc4627.

[6] Ford, Bryan: *Parsing expression grammars: a recognition-based syntactic foundation.* V *ACM SIGPLAN Notices*, svazek 39, strany 111–122. ACM, 2004. http://pdos.csail.mit.edu/papers/parsing%3Apopl04.pdf.

[7] Free Software Foundation: *Announcing ncurses 5.9 – GNU Project – Free Software Foundation (FSF).* http://www.gnu.org/software/ncurses/ncurses.html.

[8] Free Software Foundation: *GNU Bison – The Yacc-compatible Parser Generator.* http://www.gnu.org/software/bison/manual/index.html.

[9] Hudak, Paul, John Hughes, Simon Peyton Jones a Philip Wadler: *A history of Haskell: being lazy with class.* V *Proceedings of the third ACM SIGPLAN conference on History of programming languages.* ACM, 2007. http://www.cs.tufts.edu/~nr/cs257/archive/simon-peyton-jones/history.pdf.

[10] Hughes, John: *Why functional programming matters.* The computer journal, 32(2):98–107, 1989. http://cs.simons-rock.edu/cmpt312/whyfp.pdf.

[11] Jones, Simon Peyton (editor): *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003. http://www.haskell.org/onlinereport/.

[12] Launchbury, John, Simon Peyton Jones *a kol.*: *Lazy functional state threads.* ACM SIGPLAN Notices, 29(6):24–35, 1994. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.3299.

[13] Leijen, Daan: *Parsec, a fast combinator parser.* University of Utrecht, Dept. of Computer Science. http://legacy.cs.uu.nl/daan/download/parsec/parsec.pdf, [15.2.2013].

[14] Lipovača, Miran: *Learn You a Haskell for Great Good!: A Beginner's Guide.* No Starch Press, 2011. http://learnyouahaskell.com/.

[15] Millikin, John: *HackageDB: ncurses-0.2.3*, 2012. http://hackage.haskell.org/package/ncurses.

[16] Santos, André Luís de Medeiros: *Compilation by Transformation in Non-Strict Functional Languages*. disertace, Univerzity of Glasgow, 1995.

[17] Wikipedia: *Sierpiński arrowhead curve — Wikipedia, The Free Encyclopedia*, 2012. `http://en.wikipedia.org/w/index.php?title=Sierpi%C5%84ski_arrowhead_curve&oldid=495447131`, [Online; accessed 2-March-2013].

[18] Wikipedia: *Gosper curve — Wikipedia, The Free Encyclopedia*, 2013. `http://en.wikipedia.org/w/index.php?title=Gosper_curve&oldid=540883222`, [Online; accessed 2-March-2013].

[19] Wikipedia: *Hilbert curve — Wikipedia, The Free Encyclopedia*, 2013. `http://en.wikipedia.org/w/index.php?title=Hilbert_curve&oldid=540435808`, [Online; accessed 2-March-2013].

[20] Wikipedia: *Koch snowflake — Wikipedia, The Free Encyclopedia*, 2013. `http://en.wikipedia.org/w/index.php?title=Koch_snowflake&oldid=541215024`, [Online; accessed 2-March-2013].