Interpret jazyka Scheme - Jakub Arnold

Tématem zápočtového programu je implementace interpretu jazyka Scheme, resp. jeho podmnožiny. Program umí fungovat ve dvou módech, interaktivně v příkazové řádce, a nebo v dávkovém režimu, kdy vyhodnotí celý soubor najednou.

Kompilace a spuštění

Pro spuštění je potřeba mít nainstalovaný GHC Haskell (testováno na verzi 7.8.3, ale měla by fungovat i libovolná jiná). Poté stačí již v adresáři projektu spustit instalaci závislostí

```
cabal install --only-dependencies
```

zkompilovat zdrojový kód

```
cabal build
```

a spustit.

```
./dist/build/Scheme/scheme
```

Pro spuštění je potřeba předat jeden ze dvou flagů na příkazové řádce. Pro dávkový režim stačí zadat název souboru, např.

```
./dist/build/Scheme/scheme examples/1_simple.lisp
```

a pro interaktivní režim (REPL) slouží přepínač `-i`.

```
./dist/build/Scheme/scheme -i
```

Program také umožnuje zobrazit detailnější výstup během vyhodnocování, k čemuž slouží přepínač `-v` (je nutno ho zadat jako poslední.)

```
./dist/build/Scheme/scheme examples/1_simple.lisp -v
```

resp.

```
./dist/build/Scheme/scheme -i -v
```

Rozsah implementace

Program implementuje podstatnou podmnožinu jazyka scheme, a některé zabudované funkce. Dostupné jsou následující speciální formy

- define
- lambda
- let
- if
- undef

a základní operátory pro práci s celými čísly (desetinná čísla nejsou podporována.) V základní podobě je tedy možné pouze provádět celočíselné operace, pričemž ale libovolné rozšíření na větší část jazyka by nemělo být složité (viz. další sekce.) Pro aritmetické operace je dostupné `+ - * / =`, kde `/` představuje celočíselné dělení.

Forma `define` dovoluje pouze syntaxi typu `(define nazev hodnota)`, kde hodnota může být lambda funkce. Zkrácená syntaxe typu `(define (nazev parametry) hodnota)` není podporována.

Soubory jsou zpracovávány po jednotlivých výrazech, přičemž každý výraz může obsahovat `define`, čímž změní *vyhodnocovací prostředí* pro následující příkazy, např.

```
(define x 3)
(define y (+ x 1))
(+ x y)
```

se postupně vyhodnotí na následující posloupnost

```
#t
#t
7
```

kde je třeba podotknout, že hodnota výrazu define je vždy `#t` (tedy symbol pro booleovskou hodnotu `True`.)

To samé je možné zkusit i v interaktivním režimu, kde navíc existuje speciální forma `undef`, sloužící pro zrušení definice, viz např. (`>` značí příkazový prompt.)

```
> (define x 3)
#t
> (define y (+ x 1))
#t
```

```
> (+ x y)
7
> (undef x)
undefined
> y
4
> x
x
```

Na tomto příkladu je názorně vidět, že vyhodnocení hodnoty pro proměnnou `y` proběhne již v místě `define`, a její hodnota se tedy zachová, i poté co `x` ztratí svojí hodnotu (nedefinovaná proměnná se vyhodnotí sama na sebe jako atom.)

Pokud je zadán neplatný výraz, program okamžitě skončí s chybovou hláškou značící v čem nastala chyba.

Implementace

Program je rozdělen na několik logických modulů:

- Parser - Samotný parser jazyka Scheme, velmi jednoduše rozšířitelný.
- Parser.Combinators - Implementace jednoduchého monadického parseru.
- Evaluator - Vyhodnocovací logika nad výslednou abstraktní reprezentací syntaxe jazyka Scheme. Obsahuje veškeré zabudované funkce a speciální formy.
- Types - Definice základních typů, se kterými zbytek programu pracuje.
- Printer - Formátovací funkce, které se používají pro uživatelský výstup.
- REPL - Implementace interaktivního módu, aka. Read Eval Print Loop.

Parser a Parser.Combinators

Samotné jádro parseru je implementováno na základě dvou článků *Monadic Parsing in Haskell*.

- https://www.cs.nott.ac.uk/~gmh/monparsing.pdf
- https://www.cs.nott.ac.uk/~gmh/pearl.pdf

Celé jádro tkví v monádě `Parser a`, která reprezentuje parser hodnoty typu `a`. Konkrétně jsem se rozhodl upravit původní verzi aby bylo možné zjistit jaká chyba při parsování nastala (což je něco co ani jeden z článků neřeší) a proto jsem zvolil následující typ

```
data Parser a = Parser (String -> ParseResult a)
type ParseResult a = Either String (a, String)
```

kde místo nedeterministického `[(a, String)]` používám raději `Either String (a, String)`, kde `Left` obsahuje konkrétní chybovou hlášku, a `Right` obsahuje dvojici s výslednou naparsovanou hodnotou a zbytkem nepoužitého stringu. Každý parser je tedy funkce která jako parametr bere string, a vrací buď chybu, nebo výslednou hodnotu a zbylou nepoužitou část stringu.

Implementace instance monády je poté prakticky triviální

```
instance Monad Parser where
  return x = Parser (\s -> Right (x, s))
  (Parser p) >>= f = Parser (\x -> do
                                (a, s) <- p x
                                let (Parser q) = f a
                                q s)
```

kde `return` je vytvoří parser, který pro každý vstup rovnou vrací zadanou hodnotu, a nic ze vstupu nespotřebuje, a `bind` (`>>=`) vytvoří parser, který umožnuje parsovat v závislosti na kontextu. Nejprve se spustí první parser, z jehož výsledné hodnoty se vytvoří parser nový, který se poté aplikuje na zbytek stringu z aplikace prvního parseru.

Zde je např. parser `bracket`, který parsuje výraz uvnitř závorek:

```
bracket :: Char -> Char -> Parser a -> Parser a
bracket left right middle = do
  char left
  m <- middle
  char right
  return m
```

Síla monadického parsování je tedy převážně v jednoduché kompozici parserů dohromady, kde zpravidla stačí jenom popsat jak má vypadat výsledný výraz (za pomocí jiných parserů.), a implementace monády se už postará o to, aby se všechny parametry předaly tak jak mají.

Parser pro konkrétní výraz vytvoříme jednoduše, např. pokud bychom chtěli řetězec `"(123)"` parsovat jako *číslo uvnitř závorek*, stačí použít následující parametry

```
bracket '(' ')' number
```

kde `number` je opět parser čísla, s prakticky triviální implementací (`many1` parsuje *alespoň jednu hodnotu daného parseru.*)

```
number :: Parser Int
number = do
  digits <- many1 digit
  return $ foldl (\acc x -> acc * 10 + x) 0 digits
```

Evaluator

Vyhodnocování jazyka Scheme je až překvapivě jednoduché, ikdyž mi při implementaci trvalo, než jsem našel vhodnou úroveň abstrakce. Důležitý je převážně typ `AST`, reprezentující abstraktní syntaxi jazyka Scheme.

```
data AST = ASTList [AST]
         | ASTLambda [AST] AST
         | ASTNumber Int
         | ASTAtom String
         deriving (Show, Eq)
```

Každá hodnota je buď list, lambda funkce, celé číslo, nebo atom. Možná stojí za zmínku že jsou lambda funkce reprezentovány jako zvláštní typ, a ne jako list. K této implementaci jsem došel převážně kvůli uživatelsky přívětivějším chybám při parsování, kde lambda funkce mohou vést na poměrně složitou a těžce laditelnou syntaxi, kde sebemenší chyba je těžká najít, pokud člověk přesně neví kde nastala (implementovaný parser bohužel neudržuje pozici o aktuálně zpracovávané řádce a sloupcí.)

Důležitý je také typ `Env`

```
type Env = [(String, AST)]
```

který reprezentuje prostředí, ve kterém se výrazy vyhodnocují, a udržuje hodnoty lokálních a globálních proměnných. Vyhodnocování vždy začíná s prázdným prostředím `Env`.

Pravidla jsou následující:

- Atom se vyhodnotí na svojí hodnotu v `Env`, případně sám na sebe, pokud taková hodnota neexistuje.
- Lambda funkce se vyhodnotí sama na sebe.
- Celé číslo se vyhodnotí samo na sebe.
- List se vyhodnotí jako volání funkce následujícím způsobem:
    0. Prázdný list je vždy vyhodnocen jako chybné volání funkce.
    1. Vždy se první vyhodnotí hlava, což umožnuje dynamicky rozlišit, jaká funkce se zavolá, např. `((if #f foo bar) 1)` zavolá funkci `bar` s parametrem `1`.

2. Pokud je vyhodnocená hlava číslo, nastává chyba.
3. Pokud je vyhodnocená hlava atom, musí se jednat o speciální formu, nebo vestavěnou funkci (uživatelsky definované funkce se vyhodnotí na příslušnou lambda funkci.)
4. Pokud je vyhodnocená hlava lambda funkce, zavolá se s příslušnými parametry.

Vyhodnocování vestavěných funkcí je vesměs triviální, za poznámku stojí pouze speciální formy, kde každá má svoje zvláštní vyhodnocovací pravidla (viz. komentáře ve zdrojovém kódu v modulu `Evaluator`.)

Možnosti rozšíření

Přestože program neimplementuje celý jazyk Scheme, obsahuje jeho podstatnou podmnožinu, přičemž libovolné rozšíření by mělo být jednoduché.

Podpora pro další formy, např. `cond`, je pouze otázka napsání příslušného parseru a vyhodnocovací strategie (případně parsovat `cond` jako list a kontrolovat chyby až při vyhodnocování.) Vyhodnocovací strategie pro nové formy lze přidat jednoduše úpravou funkcí `Evaluator.isSpecialForm` a `Evaluator.evalSpecialForm`.

Pravděpodobně nejsložitější rozšíření by bylo přidat variadické vestavěné funkce, např. `apply` nebo variadické `+` (umožňující např. `(+ 1 2 3 4 5 6 7 8 9)`.) Zde by bylo nutné upravit typ `Evaluator.BuiltinFunction` tak, aby dovolil různé arity funkcí, a poté upravit příslušné vyhodnocovací funkce, převážně `Evaluator.evalBuiltin`.

Další možností rozšíření by mohla být např. forma `quote`, umožnující práci se seznamy. Zde se opět nejedná o nic moc složitého, neboť samotný jazyk již seznamy obsahuje. Dokonce i podpora pro makra.

Další možné rozšíření by mohlo být práce s reálnými čísly, kde by opět stačilo přidat příslušný parser pro reálná čísla, položku do abstraktního typu syntaxe, a příslušné vestavěné funkce.

Závěr

Toto téma jsem si zvolil převážně proto, že jsem si chtěl zkusit napsat parser bez použití knihoven, které všechnu práci udělají samy. Proto jsem dal většinu energie do toho, aby parser byl jednoduše použitelný (chybové hlášky) a také rozšiřitelný na další syntaktické konstrukce.

Vyhodnocování jazyka Scheme se řídí podle jednoduchých pravidel, které vedou na velmi přímou a přehlednou implementaci. Proto jsem se taky nepouštěl do implementaci exotičtějších forem, jako např. formy `cond`, `quote`, atd. Přidání

další funkcionality již nepřináší implementačně nic nového ani zajímavého, ani z hlediska parsování syntaxe, ani z hlediska strategie vyhodnocování výrazů. Možná by stálo za zmínku líné vyhodnocování, které by mohlo vést na zajímavou implementaci, ale pokud vím není standardní součástí jazyka Scheme.