Překladač jazyka Zyba

Staticky typovaný jazyk kompilovaný do PHP

Zyba language compiler

Language with static typing transpiled into PHP

Středoškolská odborná činnost, rok 2022
Richard Blažek
Gymnázium Brno, třída Kapitána Jaroše 14

Prohlášení

Prohlašuji, že jsem svou závěrečnou maturitní práci vypracoval samostatně a použil jsem pouze prameny a literaturu uvedené v seznamu bibliografických záznamů.

Prohlašuji, že tištěná verze a elektronická verze závěrečné maturitní práce jsou shodné.

Nemám závažný důvod proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších předpisů.

V Brně dne 20. února 2022 .....

Poděkování

Tímto bych chtěl poděkovat Matěji Žáčkovi za odborné vedení práce.

Anotace

Práce se zabývá navržením jazyka Zyba a implementací překladače tohoto jazyka do PHP, což by mělo umožnit používání tohoto jazyka na všech webhostinzích, které podporují PHP skripty. Rovněž bude možné vyvíjet část projektu v Zybě a část v PHP a používat funkce z jednoho jazyka ve druhém.

Klíčová slova

programovací jazyk; překladač; transpiling; webová aplikace; PHP; Zyba

Annotation

The thesis is concerned with the design of the Zyba language and implementing its compiler. The compiler generates PHP code, making it possible for the language to be used on all webhostings where PHP is supported. Also, it will be possible to combine Zyba and PHP when developing web applications and use functions from one of the languages in the other one.

Keywords

programming lanugage; compiler; transpiling; web application; PHP; Zyba

Obsah

1 Úvod

Cílem této práce je navrhnout programovací jazyk Zyba, který by měl umožnit psaní přehlednějšího kódu než PHP, a implementovat překladač ze Zyby do PHP, aby bylo možné programy v Zybě používat na všech na všech serverech s podporou PHP skriptů. Navrhl jsem Zybu jako staticky typovaný jazyk, protože překlad z jednoho dynamicky typovaného jazyka do druhého by umožnil provádět při překladu pouze syntatickou kontrolu. Vyhodnocení typů by mohlo proběhnout až za běhu programu a Zyba by tak představovala jen alternativní syntaxi pro PHP. Zybu jsem navrhl jako jazyk jednoduchý na naučení, ale s dostatečnou funkcionalitou pro psaní webových stránek.

Překladač jsem se rozhodl napsat v jazyce Haskell, protože umožňuje psát velmi stručné a přehledné programy. K čitelnosti programu přispívá jednak syntaxe jazyka, v němž se struktura programu vyjadřuje formátováním zdrojového kódu a ne oddělovači, jednak jeho striktní dodržování funkcionálního paradigmatu, které vyžaduje, aby funkce byly referenčně transparentní (tzn. bez vedlejších efektů). Navíc jeho typový systém obsahuje algebraické datové typy a umožňuje zápis rekurzivních typů, což se u překladače hodí například na zápis syntaktického stromu.

2 Návrh jazyka

2.1 Syntaxe

Syntaxe Zyby je stejně jako u řady dalších jazyků (např. C++, Java, C#, JavaScript) odvozená z jazyka C, aby byla blízká ostatním programátorům, ale liší se v řadě detailů. Všechny operátory jsou binární, zleva asociativní a mají stejnou prioritu. Práce s poli a slovníky (tj. jejich vytváření a přístup k prvkům a jejich změna) není záležitostí speciální syntaxe, ale provádí se zabudovanými metodami až na úrovni sémantiky. Volání funkce je vyjádřeno hranatými závorkami. Díky těmto změnám je syntaxe jednoznačná, aniž by musel uživatel používat středníky. Středníky a čárky jsou považovány za bílé znaky, takže je možné je používat, kde to programátor uzná za vhodné pro přehlednost kódu. Komentáře jsou jednořádkové a začínají křížkem (`#`).

Následuje popis prvků syntaxe a její formální popis v rozšířené Backus-Naurově formě.

2.1.1 Deklarace

Soubor v Zybě se skládá z deklarací. Každá deklarace buď přiřadí určité globální konstantě hodnotu určitého výrazu, nebo importuje deklarace z jiného souboru. Importují se přitom pouze ty deklarace, které byly uvozeny slovem `export`

⟨*file*⟩ ::= ((`"export"`)? ⟨*declaration*⟩)*

⟨*declaration*⟩ ::= ⟨*assignment*⟩ | ⟨*import*⟩

⟨*assignment*⟩ ::= ⟨*name*⟩ `"="` ⟨*expression*⟩

Importovat lze jak ze Zyby tak i z PHP. Import ze Zyby vyžaduje cestu k souboru, který má být importován, a název jmenného prostoru, do kterého budou importované hodnoty spadat. Import z PHP to vyžaduje rovněž, ale před názvem jmenného prostoru musí být napsáno slovo `php` a za cestou k souboru musí následovat záznam se jmény a typy importovaných hodnot. Záznam tvoří závorky a v nich několik dvojic názvů (tj. jména importovaných hodnot) a výrazů (tj. jejich typy); mezi názvem a výrazem je vždy dvojtečka.

⟨*import*⟩ ::= ⟨*import-zyba*⟩ | ⟨*import-php*⟩

⟨*import-zyba*⟩ ::= `"import"` ⟨*name*⟩ ⟨*literal-text*⟩

⟨*import-php*⟩ ::= `"import php"` ⟨*name*⟩ ⟨*literal-text*⟩ ⟨*record*⟩

⟨*record*⟩ ::= `"("` (⟨*name*⟩ `":"` ⟨*expression*⟩)* `")"`

2.1.2 Výrazy

Výraz tvoří jeden či více podvýrazů oddělených binárními operátory. Podvýraz tvoří jednotka, kterou můžou následovat volání funkce a přístupy k prvkům záznamu nebo jmenného prostoru. Volání funkce tvoří několik výrazů v hranatých závorkách; přístup k prvku tvoří tečka následovaná jménem. Zabudované metody se volají způsobem, který kombinuje syntaxi přístupu a syntaxi volání funkce: `argument1.metoda[argument2 ... argument`$_N$, přičemž mají-li jen jeden argument, je možné závorky vynechat.

⟨*expression*⟩ ::= ⟨*call*⟩ (⟨*operator*⟩ ⟨*call*⟩)*

⟨*subexpression*⟩ ::= ⟨*unit*⟩ (⟨*call*⟩ | ⟨*access*⟩)*

⟨*call*⟩ ::= `"["` ⟨*expression*⟩* `"]"`

⟨*access*⟩ ::= `"."` ⟨*name*⟩

Jednotek je několik druhů: výraz v závorkách; literál celého čísla, reálného čísla, logické hodnoty či text; záznam nebo lambda funkce.

⟨*unit*⟩ ::= `"("` ⟨*expression*⟩ `")"`

⟨*unit*⟩ ::= ⟨*literal-int*⟩ | ⟨*literal-real*⟩ | ⟨*literal-bool*⟩ | ⟨*literal-text*⟩

⟨*unit*⟩ ::= ⟨*record*⟩

⟨*unit*⟩ ::= ⟨*lambda*⟩

2.1.3 Funkce a příkazy

Lambda funkce začíná slovem `fun`. Následují argumenty funkce v hranatých závorkách, výraz specifikující typ navrácené hodnoty a blok, který tvoří několik příkazů ve složených závorkách. Argumenty funkce, pokud nějaké jsou, se zapisují po skupinách, z nichž každá se skládá z názvů jednoho či více argumentů následovaných dvojtečkou a výrazem specifikujícím typ těchto argumentů.

⟨*lambda*⟩ ::= `"fun" "["` ⟨*arguments*⟩ `"]"` ⟨*expression*⟩ ⟨*block*⟩

⟨*arguments*⟩ ::= (⟨*name*⟩+ `":"` ⟨*expression*⟩)*

⟨*block*⟩ ::= `"{"` ⟨*statement*⟩* `"}"`

Příkazem může být výraz, jenž má být vyhodnocen, přiřazení, příkazy `return` a `if` nebo cykly `while` či `for`. Příkaz `return` je slovo `return` následované výrazem, jenž má být navrácená hodnota. Příkaz `if` začíná slovem `if` následovaným výrazem (podmínkou) a blokem příkazů; za blokem můžou následovat další části, jež začínají slovy `else if` a rovněž pokračují podmínkou a blokem příkazů; poté může následovat slovo `else` následované blokem, který se provede, pokud žádná z předchozích podmínek nebyla splněna. Cyklus `while` začíná slovem `while` následovaným výrazem (podmínkou) a blokem příkazů.

⟨*statement*⟩ ::= ⟨*expression*⟩ | ⟨*assignment*⟩ | ⟨*if*⟩ | ⟨*while*⟩ | ⟨*for*⟩

⟨*return*⟩ ::= `"return"` ⟨*expression*⟩

⟨*if*⟩ ::= `"if"` ⟨*condition*⟩ ⟨*block*⟩ (`"else if"` ⟨*condition*⟩ ⟨*block*⟩)* (`"else"` ⟨*block*⟩)?

⟨*while*⟩ ::= `"while"` ⟨*condition*⟩ ⟨*block*⟩

Cyklus `for` lze použít pro iteraci přes pole, slovník nebo celé číslo. Zapisuje slovem `for`, po němž následují jména dvou konstant, z nichž první bude obsahovat index současného prvku a druhá jeho hodnotu, oddělovač `:`, výraz, přes jehož hodnotu se iteruje, a blok příkazů. První jméno je nepovinné, v tom případě se hodnota indexu nebude do žádné konstanty ukládat. Iterace přes celé číslo znamená, že hodnota současného prvku bude postupně nabývat hodnot od toho čísla až do nuly; index současného prvku při tom do konstanty uložit nelze.

⟨*for*⟩ ::= `"for"` ⟨*name*⟩? ⟨*name*⟩ `":"` ⟨*expression*⟩ ⟨*block*⟩

2.1.4 Tokeny

Nejmenší jednotkou syntaxe jsou tokeny. Token může být jedno z následujících:

- **Jméno** je posloupnost znaků, jež obsahuje pouze číslice, malá a velká písmena latinky a podtržítka a jejíž první znak není číslice. Zyba nemá klíčová slova, ale některá jména (např. `if`, `fun`, `export`) mají zvláštní význam na určitých místech kódu, takže není vhodné je používat pro pojmenování vlastních proměnných.

- **Literál celého čísla** je buď posloupnost dekadických číslic tvořící číslo, nebo posloupnost dekadických číslic, udávající základ číselné soustavy, a po ní znak `r` a posloupnost číslic z oné číselné soustavy, písmena A až z bez ohledu na velikost se považují za číslice 10 až 35. Lze používat i soustavy o základu vyšším než 36, ale není v nich možné zapsat všechna čísla protože pro ně nejsou další číslice.
- **Literál reálného čísla** začíná jako literál celého čísla, ale pokračuje desetinnou tečkou po které můžou následovat další číslice. Rovněž lze používat libovolnou číselnou soustavu.
- **Literál logické hodnoty** je `1b` pro pravdu a `0b` pro nepravdu. Je to totiž kratší než tradiční `true` a `false` a navíc je zápis obou hodnot stejně dlouhý.
- **Literál textu** je posloupnost znaků v uvozovkách. Patří do něj všechny znaky mezi nimi, tedy i konce řádků a speciální znaky, ale je-li potřeba zapsat znak uvozovek, musí se zdvojit, aby nedošlo k ukončení literálu.
- **Operátor** je libovolná poslounost znaků „`+-*/%&|~^<>=!`“. Jestli dotyčný operátor opravdu existuje a lze jej aplikovat na tyto argumenty se posoudí až při vyhodnocování sémantiky.
- **Oddělovač** je jeden znak; některý ze znaků „`()[]{}.:`“.

2.1.5 Příklad

```
# Toto je komentář
# Deklarace konstant
a = 1103515245
# 12345 zapsané v osmičkové soustavě
c = 8r30071
# Tady využíváme literály v soustavě o základu 256
# Číslice 36 až 255 neexistují, ale toto číslo je nepotřebuje
m = 256r10000
# Pokud je poslední příkaz ve funkci vyhodnocení výrazu,
# vrátí se hodnota toho výrazu
next = fun[x: int] int {
  x * a + c % m
}
# Zápis funkce s více argumenty
mocnina = fun[zaklad exp: int] int {
  result = 1
  while exp > 0 {
    result = result * zaklad
    exp = exp - 1
  }
  return result
}
```

2.2 Kontext

Každé jméno v kódu překladač vyhodnoduje v určitém kontextu, který zahrnuje všechna jména, jež na daném místě v kódu označují nějakou proměnnou, konstantu či jmenný prostor. Proměnné i konstanty jsou jména zastupující určitou hodnotu. Jmenný prostor je kolekce obsahující další proměnné, konstanty a jmenné prostory, k nimž se přistupuje skrz onen jmenný prostor.

Proměnná se vytváří uvnitř bloku příkazů přiřazením hodnoty určitému jménu, jenž na daném místě ještě nic neoznačuje, a je přístupné až do konce toho bloku. Argumenty lambda funkcí jsou rovněž proměnné, jejichž hodnoty se nastaví při zavolání funkce na hodnoty předaných argumentů a jsou dostupné uvnitř celé lambda funkce. Proměnnou jde změnit tím, že se do ní přiřadí nová hodnota; ta musí mít stejný typ jako předchozí hodnota proměnné, kterou nová hodnota nahradila.

Konstanta se deklaruje buď přiřazením na úrovni souboru (vně všech bloků), tehdy je dostupná od přiřazení až po konec souboru, nebo vzniká v cyklu `for` a je přístupná uvnitř jeho bloku. Přiřadíme-li na úrovni souboru konstantě lambda funkci, bude konstanta přístupná v celém souboru, nejen po přiřazení. Jednou vytvořené konstantě už není možné přiřadit novou hodnotu a není možné vytvořit ani jmenný prostor se stejným názvem, ale konstanta vytvořená v cyklu `for` má novou hodnotu při každé iteraci.

Jmenný prostor vzniká při importu z jiného souboru a obsahuje všechny exportované konstanty a jmenné prostory deklarované v onom souboru. Od dotyčného importu do konce souboru přes něj lze přistupovat k jeho prvkům pomocí syntaxe `prostor.prvek`. Obsahuje-li jmenný prostor další exportované jmenné prostory, lze přístupy zřetězit a přistupovat i k jejich prvkům: `prostor1.prostor2.prostor3.prvek`. Kruhové závislosti mezi soubory jsou zakázané.

2.3 Typový systém

Zyba je staticky typovaný jazyk se silnou typovou kontrolou; typy všech hodnot tedy musí být známy v čase překladu a nemůžou se implicitně konvertovat. Pro konverzi hodnoty z jednoho typu na druhý lze použít zabudované metody, pokud programátor nechce konverzi implementovat ručně. Určitou zvláštností Zyby je to, že v místě, kde je očekáván typ, se dá použít libovolná hodnota onoho typu (např. hodnota `3+4` se dá použít jako typ `int`). Jména `int`, `bool` a další totiž ve skutečnosti znamenají pouze výchozí hodnoty těchto typů, nikoliv typy samotné (např. `int` je totéž jako `0`). Samozřejmě by se při psaní kódu měly používat názvy typů na místech, kde je očekáván typ, a hodnoty na místech, kde je očekávána hodnota; je sice možné to nedodržovat, ale vzniklý kód bude matoucí. Tato zvláštnost ale má i některé výhody, protože jednak činí jazyk jednodušším a navíc umožňuje vytvářet typové aliasy úplně stejně jako se vytváří proměnné a konstanty. Ostatně tento přístup má určitou paralelu v chování některých objektově orientovaných jazyků (např. C++[1]), které umožňují volat statické metody jak na třídě, tak i na objektu; i v nich se tedy dá někdy použít hodnota (objekt) na místo typu (třídy).

2.3.1 Datové typy

Název	Popis	Výchozí hodnota
`void`	Prázdná hodnota, návratový typ funkcí, které nic nevracejí	Bezvýznamná hodnota
`int`	Celé číslo, 32bitové nebo 64bitové v závislosti na systému	Nula (`0`)
`real`	Reálné číslo	Nula jako reálné číslo (`0.0`)
`bool`	Logická hodnota	Nepravda (`0b`)
`text`	Textový řetězec	Prázdný text (`""`)
`db`	Připojení k databázi	Nezahájené připojení
`T.list`	Pole hodnot typu `T`	Prázdné pole
`T.dict[K]`	Slovník s klíči typu `K` (musí být `int` nebo `text`) a hodnotami typu `T`	Prázdný slovník
`T.fun[`A_0 ... A_m`]`	Funkce přebírající argumenty typů A_0 až A_m a vracející `T`	Funkce přebírající libovolné argumenty typů A_0 až A_m vracející vždy výchozí hodnotu typu `T`
`(`N_0`:`T_0 ... N_m`:`T_m`)`	Záznam s prvky, jejichž názvy jsou N_0 až N_m a typy jsou T_0 až T_m	Záznam se všemi prvky nabývajícími svých výchozích hodnot

Funkce se vytváří pomocí syntaxe lambda funkce, jež specifikuje typy argumentů a návratové hodnoty a obsahuje kód, který se při zavolání funkce vykoná. Pokud se program dostane k příkazu `return`, funkce skončí a vrátí hodnotu předanou tomuto příkazu. Poslední výraz ve funkci, jejíž návratový typ není `void`, musí být `return` nebo vyhodnocení výrazu, které na tomto místě znamená, že funkce výsledek tohoto výrazu vrátí. Funkce se volá pomocí hranatých závorek, předají se jí argumenty odpovídajících typů a výsledkem volání funkce je hodnota, již funkce vrátila. Každý soubor může obsahovat funkci s názvem `main`, všechny tyto funkce se po spuštění zavolají, přičemž je zaručeno, že `main` z určitého souboru se zavolá až po zavolání funkcí z importovaných souborů a jejich závislostí.

Záznam se vytváří stejným způsobem jako se značí typ záznamu, jen se píší hodnoty jeho prvků a ne typy. Zápis `{`N_0 H_0 ... N_m H_m`}` vytvoří záznam, jehož prvky budou mít názvy N_0 až N_m a nabývají hodnot T_0 až T_m. K prvkům záznamu se přistupuje pomocí tečky (např. `zaznam.prvek`), přičemž obsahuje-li záznam další záznamy, lze přístupy řetězit stejně jako u jmenných prostorů.

Pole a slovníky se vytváří zcela totožným způsobem, jakým se značí typ pole či slovníku, právě proto, že typ je skutečnosti sám jen prázdné pole či slovník. Při vytváření lze rovnou přidat prvky tím, že se zabudované metodě `list` či `dict` předají další argumenty, což budou hodnoty oněch prvků. U slovníku se jako argumenty střídají klíče a prvky: `T.dict[K` K_0 V_0 ... K_n V_n`]`. Připojení k databázi se vytváří zabudovanou metodou `connect`, jež přebírá tři argumenty typu `text`: *connection string* používaný PHP knihovnou PDO, uživatelské jméno a heslo.

Pole, slovníky i připojení k databázi jsou tzv. referenční typy, což znamená, že do proměnných se neukládá jejich hodnota, ale pouze reference na místo v paměti, kde je jejich hodnota uložená. Přiřadíme-li je proto do více proměnných, budou všechny obsahovat reference na stejné místo, takže změníme-li pomocí zabudovaných metod hodnotu na jednom místě, projeví se tato změna ve všech proměnných, které obsahují referenci na toto místo.

2.3.2 Zabudované metody a operátory

Datové typy mají svoje zabudované metody a operátory. Obojí jsou zvláštní výrazy, jež něco vykonají a vrátí nějakou hodnotu, ale liší se syntaxí. Syntaxe volání zabudovaných metod i operátorů již byla popsána výše. Operátory logické konjunkce a disjunkce svoje argumenty vyhodnocují líně, tedy plyne-li ze znalosti prvního argumentu znalost výsledku, druhý argument se nevyhodnoduje. Operátory se dělí na aritmetické, bitové, logické, porovnávací a složené. Aritemetické operátory přebírají dva argumenty typu `int` nebo `real`, přičemž oba argumenty nemusí mít stejný typ. Patří mezi ně:

Operátor	Výsledek	Typ
`+`	Součet	`int`, jsou-li argumenty `int`, jinak `real`
`-`	Rozdíl	
`*`	Součin	
`**`	Mocnina	
`%`	Zbytek po dělení (pro záporné dělence je záporný)	
`/`	Podíl	`real`
`//`	Podíl (zaokrouhlený k nule)	`int`

Bitové operátory přijímají dva argumenty typu `int` a vrací rovněž `int`.

Operátor	Výsledek	Typ
`&`	Bitová konjunkce	`int`
`|`	Bitová disjunkce	`int`
`^`	Bitová nonekvivalence	`int`

Logické operátory přebírají dva argumenty typu `bool` a vrací `bool`, přičemž logická konjunkce a disjunkce svoje argumenty vyhodnocují líně, což znamená, že plyne-li ze znalosti prvního argumentu znalost výsledku, druhý argument se nevyhodnoduje.

Operátor	Výsledek	Typ
`&`	Logická konjunkce	`bool`
`|`	Logická disjunkce	`bool`
`^`	Logická nonekvivalence	`bool`

Porovnávací operátory zjišťují buď rovnost nebo uspořádání dvou hodnot. Ty pro zjišťování rovnosti přebírají dva argumenty stejného typu, který ovšem není funkce, `void`, `real`, `db`, ani pole, slovník či je obsahující. Zjišťování rovnosti u reálných čísel není povo-

lené, protože kvůli jejich reprezentaci v počítači tato operace může dávat špatné výsledky. Pole, slovníky a záznamy se považují za rovné, pokud jsou si rovné všechny jejich prvky. Operátory zjišťující uspořádání přebírají buď dvě hodnoty typu `text`, nebo dvě čísla, z nichž každé je typu `int` či `real`.

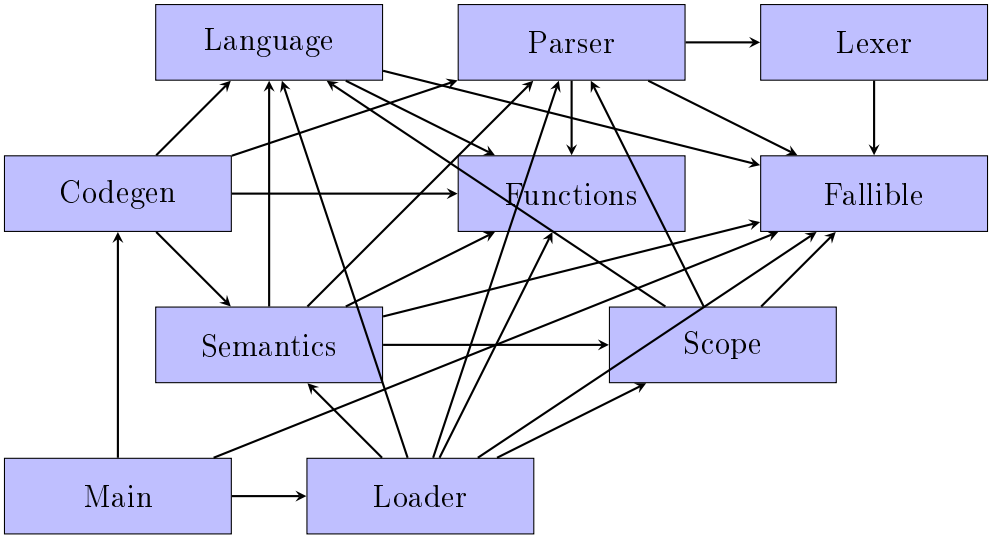
Operátor	Výsledek	Typ
`==`	Rovnost	`bool`
`!=`	Nerovnost	`bool`
`<`	Menší než	`bool`
`>`	Větší než	`bool`
`<=`	Menší nebo rovno	`bool`
`>=`	Větší nebo rovno	`bool`

Složené operátory pracují s poli, slovníky, záznamy a typem `text`. Operátory pracující s poli a slovníky přebírají dva argumenty stejného typu, nazvěme jej `T.list`, jde-li o pole, a `T.dict[K]`, jde-li o slovník. Dojde-li při průnicích nebo sjednoceních slovníků a záznamů ke kolizi dvou prvků se stejným klíčem nebo názvem, započítá se do výsledku prvek z pravého argumentu.

Operátor	Výsledek	Typ
`+`	Spojení textů	`text`
`+`	Spojení polí	`T.list`
`|`	Sjednocení slovníků	`T.dict[K]`
`&`	Průnik slovníků	`T.dict[K]`
`|`	Sjednocení záznamů	Typ záznamu, který obsahuje všechny prvky, které byly alespoň v jednom ze záznamů
`&`	Průnik záznamů	Typ záznamu, který obsahuje všechny prvky, které byly v obou záznamech

3 Konstrukce překladače

Překladač je napsaný v jazyce Haskell a jeho kód je rozdělený do několika modulů.



Obrázek 1: Graf závislostí mezi moduly

3.1 Přehled modulů

Modul `Functions` zahrnuje pouze pomocné funkce, které nemají nic společného s logikou tohoto konkrétního programu a pracují s textem, seznamy a monádami. Slouží ke zkrácení kódu v ostatních modulech tím, že abstrahují chování, jež se v nich opakuje.

Modul `Fallible` obsahuje stejnojmennou monádu, její transformátor a funkce pro práci s nimi. Monáda `Fallible` je alias pro `Either String` a obsahuje buď hodnotu nějakého výsledku, nebo text s chybovou hláškou. Její spojování je definováno tak, že spojení chybové hodnoty s další funkcí bude tuto funkci ignorovat a vrátí původní chybovou hodnotu. Pokud v některé funkci při překladu programu dojde k chybě, což může nastat jen tehdy, když je program špatně napsaný, vrátí příslušná funkce hodnotu `Fallible` reprezentující chybu a vzhledem k vlastnostem spojení se další části překladu neprovedou a výsledkem programu bude chyba. Dále je zde definována monáda `FallibleIO`, což je alias pro výsledek monádového transformátoru, který zabalí monádu `IO` do `Fallible`.

Modul `Lexer` implementuje lexikální analyzátor pro Zybu. Jeho funkce `tokenize` přebírá `String` a vrací `Fallible [(Integer, Token)]`, z čehož každá dvojice reprezentuje token a číslo řádku, na němž se token nacházel. Čísla řádků se používají při generování chybových hlášek. `Token` je algebraický datový typ, jehož konstruktory odpovídají jednotlivým typům tokenů zmiňovaných v sekci Syntaxe.

Modul `Parser` provádí syntaktickou analýzu programu. Obsahuje funkci `parse`, jež dostane na vstupu `String`, rozdělí jej na tokeny pomocí funkce `tokenize`, sestaví syntaktický strom a vrátí `Fallible File`. Typ `File`obsahuje seznam trojic `(Integer, Visibility, Declaration)`, kde první prvek je opět číslo řádku, druhý, `Visibility`, udává, zda je deklarace exportovaná, a třetí je samotná deklarace. Dále modul obsahuje algebraické datové typy modelující syntaktický strom `Declaration` pro deklarace, `Statement` pro příkazy, `Value` pro výrazy a `Literal` pro literály. Mají konstruktory pro každý typ syntaktické konstrukce zmíněné v sekci syntaxe.

Modul `Language` definuje typy (algebraický typ `Type`) a zabudované metody a operátory (algebraický typ `Builtin`) jazyka Zyba. Kromě toho jsou v něm obsaženy funkce pro jejich vyhledávání a typovou kontrolu při operacích s nimi.

Modul `Scope` definuje stejnojmenný typ, který reprezentuje kontext pro proměnné, konstanty a jmenné prostory. Modul ovšem neexportuje konstruktor typu `Scope`, takže z jiných modulů lze kontext číst a měnit jen pomocí dalších funkcí z tohoto modulu.

Modul `Semantics` se zabývá sémantickou analýzou. Jsou tu definovány typy `Value` a `Statement`, jejichž hodnoty se vytváří ze stejnojmenných typů z modulu `Parser` po sémantické analýze. Hlavní funkce `analyse` přijímá hodnotu typu `File` se syntaktickým stromem analyzovaného souboru, mapu `Map String Scope`, obsahující kontexty s deklaracemi exportovanými z již analyzovaných souborů překládaného programu, a `String` obsahující cestu k právě analyzovanému souboru; návratová hodnota je typu `Fallible (Scope, [Declaration])`. První prvek dvojice je kontext s exportovanými deklaracemi, druhým prvkem je seznam deklarací; v seznamu však jsou jen hodnoty a nikoliv importy, protože seznam se používá jen při generování kódu pro deklarované hodnoty. `Declatation` se definuje jako `(String, (Value, Type))`, což jsou název, hodnota a typ deklarované hodnoty.

Modul `Loader` zajišťuje správu importů a spojení syntaktické a sémantické analýzy. Obsahuje funkci `load`, jež dostane na vstupu cestu k hlavnímu souboru a vrátí jako výsledek `FallibleIO ([String], [(String, [Declaration])])`. První prvek vnější dvojice je seznam obsahů všech importovaných PHP souborů. Druhý prvek je seznam dvojic, z nichž každá obsahuje název a seznam deklarací hodnot jednoho souboru Zyby. Zyba soubory jsou seřazeny v takovém pořadí, že pokud modul A závisí na modulu B, bude v seznamu modul B před modulem A. Z toho taky plyne, že není možné mít kruhové závislosti a v případě výskytu takové závislosti vrátí funkce `load` hodnotu chyby.

Modul `Codegen` završuje kompilaci, neboť jeho funkce `gen` vygeneruje PHP kód na základě výstupu z funkce `load`. V této funkci už nemůže dojít k chybě, protože program prošel všemi fázemi kontroly, takže návratová hodnota funkce je pouze `String`.

Modul `Main` poté ve funkci `main` přečte dva argumenty z příkazové řádky – cestu ke vstupnímu Zyba souboru a výstupnímu PHP souboru. Zavolá funkci `load` s cestou ke vstupnímu souboru, její výsledek předá funkci `gen` a vzniklý PHP kód zapíše do zadaného souboru. Dojde-li k chybě, nic se do zadaného souboru nezapíše a chybová hláška bude vypsána na standardní chybový výstup.

3.2 Ukázka překladu

Zde je ukázka jednoduchého programu v Zybě a výsledného kódu v PHP. Jedná se o webovou stránku pro výpočet faktoriálu. Pokud bude v URL argument `num`, převede jeho hodnotu na číslo a zobrazí stránku s faktoriálem z tohoto čísla; jinak zobrazí formulář, do něhož uživatel může zadat číslo a po stisku tlačítka bude přesměrován na stejnou stránku se zadaným číslem předeným v URL jako argument `num`, takže se zobrazí faktoriál zadaného čísla.

Program používá pomocnou knihovnu `web` napsanou v Zybě a PHP, jež zajišťuje přístup

k potřebným funkcím a proměnným z PHP. Celý program tak tvoří tři soubory: `web.php`, `web.zyba` a `example.zyba`.

Soubor `web.php`:

```
<?php
$__GET = z1array::n($_GET);
$__POST = z1array::n($_POST);
$__COOKIE = z1array::n($_COOKIE);
$__SESSION = z1array::n($_SESSION);
$__FILES = [];
foreach ($_FILES as $fname => $f) {
  $__FILES[$fname] = z1array::n([
    'name' => $f['name'],
    'type' => $f['type'],
    'size' => $f['size'],
    'ok' => $f['code'] === 0,
    'tmp_name' => $f['tmp_name']
  ]);
}
$__FILES = z1array::n($__FILES);
$move_uploaded_file = function($from, $to) {
  move_uploaded_file($from, $to);
};
$echo = function($s) {
  echo($s);
};
?>
```

Soubor `web.zyba`:

```
export File = (
  name: text
  type: text
  size: int
  ok: bool
  tmp_name: text
)

import php old "web.php" (
  __GET: text.dict[text]
  __POST: text.dict[text]
  __COOKIE: text.dict[text]
  __SESSION: text.dict[text]
  __FILES: File.dict[text]
  move_uploaded_file: void.fun[text text]
  echo: void.fun[text]
)
```

```
export args = old.__GET
export form = old.__POST
export cookies = old.__COOKIE
export session = old.__SESSION
export files = old.__FILES

export save_uploaded = fun[key where : text] void {
  old.move_uploaded_file[files.get[key].tmp_name where]
}

export print = fun[value : text] void {
  old.echo[value]
}
```

Soubor `example.zyba`:

```
import web "web.zyba"

factorial = fun[n : int] int {
  result = 1
  for i : n {
    result = result * i
  }
  result
}

main = fun[] void {
  web.print["<!DOCTYPE html>
  <html lang=""cs-cz"">
    <head>
      <title>Faktoriál</title>
      <meta charset=""utf-8"" />
    </head>
    <body>"]
  if web.args.has["num"] {
    num = web.args.get["num"].asInt
    fact = factorial[num]
    web.print["<p>" + num.asText + "! = " + fact.asText + "</p>"]
  } else {
    web.print["<form>
      Číslo: <input type=""number"" name=""num""/>
      <input type=""submit"" value=""Spočítat""/>
    </form>"]
  }
  web.print["</body></html>"]
}
```

4 Závěr

Navrhl jsem programovací jazyk Zyba a zkonstruován jeho překladač. Jazyk lze použít pro psaní webových stránek, přičemž funkce pro interakci s okolím jsou buď obsaženy jako zabudované metody (práce s databází) nebo je lze importovat z PHP. Kód v Zybě napsaný je dobře čitelný, překladač provádí statickou typovou kontrolu a systém importů a exportů je poměrně silný, třebaže by bylo možné implementovat ještě kruhové importy. Hlavní slabinou jazyka je chybějící podpora pro generické programování, což znemožňuje tvorbu funkcí, které by pracovaly s libovolnými daty (řazení, přístup k databázi) a je tak nutné pro ně buď vytvořit v jazyce zabudované metody, nebo mít jednoúčelové funkce. To jsou tedy možné směry pro budoucí vývoj jazyka.

Použitá literatura

[1] *Working Draft, Standard for Programming Language C++*. Tech. zpr. Břez. 2017. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf.