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 5. 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 **5**

2 Návrh jazyka **5**

2.1 Syntaxe 5

2.1.1 Deklarace 5

2.1.2 Výrazy 6

2.1.3 Funkce a příkazy 6

2.1.4 Tokeny 7

2.1.5 Příklad 8

2.2 Kontext 9

2.3 Typový systém 9

2.3.1 Datové typy 10

2.3.2 Zabudované metody a operátory 10

3 Konstrukce překladače **11**

4 Závěr **12**

Použitá literatura **13**

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. K vyhodnocení typů by mohlo dojít 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 složené závorky a v nich několik dvojic názvů (tj. jména importovaných hodnot) a výrazů (tj. jejich typy).

⟨*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` má dvě formy. První z nich se používá pro daný počet opakování, jež je specifikován nějakým výrazem, v tom případě po slově `for` následuje jméno konstanty, která udává, kolik opakování už proběhlo, operátor `<`, onen výraz a blok příkazů. Druhou lze použít pro iteraci přes pole (včetně asociativního), ta se zapisuje slovem `for`, po němž následují jména dvou konstant, z nichž první bude obsahovat index současného prvku a druhá jeho hodnotu, oddělovač `:`, výraz, jehož hodnotou je ono pole, a blok příkazů. První jméno je nepovinné, v tom případě se hodnota indexu nebude do žádné konstanty ukládat.

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

⟨*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 již zmiňované 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 (`""`)
`T.list`	Pole hodnot typu `T`	Prázdné pole
`K.dict[V]`	Asociativní pole s klíči typu `K` a hodnotami typu `V`	Prázdné asociativní pole
`R.fun[`$\texttt{T}_0$ `...` $\texttt{T}_m$`]`	Funkce přebírající argumenty typů $\texttt{T}_0$ až $\texttt{T}_m$ a vracející `R`	Funkce přebírající libovolné argumenty typů $\texttt{T}_0$ až $\texttt{T}_m$ vracející vždy výchozí hodnotu typu `R`
`{`$\texttt{N}_0$ $\texttt{T}_0$ `...` $\texttt{N}_m$ $\texttt{T}_m$`}`	Záznam s prvky, jejichž názvy jsou $\texttt{N}_0$ až $\texttt{N}_m$ a typy jsou $\texttt{T}_0$ až $\texttt{T}_m$	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. V každém souboru může být funkce s názvem `main`, všechny tyto funkce se po spuštění zavolají, přičemž

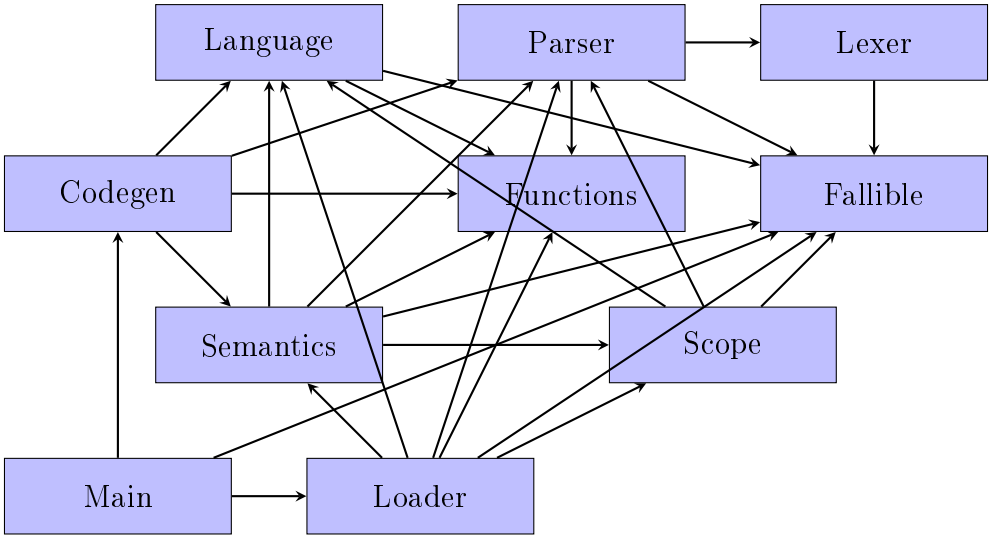
Záznam se vytváří stejným způsobem jako se značí typ záznamu, jen se píší hodnoty jeho prvků a ne typy. Zápis `{`$\texttt{N}_0$ $\texttt{H}_0$ `...` $\texttt{N}_m$ $\texttt{H}_m$`}` vytvoří záznam, jehož prvky budou mít názvy $\texttt{N}_0$ až $\texttt{N}_m$ a nabývají hodnot $\texttt{T}_0$ až $\texttt{T}_m$. K prvkům záznamu se přistupuje pomocí tečky (např. `zaznam.prvek`), přičemž obsahuje-li záznam další záznamy, lze přístupy řetězit stejně jako u jmenných prostorů.

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 popsáná výše.

3 Konstrukce překladače

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



Obrázek 1: Graf závislostí mezi moduly

Modul `Functions` obsahuje jen pomocné funkce, které nemají nic společného s logikou tohoto konkrétního programu, jedná se o různé funkce pro práci s textem, seznamy a monádami. Modul `Fallible` definuje stejnojmennou monádu, její transformátor a funkce pro práci s nimi, což slouží k práci s chybami při překladu.

Lexikální analyzátor pro Zybu implementuje modul `Lexer`, jehož funkce `tokenize` dostane na vstupu text a vrátí seznam dvojic `(Integer, Token)`, jež reprezentují tokeny a čísla řádků, na nichž se tokeny nacházely (kvůli generování chybových hlášek). Typ `Token` je algebraický datový typ, jehož konstruktory odpovídají typům tokenů zmiňovaných v sekci Syntaxe.

Syntaktická analýza se odehrává v modulu `Parser`, jehož funkce `parse` přijímá text, funkcí `tokenize` jej rozdělí na tokeny a pak sestaví syntaktický strom. Výsledkem je hodnota typu `File`, který obsahuje seznam trojic `(Integer, Visibility, Declaration)`, kde první prvek je opět číslo řádku, druhý je viditelnost deklarace (tj. zda je deklarace exportovaná) a třetí je samotná deklarace. Dále modul obsahuje algebraické datové typy `Declaration`, pro reprezentaci deklarací, `Visibility`, která udává, zda je deklarace exportovaná, `Statement`, který reprezentuje příkaz (a má konstruktor pro každý typ příkazu), `Value` reprezentující výraz a `Literal`, který obsahuje hodnotu literálů.

V modulu `Language` jsou definované zabudované metody a operátory (algebraický typ `Builtin`), typy (algebraický typ `Type`) a funkce pro práci s nimi. Další modul, `Scope`, obsahuje stejnojmenný typ, který reprezentuje kontext pro proměnné, konstanty a jmenné prostory, funkce pro jejich přidávání do kontextu a pro zjišťování obsahu kontextu. Typ `Scope` totiž neexportuje konstruktor, takže z jiných modulů lze k jeho obsahu přistupovat jen pomocí těchto funkcí.

Tyto dva moduly se využívají při sémantické analýze v modulu `Semantics`. Jsou tu definovány typy `Value`, `Statement` a `Declaration`, jejichž hodnoty se vytváří ze stejnojmenných typů z modulu `Parser` po sémantické analýze. Hlavní funkce `analyse` přijímá jednak hodnotu typu `File`, která obsahuje syntaktický strom analyzovaného souboru, ale taky mapu `Map String Scope`, jež obsahuje kontexty s exportovanými deklaracemi z již analyzovaných souborů překládaného programu, a `String` obsahující cestu k právě analyzovanému souboru. Výsledkem analýzy je dvojice `(Scope, [(String, TypedValue)])`, jejíž první prvek je kontext, který se případně předá této funkci při analýze dalších souborů, druhým prvkem je seznam deklarací v analyzovaném souboru.

Správu importů a spojení syntaktické a sémantické analýzy zajišťuje modul `Loader`. Obsahuje funkci `load`, jež dostane na vstupu cestu k hlavnímu souboru a vrátí jako výsledek dvojici. První prvek dvojice je seznam obsahů všech importovaných PHP souborů. Druhý prvek je seznam dvojic, z nichž každá reprezentuje jeden soubor Zyby. Obsahuje jeho název a seznam deklarací (typ `Semantics.Declaration`) z onoho souboru. Zyba soubory jsou seřazeny v takovém pořadí, že pokud modul A závisí na modulu B, bude v seznamu modul B před modulem A.

Kompilaci završí modul `Codegen`, jehož funkce `gen` vygeneruje PHP kód na základě výstupu z funkce `load`. Modul `Main` poté ve funkci `main` očekává dva parametry v příkazové řádce – cestu ke vstupnímu Zyba souboru a výstupnímu PHP souboru. Zavolá funkci `load` s cestou ke vstupnímu souboru, výsledek předá funkci `gen` a vzniklý PHP kód zapíše do zadaného souboru.

4 Závěr

Byl úspěšné navržen 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.