FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Návrh šablonovacího jazyka a implementace překladače
Student:	Adam Plodek
Vedoucí:	Ing. Štěpán Plachý
Studijní program:	Informatika
Obor / specializace:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Proveďte rešerši existujících šablonovacích jazyků.
Po dohodě s vedoucím navrhněte šablonovací jazyk se zaměřením na generování HTML pro zobrazování strukturovaných dat, kolekcí a objektů. V návrhu jazyka se zaměřte na funkcionální principy.
Váš jazyk popište formální gramatikou a implementujte pro něj překladač ve vhodném programovacím jazyce.
Vaši implementaci otestujte.

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 25. února 2022 v Praze.

Bakalářská práce

NÁVRH ŠABLONOVACÍHO JAZYKA A IMPLEMENTACE PŘEKLADAČE

Adam Plodek

Fakulta informačních technologií
Katedra teoretické informatiky
Vedoucí: Ing. Štěpán Plachý
9. května 2022

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat především mému vedoucímu Ing. Štěpánu Plachému a to nejen za časté konzultace a odborné rady, ale i za vlídný přístup a pochopení během obou semestrů, ve kterých jsem na bakalářské práci pracoval. Dále bych chtěl poděkovat mojí rodině za pochopení, když jsem musel upřednostnit studium, a mému spolubydlícímu Tomáši Věžníkovi za občasné sdílení názorů na návrh jazyka.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 9. května 2022 .....

Abstrakt

Cílem této práce je návrh a implementace šablonavacího jazyka, pro tvorbu HTML stránek, který se zaměřuje na podporu funkcionálního stylu programování, pomocí nehož lze vylepšit vlastnosti šablonovacích jazyků. Základními vlastnostmi navrženého jazyka jsou podpora funkcí vyššího řádu, anonymní funkce a silný typovací systém. Pro šablonovací jazyk byla vytvořena formální gramatika a implementace překladače byla provedena v programovacím jazyce Haskell.

Klíčová slova šablonovací jazyk, implementace překladače, formální jazyky, formální gramatika, funkcionální programování, LL(1) analýza, Haskell

Abstract

Goals of my thesis are design and implementation of templating language, for creating HTML templates, which focuses on support for functional programing, which could improve properties of templating languages. Essential features of designed language are support for higher order functions, anonymous functions and strong typing system. For templating language was created formal grammar and implementation of compiler was done in Haskell programing language.

Keywords templating language, compiler implementation, formal languages, formal grammar, functional programing, LL(1) analýza, Haskell

Seznam zkratek

AST	Abstraktní syntaktický strom
HTML	HyperText Markdown Language
LIFO	Last In First Out

Kapitola 1
Úvod

Šablonovací jazyky pro HTML se používají pro programovou tvorbu HTML stránek. Díky těmto jazykům je možno vytvářet serverově rendrované weby a zjednodušit vytváření větších HTML stránek, které by byli obtížné vytvořit pouze pomocí čistého HTML. Funkcionální programovací jazyky se vyznačují především imutable datovými strukturami a first class funkcemi (k funkci se lze chovat jako k hodnotě). Spojení těchto dvou světů není velmi běžné, standardní šablonovací jazyky využívají syntaxi a sémantiku podobnou skriptovacím jazykům, jako například Javascript, Python, nebo Ruby.

Díky vlastnostem funkcionálních programování lze psát programy, které se vyznačují svojí jednoduchou rozšiřitelností a předvídatelností. Z toho samozřejmě plyne i jednodušší zajištění bezpečnosti napsaného kódu.

Na začátku práce se v její teoretické části zaměřím na algoritmy a koncepty potřebné k implementaci LL(k) parseru a vygenerování výsledku.

Následně se přesunu na vytvoření rešerše již existujících řešeních a návrh nového šablonovacího jazyka, kde bych se chtěl zaměřit na možnosti vylepšení pomocí funkcionálních praktik.

Součástí praktické části této práce pak bude implementace navrženého jazyka a to včetně implementace standardní knihovny pro tvorbu HTML. Ke konci práce se budu zabývat testováním kompilátoru.

1.1 Cíle

Cílem této bakalářské práce je návrh syntaxe a sémantiky jazyka pro HTML, který je zaměřen na funkcionální styl a praktiky, a implementovat pro tento jazyk překlakladač a vytvořit standardní knihovnu určenou pro tvorbu HTML stránek. Díky tomu by pomocí tohoto jazyky bylo možno dosáhnou spolehlivějšího kódu v šabloně.

Jazyk by měl dovolovat tvorbu strukturovaných dat a to pomocí, jak tvorbou recordů a homogeních polí, tak i algebraických typů. Dále by mělo být možno dosáhnou standardních praktik ve funkcionálních jazycích, především funkce vyššího řádu (higher order functions).

Kapitola 2
Teoretická část

V teoretické části se zaměřím na definice pojmů, které jsou nutné k tvorbě gramatik, LL(1) parserů, generaci výstupu z programu a ostatní pojmy, které jsem využil při tvorbě navrhovaného jazyka. K citacím těchto definic jsem využil Automaty a gramatiky: sbírka řešených příkladů od Elišky Šestákové [1], Konstrukce překladačů od Bořivoje Melichara [2] a Programovací jazyky od Karla Müllera [3].

2.1 Základní pojmy

V této části se budu věnovat základním definicím gramatik, jejich vlastnostmi a základům teorie grafů, které dále využiji v části o AST.

▶ **Definice 2.1** (Orientovaný graf [2]). *Orientovaný graf je uspořádaná dvojice $G = (V, E)$, kde V je konečná množina vrcholů a $E \subseteq V \times V$ je množina hran.*

- *o hraně $(x, y) \in E$ říkáme, že vychází z vrcholu x a vchází do vrcholu y*
- *konečná posloupnost hran $(x_0, x_1), (x_1, x_2), \cdots, (x_{n-1}, x_n)$ je nazývá cesta o délce n*

▶ **Definice 2.2** (Zakořeněný strom [2]). *Graf $G = (V, E)$ se nazývá stromem, pokud:*

- *obsahuje vrchol $v \in V$, takový že do něj nevchází hrana $(\forall u \in V : (u, v) \notin V)$*
- *pro každý vrchol, který není kořenem existuje právě jedna cesta z kořene*
- *vrcholy, ze kterých nevychází hrana se nazývají listy stromu*

▶ **Definice 2.3** (Formální jazyk [1]). *Formální jazyk je množina L nad Σ, taková že $L \subseteq \Sigma^*$, kde Σ je konečná množina symbolů. Kde Σ je konečné množina symbolů, které se nazývá abeceda.*

▶ **Definice 2.4** (Řetězec [1]). *Řetězec je každá konečná posloupnost symbolů abecedy (konečné množiny)*

- *ε je prázdný řetězec*
- *Σ^* je množina všech řetězců nad abecedou Σ*

▶ **Definice 2.5** (Gramatika [1]). *Gramatika je čtveřice $G = (N, \Sigma, P, S)$, kde:*

- *N je konečná množina neterminálních symbolů*
- *Σ je konečná množina terminálních symbolů ($\Sigma \cap N = \emptyset$)*

- *P je množina (přepisovacích) pravidel. Je to konečná podmnožina množiny $(N \cup \Sigma)^*.N.(N \cup \Sigma)^* \times (N \cup \Sigma)^*$ (zapisuje se $(\alpha, \beta) \in P = \alpha \to \beta$)*
- *$S \in N$ je počáteční symbol gramatiky*

▶ **Definice 2.6** (Derivace [3]). *Mějme gramatiku $G = (N, T, P, S)$ a $x, y \in (N \cup T)^*$. Pak říkáme že x přímo derivuje y ($x \Rightarrow y$), když:*

- *$x = \alpha A \beta, y = \alpha \chi \beta$*
- *$A \in N$, $\alpha, \beta, \chi \in (N \cup T)^*$*
- *Existuje pravidlo $A \to \chi \in P$*

A derivace z řetězce β z řetězce α délky k se nazývá posloupnost řetězců $\alpha_0, \alpha_1, \cdots, \alpha_k$, takových že $\alpha = \alpha_0$, $\forall i \in N : 1 \leq i \leq k, \alpha_{i-1} \Rightarrow \alpha_i$ a $\alpha_k = \beta$. Derivace derivace libovolné délky se značí $\Rightarrow^*$

▶ **Definice 2.7** (Řetězce generovaný gramatikou G=(N, T, P, S) [3]). *Řetězec generovaný gramatikou je takový řetězec α, který neobsahuje neterminální symboly a existuje $S \Rightarrow^* \alpha$*

▶ **Definice 2.8** (Bezkontextová gramatika [1]). *Gramatika $G = (N, \Sigma, P, S)$ je bezkontextové, pokud každé pravidlo je ve tvaru $A \to \alpha$ je pravidlo z P, kde $A \in N, \alpha \in (N \cup \Sigma)^*$*

▶ **Definice 2.9** (Derivační strom [3]). *Derivační strom je zakořeněný strom ($S = (V, E)$), který vyjadřuje derivaci řetězce v gramatice $G = (N, T, P, S)$, pro který platí:*

- *uzly derivačního stromu jsou terminální a neterminální symboly ($V = T \cup N$)*
- *kořen je počáteční symbol gramatiky*
- *pokud z vrcholu vychází alespoň jedna hrana, pak vrchol musí být neterminální symbol*
- *jestliže z vrcholu $v \in V$ vychází hrany $e_1, \cdots, e_n$, která vchází do vrcholů $u_1, \cdots, u_n$, seřazené v tomto pořadí, pak platí $v \to u_1 \cdots u_n \in P$*

▶ **Definice 2.10** (Deterministický konečný automat [1]). *Deterministický konečný automat je pětice $M = (Q, \Sigma, \delta, q_0, F)$, kde:*

- *Q je konečná množina stavů*
- *Σ je konečná vstupní abeceda*
- *δ je funkce z $Q \times \Sigma$ do Q*
- *$q_0 \in Q$ je počáteční stav*
- *$F \subseteq Q$ je množina koncových stavů*

▶ **Definice 2.11** (Nedeterministický konečný automat [1]). *Nedeterministický konečný automat se v definici od nedeterministického konečného automatu liší jen přechodovou funkcí a to následovně : δ je funkce z $Q \times \Sigma$ do $\mathcal{P}(Q)$*

Pomocí deterministického konečného automatu se vyjadřuje část kompilátoru, která se stará o lexikální analýzu zdrojového kódu. [3]

▶ **Definice 2.12** (Přijímání řetězce deterministickým konečným automatem [1]). *Mějme deterministický konečný automat $DKA = (Q, \Sigma, \delta, q_0, F)$ pak:*

- *Přijme vstupní řetězec, pokud jej celý přečte a skončí v některém z koncových stavů*

- *Nepřijme vstupní řetězec, pokud je řetězec celý přečten a automat skončí v nekoncovém stavu, nebo během čtení řetězce nebude definovaná přechodová funkce.*

▶ **Definice 2.13** (Přijímání řetězce nedeterministickým konečným automatem [1]). *Mějme deterministický konečný automat* $DKA = (Q, \Sigma, \delta, q_0, F)$ *pak:*

- *Přijme vstupní řetězec, pokud existuje alespoň jedna posloupnost přechodů, kdy je přečten celý řetězec a automat se nachází v konečném stavu.*
- *Nepřijme vstupní řetězec, pokud neexistuje alespoň jedna posloupnost přechodů, kdy je přečten celý řetězec a automat se nachází v konečném stavu.*

▶ **Definice 2.14** (Jazyk přijímaný konečným automatem [1]). *Jazyk přijímaný konečným automatem je množina všech řetězců, které tento automat přijímá.*

Pro každý nedeterministický konečný automat existuje deterministický konečný automat, který přijímá stejný jazyk a naopak. [1]

▶ **Definice 2.15** (Regulární výrazy [1]). *Mějme abecedu* Σ *pak:*

- $\emptyset, \varepsilon, a$ *jsou regulární výrazy pro všechna* $a \in \Sigma$
- *Jsou-li* x, y *regulární výrazy nad* Σ, *pak* $(x + y)$, $(x \cdot y)$ *a* $(x)^*$ *jsou regulární výrazy*

▶ **Definice 2.16** (Hodnota regulárního výrazu [1]). *Mějme regulární výraze RV, pak jeho hodnota h(RV) je regulární jazyk, který je definovaný následovně:*

- $h(\emptyset) = \emptyset$
- $h(\varepsilon) = \{\varepsilon\}$
- $h(a) = \{a\}$, *kde* $a \in \Sigma$
- $h(x + y) = h(x) \cup h(y)$, *kde* x, y *jsou regulární výrazy*
- $h(x \cdot y) = h(x) \cdot h(y)$, *kde* x, y *jsou regulární výrazy*
- $h(x^*) = (h(x))^*$, *kde* x *je regulární výraz*

Pro zkrácení zápisu využívám v dalším textu prioritu operátorů a to v následujícím pořadí $^*, \cdot, +$ pokud potřebuji změnit prioritu, tak využiji závorek, následně lze zapsat $x \cdot y$ jakožto xy, kde x, y jsou regulární výrazy. Dále využívám při zápisu většího počtu symbolů z jazyka Σ, které lze vyjádřit jakožto interval, zápisu $[a_1 - a_n] = (a_1 + a_2 + \cdots + a_n)$, kde $a_1, \cdots, a_n \in \Sigma$, například $[0 - 9]$.

▶ **Definice 2.17** ((Nedeterministický) Zásobníkový automat [1]). *je sedmice* $R = (Q, \Sigma, G, \delta, q_0, Z_0, F)$, *kde*

- Q *je konečná množina vnitřních stavů*
- Σ *je konečná vstupní abeceda*
- G *je konečná neprázdná abeceda zásobníku*
- δ *je zobrazení z konečné podmnožiny* $Q \times (\Sigma \cup \{\varepsilon\}) \times G^*$ *do množiny konečných podmnožin* $Q \times G^*$
- $q_0 \in Q$ *je počáteční stav*
- $Z_0 \in G$ *je počáteční symbol zásobníku*

- $F \subseteq Q$ *je množina koncových stavů*

▶ **Definice 2.18** (Deterministický zásobníkový automat [1]). *Zásobníkový automat* $R = (Q, \Sigma, G, \delta, q_0, Z_0, F)$ *je deterministický, pokud* $\forall q \in Q \wedge \forall a \in \Sigma \cup \{\varepsilon\} \wedge \forall y, z \in G^*$ *platí:*

- $|\delta(q, a, z)| \leq 1$
- *pokud* $\delta(q, a, z) \neq \emptyset \wedge \delta(q, a, y) \neq \emptyset$, *pak* z *není předponou* y *a opačně*
- *pokud* $\delta(q, a, z) \neq \emptyset \wedge \delta(q, \varepsilon, y) \neq \emptyset$, *pak* z *není předponou* y *a opačně*

Zásobníkové automaty lze vnímat jako rozšíření konečného automatu o zásobník, což je datová LIFO struktura s operacemi push a pop. Díky tomuto rozšíření lze pomocí zásobníkového automatu přijímat i bezkontextové jazyky (tzn. nejen regulární). [1]

Zásobníkové automaty se rozdělují na dva typy podle způsobu přijímání a to přijímající přechodem do koncového stavu a přijímající s prázdným zásobníkem. Tyto dva způsoby jsou ekvivalentní výpočetní silou. [1]

▶ **Definice 2.19** (Přijetí řetězce deterministickým zásobníkový automatem [1]). *Deterministický zásobníkový automat DZA přijme řetězec, pokud jej celý přečte a skončí:*

- *v některém z koncových stavů (přijímá-li přechodem do koncového stavu)*
- *v libovolném stavu a má prázdný zásobník (přijímá-li s prázdným zásobníkem)*

DZA nepřijme řetězec, pokud se během čtení dostane do situace, kdy je výsledkem přechodové funkce prázdná množina, nebo pokud celý řetězec přečte a skončí:

- *v některém z nekoncových stavů (přijímá-li přechodem do koncového stavu)*
- *v libovolném stavu a má neprázdný zásobník (přijímá-li s prázdným zásobníkem)*

▶ **Definice 2.20** (Přijetí řetězce nedeterministickým zásobníkový automatem [1]). *Nedeterministický zásobníkový automat NZA přijme řetězec, pokud existuje alespoň jedna posloupnost přechodů, kdy je přečten celý řetězec a výpočet skončí:*

- *v některém z koncových stavů (přijímá-li přechodem do koncového stavu)*
- *v libovolném stavu a má prázdný zásobník (přijímá-li s prázdným zásobníkem)*

NZA nepřijme řetězec, pokud neexistuje posloupnost přechodů vedoucí k přijetí daného řetězce

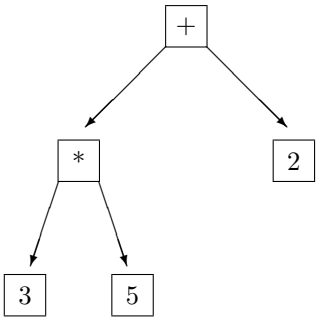
Na rozdíl od konečných automatů, mají deterministické a nedeterministické varianty zásobníkového automatu rozdílné výpočetní síly. [1]

▶ **Definice 2.21** (Zásobníkový překladový automat [2]). *je osmice* $ZPA = (Q, \Sigma, G, D, \delta, q_0, Z_0, F)$, *kde:*

- Q *je konečná množina vnitřních stavů*
- Σ *je konečná množina vstupních symbolů*
- G *je konečná množina symbolů (zásobníkové symboly)*
- D *je konečná množina výstupních symbolů*
- δ *je konečné zobrazení z* $Q \times (\Sigma \cup \{\varepsilon\}) \times G^*$ *do množiny konečných podmnožin* $Q \times G^* \times D^*$
- $q_0 \in Q$ *je počáteční stav*
- $Z_0 \in G$ *je počáteční symbol zásobníku*
- $F \subseteq Q$ *je množina koncových stavů*

Abstraktní syntaktický strom

Abstraktní syntaktický strom je zakořeněný strom, který reprezentuje zdrojový program jako strukturu složenou z operátorů a jejich operandů. Vnitřní vrcholy jsou operátory a jejich následníci jsou jejich operandy [3]. Ukázku lze vidět na obrázku 2.1



Obrázek 2.1 Ukázka AST

2.2 LL(k) parser

▶ **Definice 2.22** (Formální překlad [1]). *Mějme formální překlad Z mezi formální jazyky L a V, pak platí $Z \subseteq L \times V$.*

▶ **Definice 2.23** (Překladová gramatika [2]). *je pětice $PG = (N, T, D, R, S)$, kde*

- *N je konečná množina neterminálních symbolů*
- *T je konečná množina vstupních symbolů*
- *D je konečná množina výstupních symbolů*
- *R je konečná množina pravidel tvaru $A \to \alpha$, kde $A \in N, \alpha \in (N \cup T \cup D)^*$*
- *S je počáteční symbol gramatiky*

▶ **Definice 2.24** (Vstupní a výstupní homomorfismus [2]). *Mějme překladovou gramatiku $PG = (N, T, D, R, S)$, pak vstupní homomorfismus h_i a výstupní homomorfismus h_o jsou definovány takto:*

$$\forall a \in T \cup N : h_i(a) = a, \forall a \in D : h_i(a) = \varepsilon$$

$$\forall a \in D : h_o(a) = a, \forall a \in T \cup N : h_o(a) = \varepsilon$$

2.2.1 Atributované gramatiky a překlady

Jedná se o rozšíření standardních překladů, které jsou rozšířené o atributy. Atribut je veličina, která nabývá hodnoty z nějaké množiny a ke každému symbolu se přiřadí konečná množina atributů. [2]

▶ **Definice 2.25** (Atributovaný překlad [2]). *je relace $Z_A = T^* \times D^*$, kde:*

- *T^* je množina vstupních atributovaných řetězců*

- D^* *je množina výstupních atributovaných řetězců*

▶ **Definice 2.26** (Atributovaná překladová gramatika [2])**.** *je čtveřice* $APG = (PG, A, V, F)$*, kde:*

- $PG = (N, T, D, R, S)$ *je překladová gramatika, pro kterou platí že všechny pravidla v* R *jsou ve tvaru* $X \to X_1 X_2 \cdots X_n$*, kde* $X \in N$ *a* $X_k \in (N \cup T \cup D)$ *pro každé* $k \in [1, n]$
- A *je konečná množina atributů, která se dělí na dvě disjunktní podmnožiny* S *(pro syntetizované atributy) a* I *(dědičné atributy)*
- V *je zobrazení, které ke každému neterminálnímu symbolu* $X \in N$ *přiřazuje množinu atributů, každému vstupním symbolu* $X \in T$ *přiřazuje množinu syntetizovaných atributů a každému výstupnímu symbolu* $X \in D$ *přiřazuje množinu dědičných atributů*
- F *je konečná množina sémantických pravidel.*
  Pro každý symbol na pravé straně pravidla je dáno právě jedno sémantické pravidlo, které vyjadřuje jeho dědičný atribut,

  $$d = f(a_1, a_2, \cdots, a_m)$$

  kde $a_1, a_2, \cdots, a_m$ *jsou atributy symbolů v témže pravidle.*
  Pro každý symbol na levé straně pravidla je dáno právě jedno sémantické pravidlo, které vyjadřuje jeho syntetizovaný atribut,

  $$s = f(a_1, a_2, \cdots, a_m)$$

  kde $a_1, a_2, \cdots, a_m$ *jsou atributy symbolů v témže pravidle.*

2.2.2 Syntaktická analýza shora dolů

Syntaktická analýza nám umožňuje vytvořit derivační strom ze vstupního řetězce bezkontextového jazyka. V případě analýzy z hora dolů se tvorba stromu provádí od počátečního symbolu (kořene) a uzly se doplňují zleva doprava. [3]

Pro rozpoznávání řetězce z bezkontextové gramatiky se používá zásobníkový automat. Jeho nedeterministickou variantu lze vytvořit z bezkontextové gramatiky následovně: [3]

▶ **Algoritmus 2.27** (Vytvoření zásobníkového automatu z bezkontextové gramatiky)**.** *[3]*
Vstup : Bezkontextová gramatika $G = (N, T, P, S)$
Výstup : Zásobníkový automat $R = (\{q\}, T, N \cup T, \delta, q_0, Z_0, F)$

1. *Zobrazení* δ *sestrojíme*
   - $\delta(q, \varepsilon, A) = \{(q, \alpha) : A \to \alpha \in P\}$ *pro všechna* $A \in N$ *(expanze)*
   - $\delta(q, a, a) = \{(q, \varepsilon)\}$ *pro všechna* $a \in T$ *(srovnání)*
2. *Počáteční stav* $F = q$
3. *Počáteční symbol je* S
4. *Množina koncových stavů je prázdná*

Derivační strom z takto vytvořeného zásobníkového automatu se vytváří podle pořadí provedených expanzí, při přijímání řetězce. [1]

2.2.3 Vlastnosti LL(1) jazyků

K efektivní implementaci překladů bezkontextových gramatik je nutné mít deterministickou variantu zásobníkového překladového automatu. Z toho důvodu musí naše gramatika splňovat několik podmínek. [2]

1. pro každé dvě pravidla $A \to \alpha, A \to \beta$ platí, že $h_i(\alpha) \neq h_i(\beta)$
2. Vstupní gramatika G_i gramatiky G je LL

Následně se musí vytvořit rozkladová tabulka pomocí funkcí FIRST a FOLLOW. V jejich definici využiji pouze G_i [2]

▶ **Definice 2.28** (First a Follow [2]). *Mějme gramatiku $G = (N, T, P, S)$, pak*

$$FIRST(\alpha) = \{a : \alpha \Rightarrow^* x\beta, x \in T, \beta \in (N \cup T)^*\} \cup \{\varepsilon : \alpha \Rightarrow^* \varepsilon\}, \alpha \in (N \cup T)^*$$

$$FOLLOW(A) = \{\omega : S \Rightarrow^* \alpha A\beta, \omega \in FIRST(\beta)\}, A \in N$$

▶ **Věta 2.29** (Podmínky pro LL(1) gramatiku [3]). *Aby bezkontextová gramatika $G = (N, T, P, S)$ byla LL(1), tak pro každé dvě pravidla $A \to \alpha, A \to \beta \in P$ musí platit*

- $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
- *pokud* $\varepsilon \in FIRST(\alpha) \wedge \varepsilon \notin FIRST(\beta)$ *pak* $FOLLOW(A) \cap FIRST(\beta) = \emptyset$

▶ **Definice 2.30** (Rozkladová tabulka [3]). *Mějme LL(1) gramatiku $G = (N, T, P, S)$ Rozkladovou tabulka je relace $M \subseteq N \times (T \cup \{\varepsilon\})$, pro kterou platí že [2]:*

- *je-li pravidlo i-té $A \to \alpha$ v množině P, pak $M(A, a) = i$, pokud $a \in FIRST(\alpha) \setminus \{\varepsilon\}$.*
- *je-li pravidlo i-té $A \to \alpha$ v množině P a zároveň $\varepsilon \in FIRST(\alpha)$, pak $M(A, b) = i$, pokud $b \in FOLLOW(A)$.*

Samotná syntaktická analýza se pak provádí stejně jako její nedertimistická varianta, ale pravidla se vybírají podle rozkladové tabulky. [3]

▶ **Algoritmus 2.31** (LL(1) syntaktická analýza). *[3]*

- *Konfigurace je trojice (x, α, π), kde $x \in T^*$, $\alpha \in (N \cup T)^*$ a π je posloupnost čísel*
- *Počáteční konfigurace je (w, S, ε), kde w je vstupní řetězec a S je počáteční symbol gramatiky*
- *Expanze : $(ax, A\alpha, \pi) \vdash (ax, \beta\alpha, \pi i)$, jestliže $A \in N, M(A, \alpha) = i$ a i-té pravidlo z P je $A \to \beta$*
- *Srovnání : $(ax, a\alpha, \pi) \vdash (x, \alpha, \pi)$, jestliže $\alpha in T$*
- *Přijetí : v konfiguraci $(\varepsilon, \varepsilon, \pi)$ analýza končí a π je levý rozklad vstupního řetězec*
- *Chyba ve všech ostatních situacích*

2.2.4 Implementace pomocí rekurzivního sestupu

Rekurzivní sestup je způsob jak provést implementaci LL(1) syntaktické analýzy v programovacím jazyku. Jak již název napovídá, při tomto algoritmu se využívá procedur/funkcí, které se musí vytvořit pro každý neterminální symbol gramatiky.

Každá z funkcí obsahuje rozvětvení podle LL(1) tabulky, které vybere pravidlo, a následné volání procedur/funkcí, které odpovídají jednotlivým symbolům na pravé straně vybraného pravidla. [3]

Na konci rekurzivního sestupu se musí zkontrolovat, zdali je vstupní řetězec prázdný, pokud by se tak neudělalo, tak by parser mohl přijmout i řetězce, které pouze začínají řetězcem z jazyka. [3]

Jak by takový to algoritmus mohl vypadat, pro gramatiku 2.1 s LL(1) tabulkou 2.1, jsem uvedl pomocí „pseudo c“ ve výpisu kódu 2.2.

Gramatika $G = (\{S\}, \{(,)\}, P, S)$
1. S $\rightarrow$ (S)S
2. S $\rightarrow \varepsilon$

Výpis kódu 2.1 Gramatika pro ukázku rekurzivního sestupu

Tabulka 2.1 LL(1) tabulka pro ukázkovou gramatiku

	(	)	ε
S	1	2	2

```
void S() {
    if (znak() ==  '(') {
        // podle pravidla 1
        srovnej('(');
        S();
        srovnej(')');
        S();
        return;
    }
    else if (znak() == ')') {
        // podle pravidla 2
        return;
    }
    // epsilon část v tabulce
    return;
}

int main() {
    S();

    if (!konec()) {
        chyba();
    }
    else {
        prijmuti();
    }
    return 0;
}
```

Výpis kódu 2.2 Ukázka algoritmu rekurzivního sestupu

Kapitola 3
Šablonovací jazyky

Definice šablonovacího jazyka nebývá v literatuře formální [4]. Jedním ze způsobů, jak formálně vyjádřit vlastnosti šablonovacího jazyka uvedl Terence Parr v [4] tím způsob že šablonu lze vnímat jako řetězec výstupních literálů a výpočetních výrazů.

3.1 Dosavadní řešení

Tento problém je řešen mnoha jazyky, a ještě více implementacemi, z toho důvodu zde budu vyjmenovávat pouze výběr z těchto řešení, které reprezentují standardní praktiky v těchto jazycích.

3.1.1 Jinja2

Jinja je šablonovací jazyk, který je implementovaný v programovacím jazyku Python, jehož filozofie lze napsat následovně: Přestože aplikační logika, pokud je to možné, by měla být implementovaná v Pythonu, tak by neměla být práce návrháře šablony kvůli tomu složitější. [5]

Samotný návrh šablony využívá speciálních znaků, mezi kterými lze psát kód podobný Pythonu (Výpis kódu 3.1). V tomto kódu lze využívat data, které jsou šabloně předány. [5]

Mezi jeho funkcionality patří (podle [5]):

- Šablonovací dědičnost (Templating inheretence)
- HTML šablony mužou využít ochranu proti XSS (Cross Site Scripting)
- Šablony lze zkompilovat do Pythonu pomocí „just-in-time“ kompilace, nebo muže být zkompilován předem
- Sandbox prostředí
- Definování a import maker

Jinja je velmi mocný šablonovací jazyk, který je velmi podobný Pythonu. Často se používá v tandemu s webovým frameworkem django.

3.1.2 Tera

Tera je šablonovací jazyk napsaný v Rustu, který je velmi inspirovaný Jinjou (Výpis kódu 3.2). Stejně jako Jinja využívá Tera speciální znaky na vkládání kódu do šablony. Stejně jako Jinja má dědičnost šablon.[6]

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{ variable|escape }}</title>
  </head>
  <body>
  {%- for item in item_list %}
    {{ item }}{% if not loop.last %},{% endif %}
  {%- endfor %}
  </body>
</html>
```

Výpis kódu 3.1 Ukázka jazyka Jinja

```
{% if price < 10 or always_show %}
   Price is {{ price }}.
{% elif price > 1000 and not rich %}
   That's expensive!
{% else %}
    N/A
{% endif %}
```

Výpis kódu 3.2 Ukázka jazyka Tera

3.1.3 Haml

Haml je zkratka pro „HTML abstraction markup language“. Jedním z jeho základních principů je, že šablona by neměla obsahovat zbytečné opakování, kvůli tomu se velice liší od HTML, tím že neohraničuje části šablony pomocí dvou tagů, ale spoléhá na indentaci kódu. [7]

Tím se zásadně liší od již zmíněných jazyků, ty spíše vkládali kód do již popsané stránky kolem nich. V tomto ohledu ho preferuji oproti jazyku Jinja nebo Tera a inspiruji se tím v mém vlastním návrhu (Výpis kódu 3.3).

3.1.4 Pug

Pug je šablonovací jazyk implementovaný v Javascriptu a jehož použití je velmi ovlivněno spoluprací s Javascriptem. Šablona napsaná v tomto jazyku se nejdříve převede na funkci, které jsou pak předána data k samotnému vytvoření konečného výstupu. [8]

```
#content
  .left.column
    %h2 Welcome to our site!
    %p paragraph
  .right.column
    %p onother paragraph
```

Výpis kódu 3.3 Ukázka jazyka Haml

```
ul
  li Item A
  li Item B
  li Item C
```

Výpis kódu 3.4 Ukázka jazyka Pug [8]

```
Hello {{name}}
You have just won {{value}} dollars!
{{#in_ca}}
Well, {{taxed_value}} dollars, after taxes.
{{/in_ca}}
```

Výpis kódu 3.5 Ukázka Mustache [9]

Stejně jako Halm se jeho syntaxe liší více od HTML, než Tera a Jinja, a také se opírá o odsazení textu při vytváření vnitřních elementů (Výpis kódu 3.4). [8]

3.1.5 Mustache

Toto řešení se od již výše zmíněných odlišuje ve své filozofii, a to tak že ve svých šablonách omezuje použití logiky a zaměřuje se hlavně na zobrazení dat na stránce. Jeho syntaxe je uzpůsobena tomu, aby se v kódu nevyskytovaly přebytečnosti a data se vkládají podobně jako u Jinji pomocí speciálních znaků (Výpis kódu 3.5). [9]

3.1.6 Shrnutí řešerše

Dosavadní řešení mají různé postupy, jak převést data do HTML kódu a nejen do něj. Z této analýzy jsem se zaměřil na několik vlastností, a to například zdali jazyk pouze vkládá svůj kód do textu, který ho obklopuje, nebo vyjadřuje strukturu HTML jiným způsobem například indentací a rozsah funkcionalit v jazyce, nebo zdali se považuje za jazyk bez logiky jako Mustache [9].

Bohužel dosavadní řešení se přílišně nezaměřuje silným typováním a funkcionálním stylem programování. Většinou využívají především dynamického typování, nebo se typování přílišně nevěnují, a imperativního programování.

Kapitola 4

Návrh šablonovacího jazyka

V této kapitole se zabývám návrhem šablonovacího jazyka. V první části se zaměřím na základní vlastnosti a struktury jazyka, které v následné části definuji pomocí gramatiky.

4.1 Základní návrh

Mezi hlavní vlastnosti, které chci docílit v návrhu šablonovacího jazyka, patří možnost tvorby vlastní struktury pro definici HTML, z důvodu nezávislosti na HTML syntaxi a možnosti upravit syntaxi pro lepší zápis šablon. Dále jazyk bude obsahovat silným typovací systém a podporu pro funkcionální programování.

Veškeré následující objekty a struktury, kterým se budu věnovat v průběhu této části, rozlišují mezi velkými a malými písmeny (case sensitive). Toto rozhodnutí ulehčuje implementaci v jazycích, které mají stejnou vlastnost.

4.1.1 Struktura programu

Základní struktura celého programu se skládá ze čtyř částí, a to definice typu vstupu, definice funkcí, typů a aliasů, a samotné výrazy, podle kterých se generuje výstup programu. Všechny tyto části jsou dobrovolné a můžou být, kromě definice typu vstupu, který musí být první (pokud je zadán), v jakémkoliv pořadí a počtu. Toto je ukázáno ve Výpisu kódu 4.1

Část, která odpovídá typu vstupu, je zahájena klíčovým slovem `input` a následně se zadá samotný typ. Tento typ už musí být existující typ, to znamená, že definice nového vlastního typu není v této části podporovaná.

4.1.2 Definice typů

V základu jde napsat definice typů pomocí vlastního typu, který definuje množinu konstruktorů a jejich závislosti, primitivního typu, řetězce již existujícího typu, nebo kombinací výše zmíněných možností, což se využívá při definici struktury a funkce.

První z možností se skládá z definice konstruktorů, která jsou oddělené pomocí svislé čáry. Konstruktor se skládá z jeho jména a posloupnosti typů, na kterých je závislý (viz. `type` v 4.1).

Mezi definované primitivní typy patří celé číslo (`Int`), textový řetězec (`String`), znak (`Char`) a booleovská hodnota (`Bool`).

Struktury se definují jakožto množina dvojic jména a typu. Zápis se nachází mezi složenými závorkami a jednotlivé dvojice jsou odděleny pomocí čárek. Dvojice se definuje jménem a typem, které jsou odděleny pomocí dvojtečky (viz `alias` v 4.1).

```
input {x : Int, exist : Bool}

fn f n : (Int) -> Int = n + 1

alias Person = {age : Int, name : String}

type State = Open | Closed | Error String

if (f 5) > 5
    then "It works"
    else "It doesn't work"
```

Výpis kódu 4.1 Základní struktura programu v navrhovaném programovacím jazyce

Řetězec je zapsán jakožto typ, který je zaobalen do hranatých závorek například : `[{x : Int, y : Int}]`. Takto zabalen může být jakýkoliv již definovaný typ.

Poslední zmíněná možnost je definice typu funkce, ta se skládá ze dvou částí a to definice vstupních typů a definice výstupních typů. Tyto dvě části se oddělují pomocí šipky (`->`). Funkce může mít jen jeden výstupní typ, počet vstupních typů není omezen. Syntaxe zápisu vstupních typů je seznam typů oddělených čárkami a je ohraničen pomocí závorek, například : `((Int, String) -> String)`.

4.1.3 Klíčová slova

Jedná se o množinu identifikátorů, které jsou vyhrazeny pro jiný účel, než ostatní identifikátory, to znamená, že nelze vytvořit například funkci se jménem odpovídajícímu nějakému klíčovému slovu. Klíčová slova v sobě můžou obsahovat pouze malá a velká písmena, čísla a podtržítko. Mezi klíčová slova patří:

- input
- if, then, else
- case, of
- type, alias, fn
- and, or, xor
- False, True
- import

Identifikace klíčových slov probíhá již při lexikální analýze a v syntaktické analýze se ke klíčovým slovům přistupuje jakožto jednotlivé tokeny.

4.1.4 Výrazy

Jazyk podporuje několik typů výrazu:

- Větvící výrazy `if` a `case` (viz 4.1.5)
- logický výraz – definuje logické operace nad výrazy

```
(if 5 > 3 then 5 else 3) + 3
```

Výpis kódu 4.2 Ukázka výrazů v navrhovaném jazyce

- matematický výraz – definuje základní matematické operace se standardní prioritou operátorů
- volání funkce – obsahuje jméno a parametry
- volání indexu na řetězci
- přístup k hodnotě ze struktury pomocí jména
- celočíselný výraz
- výraz textového řetězce
- výraz obecného řetězce – každý podvýraz v řetězci musí být stejného typu například `[1, 2, 3]`
- speciální zápis obecného řetězce
- výraz definice hodnoty struktury
- definice anonymní funkce
- Výraz pravda a nepravda (`True`, `False`)

Výraz muže být jak na nejvyšší úrovni, tak součástí funkcí, nebo jiných výrazů, například `if` výraz může být součástí binárního výrazu (Výpis kódu 4.2). Pro každý výraz platí to, že má návratovou hodnotu a návratový typ. Zároveň mají některé výrazy podmínky pro své podvýrazy, těmto podmínkám se budu věnovat dále.

4.1.5 Větvící výrazy

Jazyk obsahuje dva základní větvící výrazy a to `if` a `case`. První se využívá pro rozvětvení do dvou větví výpočtu podle podmínky a druhý pro rozeznávání vzoru ve vstupní hodnotě. (Výpis kódu 4.3)

Výraz `if` je uvozen pomocí klíčového slova `if` po němž se zadává výraz, který slouží jakožto podmínka, podle které se bude větvit. Následně se musí uvést dvě větve, do kterých se bude výraz větvit, první se uvozuje pomocí klíčového slova `then` a druhý pomocí `else`, za oběma klíčovými slovy se následně nachází výraz, který odpovídá větvi. Obě větve musí být stejného typu a výraz podmínky musí být typu `Bool`.

Na rozdíl od `if` má výraz `case` možnost dělit běh programu do více než dvou větví. Toto dělění neprobíhá pomocí booleovské hodnoty, ale podle vzorů, které se aplikují na vstupní hodnotu. Syntaxe `case` výrazu se skládá z klíčového slova `case`, které je následováno vstupním výrazem a klíčovým slovem `of` (v tomto pořadí), následně se zadají jednotlivé vzory a výrazy odpovídající jednotlivým větvím (viz `case` v 4.3).

Vzory se rozdělují do několika typů a to:

- Vzor vlastního typu
- Číselný vzor
- Podmínkový vzor
- Vzor, který přijme cokoliv (používaný pro záložní větěv)

```
type A = X Int | Y String | E

if 5 > 3
    then "A"
    else "B"


case (X 5) of
    X n -> toString n
    Y text -> text
    E -> "Empty"
```

Výpis kódu 4.3 Ukázka syntaxe rozvětvení pomocí `if` a `case` v navrhovaném jazyce

Tabulka 4.1 Operace a typy v matematickém výrazu

	sčítání	odčítání	násobení	dělení	modulo
Celé číslo	✓	✓	✓	✓	✓
Textový řetězec	✓				
Boolevká hodnota					
Obecný řetězec	✓				

Vzor a výraz, k němu přiřazený v `case` výrazu, se oddělují pomocí šípky `->` a stejně jako v `if` výrazu musí mít všechny větve stejný typ. V případě vzoru vlastního typu, lze do prostředí přiřazeného výrazu přidat hodnoty, na kterých je vlastní typ závislý, ke jménu, které je definováno v definici vzoru.

4.1.6 Logické a matematické výrazy

Logické operace, které jazyk podporuje, jsou `and` a `or`, obě jsou to binární operace. Pro oba z operátorů platí, že obě strany musí být typu `Bool` a výsledek je také `Bool`.

Matematické funkce lze rozdělit do dvou částí a to porovnávací a výpočetní, mezi porovnávací patří $=, <, \leq, >, \geq, \neq$, které jsou vyjádřeny v jazyce jakožto `==`, `>`, `>=`, `<=`, `/=`. Pro tento typ platí, že levý i pravý operand musí být stejného typu, a návratový typ je `Bool`.

Definované výpočetní operace jsou sčítání (+), odčítání (-), násobení (*), dělení (/) a modulo (%), pro všechny z těchto operací platí, že oba operandy musí být stejného typu. Tyto funkce nejsou definovány pro všechny typy operandů, například nelze vydělit dva textové řetězce, jaké dvojice operace a typu je možné použít lze vidět v tabulce 4.1. Precedence všech operátorů je vypsána v tabulce 4.2.

4.1.7 Volání funkcí a přístup k indexům a jménům

Volání funkce se skládá ze jména volané funkce, nebo výrazu, jehož výsledek je funkce, a argumenty pro vykonání funkce, které jsou od sebe odděleny mezerou (`funkce arg1 arg2`). Pro parametry platí, že musí být typu, který je definovaný v definici funkce (jak pojmenované, tak anonymní). Typ, který je výsledkem volání funkce, je dán v definici funkce a je zkontrolován podle vstupních typů do funkce a operací jaké se na nich ve funkci provádějí.

Syntaxe přístupu k indexu se skládá z výrazu, jehož hodnota musí být řetězec, a celočíselného výrazu, tyto dva výrazy jsou odděleny tečkou (`[1, 2, 3].1`). Podmínka pro výraz na levé straně

Tabulka 4.2 Precedence operátorů

Precendence	operátor	popis	asociativity
0	<> `seznam oddělených výrazů`	Operátor pro tvorbu obecného řetězce, který dovoluje zanoření (viz. 4.1.9)	Zprava do leva
1	a.b	Operátor, která umožňuje přistupovat k indexům, nebo jménům (viz. 4.1.7)	Zleva do prava
2	$a * b$	Násobení	Zleva od prava
	a/b	Dělení	
	$a\%b$	Modulo	
3	$a + b$	Sčítání	Zleva od prava
	$a - b$	Odčítání	
4	$a < b$	Menší než	Zleva od prava
	$a <= b$	Menší nebo rovno	
	$a > b$	Větší než	
	$a >= b$	Větší nebo rovno	
	$a == b$	Rovno	
5	a `and` b	Logická konjunkce	Zleva od prava
6	a `or` b	Logická disjunkce	

```
\x : (Int) -> Int = x * x

(\x : (Int) -> Int = x / 2) 5

map (\x : (Int) -> Int = x + 1) [1, 2, 3]
```

Výpis kódu 4.4 Ukázka syntaxe tvorby a využití anonymní funkce v navrhovaném jazyce

je jen to, že musí být typu řetězec, výsledek se odvozuje od typu, který řetězec obsahuje (například pokud se bude jednat o řetězec celých čísel, tak výsledek bude celé číslo).

Přístup k pojmenovaným hodnotám ze struktury je obdobný s přístupem k indexu z řetězce s tím rozdílem, že pravý operand není celočíselná hodnota ale identifikátor (`{name = "Adam", age = 22}.age`). Výsledek tohoto výrazu se pak odvozuje od typu pole se jménem, na které se dotazuje.

4.1.8 Anonymní funkce

Syntaxe tvorby anonymní funkce se skládá z uvozovacího zpětného lomítka, které je následováno vypsáním jmen vstupních proměnných, definice typu funkce a těla funkce (Výpis kódu 4.4). Tento konstrukt pak lze využít jakožto jakákoliv jiná hodnota. Součástí výsledné hodnoty je také prostředí, ve kterém se vytvořila, a to především aktuálně definované proměnné.

Takto vytvořené hodnota, lze využít k zavolání s později doplněnými parametry a to stejným způsobem, jakým by se zavolala standardní funkce.

4.1.9 Řetězce a hodnoty

Pro každý hodnotový a řetězcový výraz platí, že je ho hodnota je rovna jemu samotnému. Jazyk obsahuje několik druhů výrazů a to:

- Celočíselné hodnoty – podporuje decimální, hexadecimální a oktalový zápis

```
type Student =
    FirstYear {name : String}
    | SecondYear {name : String, failedLIN : Bool}
```

Výpis kódu 4.5 Ukázka zápisu definice vlastního typu a tvorba jeho hodnoty

- Textový řetězec – viz. podkapitola 4.2.1
- Booleovská hodnota
- Hodnota funkce
- Řetězec ostatních hodnot
- Hodnota vlastního typu
- Hodnota struktury

Hodnota funkce

Tato hodnota uchovává samotnou funkci a i prostředí, ve kterém byla funkce definována. Definice funkce se skládá z výčtu vstupních parametrů, typu funkce a samotného těla funkce, které je výraz. Prostředí tvoří seznam dvojic jmen a výrazů, které jsou přiřazeny k tomuto jménu.

Hodnota vlastního typu a záznam

V obou typech tvorby složitějších datových typů se uchovává seznam vnitřních hodnot. V případě záznamů se uchovává seznam jmen a k nim přiřazených hodnot. Syntaxe zápisu záznamu se skládá z ohraničujících složených závorek a výpisu dvojic jména a výrazu, jehož hodnota se jménu přiřadí, oddělených od sebe symbolem rovná se (=). Tyto dvojice jsou od sebe odděleny čárkou (viz. Výpis kódu 5.8).

Hodnoty vlastních typů si musí uchovávat, jak hodnoty vstupů, na kterých je závislý, tak i konstruktor, ze kterého se hodnota vytvářela. Pro tento účel se v hodnotě uchovává označení konstruktoru a seznam hodnot. Tvorba vlastního typu probíhá pomocí funkce, která je vytvořena z definice typu pro každý jeho konstruktor. Tato funkce má výstupní typ podle typu, z jehož konstruktoru byla vytvořena a má přednost před funkcí, která by měla stejné jméno, ale byla by vytvořena samostatně. (viz. Výpis kódu 4.5)

Řetezec ostatních hodnot

Pro zápis obecného řetězce, lze využít dva zápisy a to standardní zápis a zápis pomocí operátoru diamant, se kterým lze využít zanořování (Výpis kódu 4.6). Pro každou hodnotu řetězce musí platit, že její typ musí být stejný jako typ ostatních hodnot, obsahuje-li řetězec hodnoty typu T pak typ samotného řetězce je `TypeList T` a v případě, že řetězec je prázdný, má řetězec typ neznámý a lze ho dodefinovat z kontextu, ve které se nachází, například byl by jakožto návratová hodnota z funkce, která vrací řetězec celých čísel, tak se k němu přiřadí typ celočíselného řetězce.

4.2 Formální gramatika

V této části se zabývám návrhem formální gramatiky pro můj jazyk. Tato část se rozděluje do několika pod-částí, které odpovídají částem ze kterých se skládá kompilátor.

```
[0, 1, 1, 2, 3, 5, 8, 13]

<> 0, 1, 1, 2, 3, 5, 8, 13

<>
    1,
    2,
    3
```

Výpis kódu 4.6 Ukázka syntaxí zápisu řetězců navrhovaného jazyka

4.2.1 Tokeny

Výsledek lexikální analýzy je řetězec tokenů a jejím vstupem je textový řetězec, z toho lze vyvodit, že gramatika pro tuto část překladu bude obsahovat jakožto terminály všechny znaky UTF-8 sady a neterminály budou tokeny. Jazyk obsahuje několik tokenu:

- `<Eof>` – End of file, konec souboru
- `<Error>` – v případě erroru v lexeru
- `<Ident>` – identifikátor
- `<Keyword>` – klíčové slovo
- `<String>` – textový řetěžec
- `<Number>` – číslo v desítkovém zápisu
- `<Op>` – obecný operátor
- `<LBrac>`, `<RBrac>`, `<LCBrac>`, `<RCBrac>`, `<LSBrac>`, `<RSBrac>` – levé a pravé, normální, složené a hranaté
- `<Dot>` – tečka (”.”)
- `<Col>` – dvojtečka (”:”)
- `<ListStart>` – operátor pro speciální zápist pole
- `<Comm>` – čárka (”,”)
- `<Pipe>` – svislá čára (”|”)
- `<Arrow>` – operátor šipky
- `<Slash>` – Zpětné lomítko (”\”)
- `<Indent>` – zanořené v zarovnání
- `<Dedent>` – vynoření ze zanoření
- `<NewLine>` – začátek nové řádky

```
if 5 > 3
    then "5 je vetsi 3"
    else "Matematika se rozbila"

Výsledek :
Keyword If, Number 5, Op >, Number 3, NewLine, Indent, Indent, Indent, Indent,
↪  Keyword Then, String "5 je vetsi 3", Keyword Else, String "Matematika se
↪  rozbila", NewLine, NewLine, Dedent, NewLine, Dedent, NewLine, Dedent,
↪  NewLine, Dedent, NewLine, Eof
```

Výpis kódu 4.7 Ukázka výstupu lexikální analýzy pro navrhovaný jazyk

Definice tokenů které nejsou jednoznakové, jsou definované:
`<Keyword>` $\rightarrow$ `if|then|else|case|of|type|alias|input|and|or|xor|fn|True|False|import`
`<Ident>` $\rightarrow [a-z, A-Z][a-z, A-Z, 0-9, \_]^*$
`<String>` $\rightarrow$ "$[A-Z]^*$"
`<Number>` $\rightarrow [1-9][0-9]^* \mid 0[0-7]^* \mid 0[x, X][0-9, a-z, A-Z]^*$ (desítková, osmičková, šestnáctková soustava)
`<Op>` $\rightarrow +\mid -\mid *\mid /\mid <= \mid < \mid >= \mid > \mid == \mid =$
`<ListStart>` $\rightarrow <>$
`<Arrow>` $\rightarrow ->$

Poslední nedefinovanou věcí je výpočet tokenů zanoření (`<Indent>` a `<Dedent>`). To se musí vypočítat po každém ukončení řádku, a to tak že se porovná aktuální zanoření a nové zanoření (počet mezer na začátku řádky). Pokud se zanoření zvětší, tak se do řetězce tokenů přidá tolik `<Indent>` tokenů kolik je rozdíl, a naopak v případě snížení zanoření se přidá `<Dedent>` a `<NewLine>`, znova podle velikosti rozdílu. Ve výpisu kódu 4.7 můžete vidět příklad výstupu lexikální analýzy.

V případě, že by bylo možno jakožto další token interpretovat dva nebo více různých tokenů, tak se dává přednost tomu nejdelšímu možnému.

4.2.2 Gramatika pro syntaktickou analýzu

Jedná se o bezkontextovou gramatiku $G = (N, \Sigma, P, prog)$, kde:

- N je množina všech neterminálních symbolů viz. (Výpis kódu 4.8)
- Σ je množina tokenů (viz. podkapitola 4.2.1)
- P je množina pravidel viz. (Výpis kódu 4.8)
- prog je počáteční symbol gramatiky

prog → inputParse parseRest
inputParse → ε | <Keyword : input> typeParse
parseRest → anyExpression <NewLine>
anyExpression → typedefExpr | aliasParse | fundefExpr | expression | <NewLine> anyExpression
typedefExpr → funcType | arrayTypeParse | unionParse | recordParse | <Ident>
funcType → <LBrac> typeParse (<Comm> typeParse)* <RBrac> <Arrow> typeParse
arrayTypeParse → <LSBrac> typeParse <RSBrac>
unionParse → consParse consParse*
consParse → <Ident> consArgParse*
consArgParse → <Ident> | recordParse | arrayTypeParse
recordParse → <LCBrac> <Ident> <Col> typeParse (<Comm> <Ident> <Col> typeParse)* <RCBrac> | <LCBrac> <RCBrac>
fundefExpr → <Keyword : fn> <Ident> <Ident>* <Col> typeParse <Op : "="> expression
expression → logicexpr | ifexpr | caseexpr
logicexpr → logicexprOr logicexprprime
logicexprprime → <Keyword : and> logicexprOr logicexprprime | ε
logicexprOr → cmpexpr logicexprOrprime
logicexprOrprime → <Keyword : or> cmpexpr logicexprOrprime | ε
cmpexpr → mathexpr cmpexprprime
cmpexprprime → <Op : ">"> mathexpr cmpexprprime | ε
cmpexprprime → <Op : "<"> mathexpr cmpexprprime | ε
cmpexprprime → <Op : ">="> mathexpr cmpexprprime | ε
cmpexprprime → <Op : "<="> mathexpr cmpexprprime | ε
cmpexprprime → <Op : "=="> mathexpr cmpexprprime | ε
cmpexprprime → <Op : "/="> mathexpr cmpexprprime | ε
mathexpr → mathterm mathexprprime
mathexprprime → <Op : "+"> mathterm mathexprprime | ε
mathexprprime → <Op : "-"> mathterm mathexprprime | ε
mathterm → factor mathtermprime
mathtermprime → <Op : "*"> factor mathtermprime | ε
mathtermprime → <Op : "/"> factor mathtermprime | ε
mathtermprime → <Op : "%"> factor mathtermprime | ε
ifexpr → <Keyword : If> expression ifexprbody
ifexprbody → <Keyword : Then> expression <Keyword : Else> expression
ifexprbody → <Indent> ifexprbody <Dedent> <NewLine>
caseexpr → <Keyword : case> expression <Keyword : of> caserule*
caserule → pattern <Arrow> expression (ε + <Comm>)*

Výpis kódu 4.8 Pravidla gramatiky navrhovaného jazyka (část 1)

```
factor → factordotCall | factorcall | factorresCall | factorprim
factorprim → <Number> | <String> | <Keyword : True> | <Keyword : True> | <Ident> |
<Keyword : input> | factorarray | factorspecialArray | factorrecord | factorlambda
factordotCall → factordotIdent | factordotInt
factorcall → <Ident> factorcallargs
factorcallargs → factorprim factorcallargs | ε
factorresCall → <LBrac> expression <RBrac> factorcallargs
factordotIdent → factordotIdentSrc <Dot> <Ident>
factordotIdentSrc → factorrecord | <Ident> | <Keyword : input> | <LBrac> expression
<RBrac>
factordotInt → factordotIntSrc <Dot> <Number>
factordotIntSrc → factorarray | <Ident> | <Keyword : input> | <LBrac> expression
<RBrac>
factorarray → <LSBrac> <RSBrac> | <LSBrac> factorarrayInner expression <RSBrac>
factorarrayInner → expression <Comm> factorarrayInner | ε
factorspecialArray → <ListStart> factorspecialArrayBody
factorspecialArrayBody → factorspecialArrayElems | <Indent> factorspecialArrayBody
<Dedent> <NewLine>
factorspecialArrayElems → expression factorspecialArrayElems | expression <Comm>
factorspecialArrayElems | ε
factorrecord → <LCBrac> factorrecBody <RCBrac> | <LCBrac> <RCBrac>
factorrecBody → <Ident> <Op : "="> expression <Comm> factorrecBody | <Ident>
<Op : "="> expression
factorlambda → <Slash> <Ident>* <Col> typeParse <Op : "="> expression
```

Výpis kódu 4.9 Pravidla gramatiky navrhovaného jazyka (část 2)

Kapitola 5
Implementace překladače

5.1 Použité technologie

Pro implementaci překladače jsem zvolil programovací jazyk Haskell a to kvůli možnosti využít monadických parser kombinátorů, díky kterým lze vytvořit kód podobný zápisu formální gramatiky. Dále jsem velmi využil algebraických datových typů, které jsou velmi dobře podporovány v Haskellu. A zároveň jsem již měl zkušenosti v psaní Haskell programů.

5.2 Lexikální analýza

Z modulu lexer se exportuje funkce getTokens, jejímž vstupem je textový řetězec a vrací řetězec tokenů. Samotná implementace se skládá z přijmutí jednoduchých tokenů a následném rekurzivním volání a v případě složitějších tokenů se volají pomocné funkce, které odpovídají stavům konečného automatu, který přijímá jazyk tokenů a stejně jako jednoduchá tokeny se následně rekurzivně volá getTokens. Tento proces se ukončí na konci textového řetězce, nebo pokud při průběhu lexikální analýzy nastane chyba.

`Token` je datový typ implementovaný pomocí algebraického datového typu, který obsahuje konstruktory a jejich přiřazená data (například v případě celých čísel) a konstruktor, který odpovídá klíčovým slovům, k němuž je přiřazena proměnná typu `Keyword` (klíčové slovo). (Výpis kódu 5.1)

5.3 Implementace parseru

Parser se skládá ze dvou základních kroků a to z prvního průchodu, který vytvoří základní AST, ale nemusí znát všechny datové typy výrazů a druhého průchodu, který ověří validitu programu z hlediska použitých typů a jejich interakcí (například příkaz `if` musí mít stejný typ v obou větvích výpočtu).

5.3.1 První průchod

Prvním krokem je syntaktická analýza, která ověří zdali program odpovídá navržené gramatice. K tomu se využívá algoritmus rekurzivního sestupu, který je implementován s pomocí parse kombinátorů. K vytvoření kombinátorů je použito monád (Výpis kódu 5.2). Dále jsou v kombinátoru implementované logické funkce nad parsery jako je například or a and. Díky tomu lze tvořit kód, který se velmi podobá zápisu gramatiky (například Výpis kódu 5.3).

```haskell
data Keyword
    = KwDefn
    | KwIf
    | KwCase
    | KwOf
    | KwEq
    | KwType
    | KwAlias
    | KwIn
    | KwThen
    | KwElse
    | KwAnd
    | KwOr
    | KwXor
    | KwFn
    | KwTrue
    | KwFalse
    | KwImport
    deriving (Show, Eq)

data Token
    = TEof
    | TError String
    | TIdent String
    | TKw Keyword
    | TNumber Integer
    | TString String
    | TRNumber Double
    | TOp String
    | TLBrac
    | TRBrac
    | TLCBrac
    | TRCBrac
    | TLSBrac
    | TRSBrac
    | TDot
    | TCol
    | TComm
    | TPipe
    | TArrow
    | TSlash
    | TIgnore
    | TIndent
    | TDedent
    | TListStart
    | TNewLine
    | TInputStart
    | TInputEnd
    deriving (Show, Eq)
```

Výpis kódu 5.1 Datový typ Token a Keyword použitý pro výstup lexikální analýzy

```haskell
data Result a
    = Accept a [Token]
    | Reject String
    deriving Show

newtype Parser a = Parser {runParser :: [Token] -> Result a}

instance Functor Parser where
    fmap f (Parser p) =
        Parser $ \input ->
            case p input of
                Accept a rem -> Accept (f a) rem
                Reject msg -> Reject msg

instance Applicative Parser where
    pure a = Parser $ \input -> Accept a input
    Parser l <*> Parser r =
        Parser $ \input ->
            case l input of
                Reject msg -> Reject msg
                Accept res rem ->
                    case r rem of
                        Reject msg -> Reject msg
                        Accept res2 rem2 -> Accept (res res2) rem2

instance Monad Parser where
    return = pure
    Parser p >>= f = Parser $ \input ->
        case p input of
            Reject msg -> Reject msg
            Accept out rem -> (runParser (f out)) rem
```

Výpis kódu 5.2 Monadická datový typ Parser pro tvorbu parser kombinátorů

Dále se musí, v průběhu syntaktické analýzy, vytvořit AST a v případě chyby co nejlépe určit její místo a důvod. Tato chyba se ukládá do `Reject` možnosti typu `Result` (Výpis kódu 5.2).

5.3.2 Druhý průchod

Druhý průchod je určen ke zjištění typu výrazů, které nejsou známi během prvního průchodu a jejich kontrolu. Tento problém nastane například při použití funkce, která je definovaná dále ve zdrojovém kódu. Zároveň musí dosadit typy výrazům, které se skládají z více podvýrazu, jako je například `if` a zajistit validitu jeho typu.

K docílení tohoto výsledku se provede několik operací nad programem. Jako první se provádí substituce aliasů, během tohoto procesu se musí ověřit, zda se v definicích aliasů nenachází žádné cykly. Následně se provede vytvoření konstruktorů pro vlastní typy. Jako dalším se opraví typy parametrů funkcí a ověří se výstupní typy funkcí. Poslední krok zkontroluje typy operací jako je volání funkcí a výraz `if` a doplní se jejich typy. (Výpis kódu 5.4)

```
factorlambda :: Parser Expr
factorlambda = do
    _ <- ptok TSlash
    args <- prep pident
    _ <- ptok TCol
    typed <- typeParse
    _ <- ptok $ TOp "="
    e <- por (wrapIndent expression) expression
    return $ ExprValue (ValueFunction (FunDef args e typed) DMap.empty) typed
```

Výpis kódu 5.3 Příklad použití parser kombinátorů na části syntaktické analýzy, která odpovídá syntaxi anonymní funkce

```
fixtypes :: ProgExpr -> Either String ProgExpr
fixtypes prog = do
    prog1 <- fixaliases prog
    prog2 <- fixcustomtypes prog1
    prog3 <- fixparams prog2
    prog4 <- fixfndeftypes prog3
    return prog4
```

Výpis kódu 5.4 Hlavní část implementace druhého průchodu, ze které se volají ostatní operace

5.4 Implementace AST

Celé AST je implementováno pomocí algebraických datových typů. Kořen AST je typ „ProgExpr“ Výpis kódu (5.5). Dále se celý pýogram rozděluje na definice funkcí, typů a aliasů a samotné výrazy, které vytvoří výstup šablony.

Nejdůležitější částí AST je datový typ „Expr“ (Výpis kódu 5.6), který reprezentuje výraz. Tento datový typ obsahuje všechny možné konstrukty, které lze vytvořit v tomto jazyku.

```
data ProgExpr
    = ProgExpr
    { inputType :: TypeDef
    , types :: Map String TypeDef
    , aliases :: Map String (TypeDef, DFSState)
    , fundefs :: Map String FunDef
    , expressions :: [Expr]
    }
    deriving Show
```

Výpis kódu 5.5 Implementace, kořenového typu AST, ProgExpr

```
data Expr
    = ExprValue Value TypeDef
    | ExprUnion String [Expr] TypeDef
    | ExprRecord (Map String Expr) TypeDef
    | ExprArray [Expr] TypeDef
    | ExprBinOp OpType Expr Expr TypeDef
    | ExprCall String [Expr] TypeDef
    | ExprIndexCall Expr Integer
    | ExprIdentCall Expr String
    | ExprIf Expr Expr Expr TypeDef
    | ExprTypeDef String TypeDef
    | ExprAliasDef String TypeDef DFSState
    | ExprFnDef String FunDef
    deriving Show
```

Výpis kódu 5.6 Implementace Expr typu, který reprezentuje výrazy v kompilátoru

```
data Value
    = ValueNum Integer
    | ValueString String
    | ValueBoolean Bool
    | ValueArray [Value] TypeDef
    | ValueFunction FunDef (Map String FunDef)
    | ValueUnion TypeDef String [Value]
    | ValueRecord TypeDef (Map String Value)
    | ValueError
    deriving Show
```

Výpis kódu 5.7 Implementace Value typ, který reprezentuje všechny hodnoty

5.5 Datové typy a hodnoty

Jak je vidět na v AST, každý výraz (až na speciální výjimky, která nemají hodnotu sami o sobě) si pamatuje typ jeho výsledku (Výpis kódu 5.6). Některé typy jsou známy už při prvním průběhu a jiné se odvodí při druhém. Výsledky jednotlivých výrazů jsou hodnoty (Výpis kódu 5.7), pokud je výraz hodnota sama o sobě, jako například v případě čísla, tak se uchová přímo hodnota, která bude výsledek výrazu.

5.5.1 Primitivní datové typy

Jazyk nabízí několik základních datových typů, a to celé číslo(`Int`), booleovská hodnota(`Bool`), znak (`Char`) a textový řetězec(`String`).

5.5.2 Record

Jeden způsobů jak vytvořit spojení více typů jsou recordy (Výpis kódu 5.8), které spojí více hodnot do jedné. Jednotlivé položky se odlišují pomocí jejich jména a přístup k těmto hodnotám se využívá takzvaná tečková syntaxe (Výpis kódu 5.8).

```
{name = "Adam", age = 22}

{name = "Adam", numberOfLegs = 2}.name
```

Výpis kódu 5.8 Ukázka record

```
-- Přímý cyklus
alias A = {x : A}

-- nepřímý cyklus
alias B = (C) -> Int
alias C = [B]
```

Výpis kódu 5.9 Příklad možných cyklů aliasů, které mohou vzniknout

5.5.3 Aliasy

Alias je pouze přejmenování již existujícího typu a v průběhu druhého průchodu parseru proběhne nahrazování za reálné typy. Problém, který při tomto procesu může vzniknout, je cyklus v definici aliasů a to jak přímý, tak i nepřímý, jak je vidět ve výpisu kódu 5.9.

Kvůli tomu se musí zjistit zdali se od každého aliasu lze dostat ke kombinacím primitivních a vlastních datových typů, jinými slovy zdali, pokud by se aliasy dali do grafu závislostí, by se jednalo o strom. K dosažení tohoto cíle je použit upravený DFS algoritmus, který ověří, že v grafu závislostí neexistují cykly.

Následně se provede „rozbalení“, díky kterému pak je každý alias vyjádřen v primitivních nebo vlastních typech a jejich kombinacích. Tyto vyjádření aliasů se pak nahradí v místech programu, kde se tento alias použil (Výpis kódu 5.10).

5.5.4 Vlastní datové typy

Pro vytvoření datových typů se vytváří funkce, podle jmen a vstupních typů v konstruktorech (Výpis kódu 5.11). Tyto funkce se přidávají do seznamu funkcí v průběhu druhého průchodu ve funkci `fixcustomtypes` (Výpis kódu 5.4). Výsledek těchto funkcí je typu, do kterého patří konstruktor. Tato hodnota je implementována pomocí mapy a označení, kde mapa obsahuje data v hodnotě a označení uchovává, ze kterého konstruktoru hodnota vznikla (Výpis kódu 5.7).

5.5.5 Patterny (Vzory)

Vzory slouží k rozvětvování v `case` výrazu a rozdělují se do několika typů a to :

- Univerzální – přijímá jakoukoliv hodnotu
- Vzor vlastního typu – přijímá hodnotu vlastního typu podle jejího konstruktoru a vytváří prostředí z jejích pojmenovaných závislostí
- Číselný vzor – přijímá číslo podle jeho hodnoty

Tyto typy jsou vyjádřeny ve zdrojovém kódu pomocí algebraického typu (Výpis kódu 5.12). Samotné rozhodování při větvení se provádí v pořadí, v jakém byli vzory napsány, to znamená, že první vzor, který zadaná hodnota splňuje, je vybrán jakožto vzor, podle kterého se větví.

```
-- základ
alias A =
    { number : Int
    , child : B
    }

alias B = {arr : [Int], name : String}

fn getA n : (Int) -> A = {number = x, child = [1]}

-- rozbalení
alias A =
    { number : Int
    , child : {arr : [Int], name : String}
    }

alias B = {arr : [Int], name : String}

fn getA n : (Int) -> A =
    {number = x, child = {arr = [5], name = "Adam"}}

-- nahrazení
alias A =
    { number : Int
    , child : {arr : [Int], name : String}
    }

alias B = {arr : [Int], name : String}

fn getA n : (Int) ->  = {number : Int, child : {arr : [Int], name : String}}
    {number = x, child = {arr = [5], name = "Adam"}}
```

Výpis kódu 5.10 Ukázka rozbalování a nahrazování aliasů

```
type MyType =
    ConstrutorA Int OtherType
    | CostructorB String
    | EmptyConstructor

type OtherType = Other Int

ConstrutorA 5 (Other 3)
```

Výpis kódu 5.11 Ukázka vlastní typů v navrhovaném jazyce

```haskell
data Pattern
    = UnionPattern String [(String, TypeDef)]
    | NumberPattern Integer
    | CheckPattern FunDef
    | AnyPattern
```

Výpis kódu 5.12 Reprezentace vzorů ve zdrojovém kódu

```haskell
headEx :: [Value] -> Value
headEx [ValueArray (x:_) _] = x
headEx _ = ValueError

headType :: [TypeDef] -> TypeDef
headType [(TypeList t)] = t
headType _ = Unknown
```

Výpis kódu 5.13 Ukázka externí funkce

5.6 Standardní knihovna a knihovny pro šablonování

Velká část standardní knihovny je implementována pomocí externích funkcí, které jsou napsané v Haskellu a poskytují funkcionality, které nejsou tímto jazykem podporovány jako například generické datové typy. K napsání externí funkce je potřeba jak samotná funkční část, tak i funkce, která z typů parametrů vrátí výstupní datový typ (Výpis kódu 5.13). Poté se musí jen přidat do seznamu funkcí současně s jejím jménem, které bude využito k jejímu volání.

5.6.1 Šablonovací knihovny

Šablonovací knihovny obsahují datové typy a funkce, které jsou potřebné k vytvoření struktury cílového jazyka. Tyto knihovny se musí importovat do kódu.

Standardní HTML

V této knihovně se nachází funkce a datové typy, které jsou potřebné k vytvoření struktury jakéhokoliv HTML kódu. Základním datovým typem je vlastní datový typ `Attr`, který reprezentuje atributy přiřazené tagům. Dále se v knihovně nachází `alias Elem`, který odpovídá typu `(Attr, [String]) -> String`, tímto typem jsou označeny, všechny funkce, které tvoří HTML tagy.

Jelikož jedním ze vstupů funkce tagu je řetězec jejího výstupu, tak lze tvořit stromovou strukturu tagů a jak s pomocí standardního zápisu řetězce (`[a, b, c]`), tak i pomocí speciálního zápisu (`<> a, b, c`), který lze ohraničit zanořením. (Výpis kódu 5.14)

Jednoduché HTML

Cílem této knihovny je tvorby jednoduchých HTML stránek, které jsou tvořeny pouze tagy bez žádných přídavných atributů, s výjimkou atributů, které jsou vyžadované k základní funkci tagu (například `src` v `<a>`).

```
input {header : String, greet : String, list : [Person]}

import "lib/html.my"

alias Person =
    { name : String
    , age : Int
    }


fn showPerson person : (Person) -> String =
    li EA <>
        toString (person.age)
        ,span EA [person.name]

fn chapter name : (String) -> String =
    h1 (Class "chapter") [name]

chapter "Test"

div EA <>
    chapter (input.header)
    ,h1 (Id "header") <>
        "Header"
    ,p (Custom "style" "width=100%") <>
        input.greet
        ,"jak se mas"
    ,div (Comb [Id "main", Class "Body", Custom "style" "width=50%"]) <>
        h2 (Class "subsection") ["Subsection"]
        ,h3 EA ["Sub Subsection"]
        ,p EA ["Ja se mam dobre a ty?"]
    ,img (Custom "src" "img.jpg")
    ,ul (Class "list") (map (\x : (Person) -> String = (showPerson x)) (input.list))
```

Výpis kódu 5.14 Ukázka využití stardartní HTML knihovny

```
import "lib/basichtml.my"

div <>
    p <> "odstavec",
    span <> "kousesk textu"
```

Výpis kódu 5.15 Ukázka využití jednoduché HTML knihovny

```
import "lib/markdown.my"

fn range from to : (Int, Int) -> [Int] =
    if from >= to
        then [to]
        else [from] + range (from + 1) to

h1 "Chapter"
par (toString 1)
list (map (\x : (Int) -> String = toString x) (range 1 10))
h2 "Subchapter"
h3 "SubSubchapter"
```

Výpis kódu 5.16 Ukázka využití markdown knihovny

Pro tento účel obsahuje `alias Elem`, který se rovná `([String]) -> String`. Tento `alias` je typem pro všechny funkce, které reprezentují jednotlivé tagy. Stejně jako v případě Standardního HTML lze, díky struktuře typu, zanořovat tagy do tagů (Výpis kódu 5.15).

Markdown

Tato knihovna umožňuje vytvářet markdown pomocí navrhovaného jazyka. Obsahuje pouze základní funkce jako například tvorbu nadpisů, odstavců a listů.

Pro tento účel jsou vytvořeny funkce, které odpovídají jednotlivým elementům ve zdrojovém kódu. Tyto elementy přijímají data, které pak zobrazují v požadovaném formátu. 5.16

5.7 Testování

V této kapitole se věnuji způsobům testování implementace překladače.

5.7.1 Způsoby testování

Testování je rozděleno do dvou částí a to do jednotkových (unit) testů, které jsou vytvořeny pro všechny části překladače (lexer, parser a AST), a do integračních testů.

Samotná implementace je provedena s pomocí funkce `unless` ze standardní knihovny. (Výpis kódu 5.17)

Integrační testy jsou tvořeny pomocí kompilace celých šablon a kontroly, zda-li se výsledek neliší od očekávaného výsledku. Tyto testy jsou i randomizované pomocí vstupních parametrů do šablony.

```
lexingtest :: IO ()
lexingtest = do
    foldr (\(input, output) acc -> do {acc; basiclexer input output})
        (putStrLn "Lexing test")
        [ ("10", [TNumber 10])
        , ("0x10", [TNumber 16])
        , ("0x10 10", [TNumber 16, TNumber 10])
        , ("ident \"String\" 10", [TIdent "ident", TString "String", TNumber 10])
        ]

basiclexer :: String -> [Token] ->  IO ()
basiclexer input output = do
    unless ((getTokens input 0) == output) (error "lexing error")

main :: IO ()
main = lexingtest
```

Výpis kódu 5.17 Ukázka jednotkového testu

Kapitola 6
Závěr

Cílem této práce bylo navrhnout šablonovací jazyk, se zaměřením na funkcionální programování a vytvořit implementaci překladače pro tento jazyk. Návrh jazyka je popsán v kapitole 4, součástí jazyka jsou také konstrukty pro tvorbu funkcí vyššího řádu a anonymních funkcí. Dále byl navržen silný typový systém včetně primitivních datových typů, typů tvořených jakožto kompozice již existujících typů a možnost vytvářet vlastní typy.

Pro funkce standardní knihovny, které vyžadují generické typy, byly vytvořeny externí funkce, které nahrazují potřebu, pro definici generických typů (Výpis kódu 5.13).

Samotná implementace byla provedena i otestována pomocí jazyka Haskell. Následně byli vytvořeny knihovny pro tvorbu HTML (jednoduchá a standardní) a Markdown.

V budoucnosti by se návrh jazyka mohl rozšířit o podporu generických datových typů, interpolaci textových řetězců (string interpolation) a vytvoření dalších šablonovacích knihoven. Dále by jsem se chtěl věnovat vylepšení syntaxe typů a jejich inferenci.

Bibliografie

1. ELIŠKA, Šestáková. *Automaty a gramatiky: sbírka řešených příkladů*. České vysoké učení technické v Praze. Fakulta informačních technologií, 2020.
2. MELICHAR, Bořivoj. *Konstrukce překladačů*. Sv. 1. 1999. ISBN 80-01-02028-2.
3. KAREL, MÜLLER. *Programovací jazyky*. 2002. ISBN 80-01-02458-X.
4. PARR, Terence. *Enforcing Strict Model-View Separation in Template Engines*. Dostupné také z: `https://www.cs.usfca.edu/~parrt/papers/mvc.templates.pdf`.
5. *Jinja*. Dostupné také z: `https://jinja.palletsprojects.com/en/3.1.x/`.
6. *Tera*. Vincent Prouillet, 2017. Dostupné také z: `https://tera.netlify.app/`.
7. *Haml*. The Haml Team. Dostupné také z: `https://haml.info/`.
8. *Pug*. Dostupné také z: `https://pugjs.org/api/getting-started.html`.
9. *Mustache*. Dostupné také z: `https://github.com/Mustache/Mustache`.

Obsah přiloženého média

- `src`
  - `impl` — zdrojové kódy implementace
  - `thesis` — zdrojová forma práce ve formátu LaTeX
- `text` — text práce
  - `thesis.pdf` — text práce ve formátu PDF