VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Implementace jazyka TIL-Script

Implementation of the TIL-Script Language

Bc. Filip Peterek

Diplomová práce

Vedoucí práce: prof. RNDr. Marie Duží, CSc.

Ostrava, 2023

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Č.j.VSB/22/078945
vsboee2202603F

Zadání diplomové práce

Student: **Bc. Filip Peterek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Implementace jazyka TIL-Script
Implementation of the TIL-Script Language

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je implementovat funkcionální jazyk TIL-Script. Implementace bude vycházet z gramatiky jazyka, která je v souladu s logickým systémem Transparentní Intensionální Logiky (TIL).

Práce bude obsahovat:
1. Popis systému TIL, tj. jazyk konstrukcí a rozvětvená teorie typů.
2. Definici jazyka TIL-Script.
3. Analýzu, návrh a implementaci jazyka TIL-Script včetně typové kontroly.
4. Dokumentace celého projektu včetně uživatelské příručky.

Implementace bude provedena v jazyce C# nebo Java.

Seznam doporučené odborné literatury:

[1] Duží M., Materna P. (2012): TIL jako procedurální logika (průvodce zvídavého čtenáře Transparentní intensionální logikou). Aleph Bratislava 2012, ISBN 978-80-89491-08-7
[2] Duží M., Jespersen B. and Materna P. (2010): Procedural Semantics for Hyperintensional Logic. Foundations and Applications of Transparent Intensional Logic. First edition. Berlin: Springer, series Logic, Epistemology, and the Unity of Science, vol. 17, ISBN 978-90-481-8811-6.
[3] Ciprich N., Duží, M., Košinár M. (2009): The TIL-Script Language. Frontiers in Artificial Intelligence and Applications, Amsterdam: IOS Press, vol. 190, pp. 166-179.

17. listopadu 2172/15
708 00 Ostrava-Poruba
Česká republika

spojovatelka: +420 597 321 111
epodatelna: epodatelna@vsb.cz
ID datové schránky: d3kj88v

IČ: 61989100
DIČ: CZ61989100

email: studijni.fei@vsb.cz
www.fei.vsb.cz

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **prof. RNDr. Marie Duží, CSc.**

Datum zadání: 01.09.2022

Datum odevzdání: 30.04.2023

Garant studijního oboru: prof. RNDr. Václav Snášel, CSc.

V IS EDISON zadáno: 07.11.2022 11:59:22

Abstrakt

Cílem práce je implementovat programovací jazyk TIL-Script. Jazyk TIL-Script slouží jako výpočetní varianta logického kalkulu TIL, jenž umožňuje jednoduchý strojový zápis konstrukcí Transparentní intenzionální logiky, ale také jejich následné provedení. Práce dále řeší praktické problémy s interpretací jazyka TIL-Script, a to například definice pojmenovaných funkcí, interakce s databází, apod. Dále se práce snaží navrhnout nadmnožinu jazyka TIL-Script, která umožní konstrukce TILu nejen provádět, ale také analyzovat, vytvářet je, a pracovat s nimi.

Klíčová slova

Transparentní intenzionální logika, TIL-Script, překladač

Abstract

The goal of the thesis is the definition and implementation of the TIL-Script language. TIL-Script is a scripting language which serves the purpose of a computational variant of Transparent intensional logic, a logical calculus based on typed lambda calculi. TIL-Script allows for not just representation, but also execution of TIL constructions. This work also deals with practical problems of TIL-Script implementation, such as definitions of named functions, interaction with databases, and so on. Furthermore, this thesis attempts to define a superset of the TIL-Script language, which allows for not just the execution of constructions, but also for their creation and analysis.

Keywords

Transparent intensional logic, TIL-Script, interpreter

Obsah

Seznam použitých zkratek a symbolů

TIL	–	Transparentní intenzionální logika
JVM	–	Java Virtual Machine
JRE	–	Java Runtime Environment
JAR	–	Java Archive
TCO	–	Tail Call Optimization
REPL	–	Read-Eval-Print Loop
CLI	–	Command Line Interface
AST	–	Abstract Syntax Tree
GCC	–	GNU Compiler Collection
GNU	–	GNU is not Unix
RCE	–	Remote Code Execution

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Analýza přirozeného jazyka jako disciplína stále rychleji stoupá na oblibě i důležitosti. Jistě málokomu unikly například n-gramové[1] modely založené na predikci následujícího slova z předcházejících n slov, či vektorové modely jako Word2Vec[2], umožňující reprezentovat význam slov pomocí vektorů. Poslední dobou se velmi často mluví o jazykovém modelu GPT-3[3].

Výčet přístupů k analýze přirozeného jazyka však nekončí n-gramy a neuronovými sítěmi. K analýze přirozených jazyků lze přistoupit také pomocí logiky. Mezi průkopníky tohoto přístupu patřil také český logik Pavel Tichý, tvůrce Transparentní intenzionální logiky.

Transparentní intenzionální logika (dále také TIL) je logický systém založený na typovaném lambda kalkulu[4]. TIL umožňuje modelovat a analyzovat výrazy přirozeného jazyka za pomocí logiky. Jako doplněk TIL poté vznikl také funkcionální programovací jazyk TIL-Script[7], sloužící k interpretaci konstrukcí Transparentní intenzionální logiky. Syntax i sémantika jazyka TIL-Script jsou silně inspirovány Transparentní intenzionální logikou, ovšem přirozeně s určitými úpravami tak, aby byl TIL-Script rozumně zapisovatelný a interpretovatelný na počítači.

Právě interpretací jazyka TIL-Script se zabývá tento text. Součástí práce tak je navázání na předcházející vývoj jazyka TIL-Script, rozšíření gramatiky nutné pro akomodaci nových prvků jazyka (např. výraz *Import* sloužící k importu symbolů definovaných v jiném souboru), nebo také samotná interpretace jazyka TIL-Script, společně s kontrolou typové koherence. Dále práce navrhuje způsob, jak volat funkce platformy JVM. Tento přístup otevírá jazyku TIL-Script celý ekosystém platformy JVM. Práce tedy nijak neřeší a neimplementuje získávání informací z již existující databáze, soustředí se spíše na zjednodušení přístupu k existujícím knihovnám, aby do budoucna nebyl problém rychle a jednoduše naprogramovat přístup k libovolnému zdroji dat.

Nakonec se práce snaží navrhnout určitou nadmnožinu jazyka TIL-Script tak, aby TIL-Script mohl sloužit nejen k interpretaci konstrukcí TIL, ale také k jejich tvorbě a analýze. V současné době pro analýzu a práci s konstrukcemi jazyka TIL-Script existují nástroje psané v jazyce Java. Existuje-li však v současné době programovací jazyk vytvořený přímo pro logiky pracující s TIL, není důvod tento jazyk nevyužít a nerozšířit také o nástroje pro analýzu, tvorbu i transformaci

TIL-Script konstrukcí. Pro pokročilejší práci s jazykem TIL-Script by tak nebylo třeba učit se jiný programovací jazyk (Java), nebo využívat externí nástroje, ale naopak by byl k dispozici již familiérní nástroj, jehož základy jsou již příznivcům Transparentní intenzionální logiky dobře známy. Cílem této nadmnožiny ovšem není předefinovat TIL, či nějak výrazně měnit jádro jazyka TIL-Script, jinak by se ostatně také nemohlo jednat o nadmnožinu. Tyto nové prvky jsou od jádra jazyka TIL-Script, které slouží převážně jako výpočetní varianta TIL, jednoduše oddělitelné. Při práci s překladačem, jehož implementace je součástí této práce, není problém se této nadmnožině vyhnout a nevyužívat ji. Také lze, na základě tohoto textu, implementovat překladač jazyka TIL-Script bez jakýchkoliv navrhovaných rozšíření.

Práce se do znatelné míry inspiruje výzkumem amerického profesora Johna McCarthyho a programovacím jazykem Lisp[13]. Jelikož Lisp vychází z lambda kalkulu, pracuje s parciálními funkcemi, a nerozlišuje mezi programem a daty (tedy Lisp programy můžou pracovat s, generovat a transformovat Lisp výrazy), lze mezi Lispem a Transparentní intenzionální logikou nalézt řadu podobností. Primárním odlišovacím znakem Transparentní intenzionální logiky je ovšem rigorózně definovaná hierarchie typů a silný důraz na typovou koherenci. Naprostá většina dialektů jazyka Lisp je naopak dynamicky typovaná.

Text samotný je rozdělen do čtyř částí. V první části budou představeny základy Transparentní intenzionální logiky, jejichž znalost je nutná k pochopení práce. Druhá část představí programovací jazyk TIL-Script a vyznačí změny a úpravy, které tato práce navrhuje. Třetí část popíše technické řešení a implementaci překladače. V poslední části poté bude zdokumentována standardní i matematická knihovna, které jsou součástí implementace. Dále bude poslední část obsahovat návod na použití, návod na implementaci vlastní knihovny, a nakonec také ukázky programů psaných v jazyce TIL-Script.

V terminologii programovacích jazyků a překladačů existuje spousta zavedených anglických názvů, jejichž český ekvivalent je méně zavedený, případně neexistuje vůbec. Proto bude práce ve spoustě případů uvádět také anglický ekvivalent k českému výrazu.

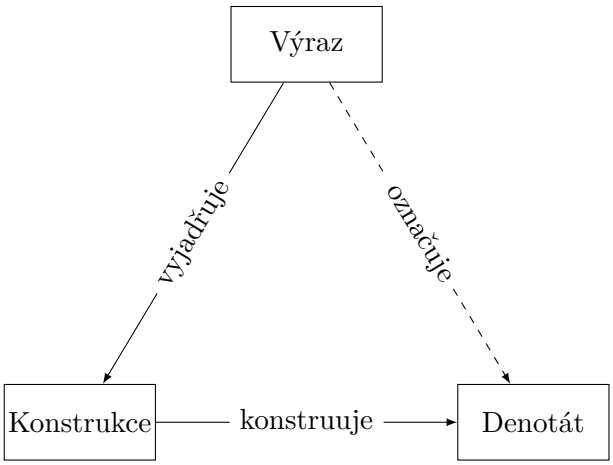
Kapitola 2

Transparentní intenzionální logika

Transparentní intenzionální logika (TIL) je logický systém založený na typovaném lambda kalkulu. TIL je využíván k logické analýze přirozeného jazyka. Oproti tradičnímu lambda kalkulu, jak jej definoval Alonso Church[14], jenž se využívá jako výpočetní model, tedy jako pouhý prostředek k dosažení konkrétní hodnoty – výsledku, v Transparentní intenzionální logice hraje konstrukce kalkulu často důležitější roli, než hodnota, kterou by konstrukce po provedení zkonstruovala[4].

Jako příklad využití lambda kalkulu jako výpočetní model lze uvést např. funkcionální programovací jazyk Haskell. Interně je Haskell kompilován do lambda kalkulu (přesněji do jeho nadmnožiny obsahující např. čísla a logické hodnoty, která jinak v lambda kalkulu musíme kódovat pomocí Churchova kódování – K-kombinátorů[14], apod.). Ultimátně v Haskellu ovšem lambda kalkul slouží pouze jako prostředek k získání konkrétního výsledku. Nadefinujeme vztah mezi vstupem a výstupem, a program napsaný v Haskellu nám vstup transformuje. Pokud zanedbáme efektivitu programu, nezajímá nás, jakým způsobem program spočítal výsledek, dokud jej spočítal správně.

Naopak Transparentní intenzionální logika je hyperintenzionálním kalkulem, který nám umožňuje vytvářet konstrukce vypovídající o jiných konstrukcích. TIL vychází z myšlenky, že výraz přirozeného jazyka sice označuje denotát – konkrétní objekt (např. individuum, číslo, konstrukci), významem výrazu ovšem není samotný denotát, který ani nemusí nutně existovat. Význam výrazu je abstraktní procedura a lze jej zachytit konstrukcí. Daná konstrukce poté při provedení zkonstruuje denotát výrazu. Jako příklad lze uvést například výraz "francouzský král." V době psaní této práce Francie krále nemá. Denotátem výrazu je tak neobsazená individuová role – výraz tedy neoznačuje žádné individuum. Přesto výrazu "francouzský král" rozumíme, výraz má svůj význam, jen v současné době neuvádí žádnou osobu. A budeme-li chtít o významu výrazu "francouzský král" něco vypovědět, například že francouzský král je monarchou v čele Francie, daný monarcha nemusí existovat. Dále lze uvést například rozdíl mezi výrazy "logaritmus 1024 o základě 2" a "5 + 5". Denotátem obou výrazů je 10. Zadáme-li do interpretu Haskellu výrazy `logBase 2 1024` a `5 + 5`, získáme v obou případech stejný výsledek. V přirozeném jazyce ovšem chápeme značný rozdíl mezi oběma výrazy, ačkoliv mají stejný denotát. "Logaritmus 1024 o základě 2" vyjadřuje číslo, kterým



Obrázek 2.1: Schéma procedurální sémantiky TIL

musíme umocnit dvojku, abychom získali 1024. Výraz "5 + 5" očividně vyjadřuje úplně jinou matematickou operaci – sčítání, a jeho výsledek spočítáme jiným postupem. Jistě by bylo zvláštní říct, že žák první třídy základní školy počítá $log_2 1024$, když žák ve skutečnosti počítá příklad $3 + 7$, ačkoliv jsou tyto výrazy ekvivalentní.

Denotátem výrazu může být nejen objekt z báze, ale i konstrukce nebo funkce.

Jak již bylo zmíněno, Transparentní intenzionální logika vychází z typovaného lambda kalkulu, proto také každý objekt musí mít svůj typ. Dále je vždy dbáno na dodržení typové koherence. Pro správné pochopení TIL, a tedy i této práce, je tak nutné znát typovou hierarchii TIL.

2.1 Objektová báze

Objektová báze je kolekce vzájemně disjunktních neprázdných množin, které dohromady vymezují nulární funkce, se kterými budeme pracovat[4]. Bázi volíme dle potřeb konkrétní aplikace a univerza diskurzu. Například používáme-li TIL k logické analýze matematických vět, jako bázi lze zvolit například množinu celých čísel, množinu reálných čísel, a množinu pravdivostních hodnot. Musíme však vzít v potaz, že tato báze neobsahuje čísla komplexní.

Patří-li objekt x do bázové množiny α, říkáme, že se jedná o objekt typu α. K explicitnímu uvedení typu objektu x využíváme zápis x/α. Množinám tvořícím bázi lze říkat typy.

Pro analýzu přirozeného jazyka se většinou volí objektová báze skládající se z typů o, ι, τ, ω. Tyto typy jsou podrobněji popsány v tabulce 2.1 – Výchozí báze pro analýzu přirozeného jazyka.

2.2 Funkce

V některých logických systémech, například v predikátové logice, se jako základní molekulární typ využívají relace[5]. Funkce je poté speciální typ relace, která je zprava jednoznačná. V TIL je

Tabulka 2.1: Výchozí báze pro analýzu přirozeného jazyka

Typ	Popis typu
o	Množina pravdivostních hodnot
ι	Množina individuí (univerzum diskurzu)
τ	Množina časových okamžiků/reálných čísel
ω	Množina logicky možných světů

však základním molekulárním typem funkce. Chceme-li v TIL vyjádřit n-ární relaci nad množinou $\alpha_1 \times ... \times \alpha_n$, lze tak samozřejmě udělat definicí n-ární funkce z $\alpha_1 \times ... \times \alpha_n$ do o, která každému prvku z $\alpha_1 \times ... \times \alpha_n$ přiřadí pravdivostní hodnotu na základě toho, zda prvek do relace patří, nebo ne[4].

Na rozdíl od tradičního lambda kalkulu je Transparentní intenzionální logika kalkulem parciálních funkcí. Z parciality funkcí poté vyplývá další vlastnost TIL – arita funkcí není shora omezená. V lambda kalkulech totálních funkcí lze využít Schönfinkelovu redukci k redukci funkcí n-árních na unární za využití skládání funkcí. Tato redukce však není ekvivalentní, pracujeme-li s funkcemi parciálními[6].

2.2.1 Intenze a extenze

V TIL dále rozlišujeme funkce na tzv. *intenze* a *extenze*. Intenze jsou funkce z možných světů. Extenze jsou funkce, jejichž doménou množina možných světů není, a tudíž jejich funkční hodnota nezávisí na stavu světa[4].

Intenze jsou obecně funkce typu $(\alpha\omega)$ pro libovolný typ α. Nejčastěji se však jedná o funkce typu $((\alpha\tau)\omega)$, tedy funkce zobrazující možné světy do chronologií objektů typu α.

2.3 Konstrukce TIL

Konstrukce v Transparentní intenzionální logice jsou abstraktní procedury. Tyto procedury jsou strukturované – nejedná se o množiny, mají pevně danou strukturu, a na uspořádání případných podprocedur záleží. Tyto konstrukce lze podle definovaných pravidel provést. Provedením konstrukce získáme výstup, případně nezískáme nic. Konstrukce, které nekonstruují žádný výstup, nazýváme *nevlastní* (anglicky *improper*). V TIL pracujeme s šesti druhy konstrukcí. [4]

Jak již bylo zmíněno, konstrukce můžou v TIL operovat nejen nad objekty, které nejsou konstrukcemi, tedy nad objekty z báze a funkcemi, ale také nad jinými konstrukcemi. Konstrukce však může operovat pouze nad konstrukcemi nižšího řádu, než je konstrukce samotná, viz 2.5 – Rozvětvěná hierarchie typů. Každou podkonstrukci, kterou musíme provést při provedení konstrukce, nazýváme *konstituentem*. V TIL existuje šest různých druhů konstrukcí. Dvě atomické – mají pouze jeden konstituent, a to sebe samotné, a čtyři molekulární. Atomickými konstrukcemi jsou *Trivializace* a

proměnné. Mezi molekulární konstrukce poté řadíme *Kompozice*, *Uzávěry*, *Provedení* a *Dvojí Provedení*.

Proměnné jsou konstrukce, které na základě valuace v v-konstruují objekty. Skutečnost, že proměnná x v-konstruuje hodnotu typu α, značíme $x \rightarrow_v \alpha$.

Trivializace pro libovolný objekt X konstruuje samotný objekt X. Konstrukce Trivializace je v Transparentní intenzionální logice potřebná, neboť výchozím módem pro konstrukce je provedení. Samotná konstrukce X by tak byla automaticky provedena, a místo konstrukce X bychom dostali pouze její denotát. Pokud bychom chtěli zkonstruovat konstrukci X, musíme ji trivializovat. Tím se provede pouze konstrukce Trivializace. A protože Trivializace nemá jiný konstituent, než sebe samotnou, trivializovaná konstrukce X se tak neprovede. V literatuře se Trivializace X tradičně značí ${}^0X$. Alternativně se používá také zápis ${}'X$. Tento zápis je poté využit i v jazyce TIL-Script.[1] Trivializace taktéž bývá využívána ke konstruování hodnot, které nelze provést (objekty z báze, funkce) a tudíž je nelze zmínit netrivializované. [2]

Kompozice je procedura aplikace funkce na argumenty. Kompozice $[XY_1...Y_m]$ značí aplikaci funkce konstruované konstrukcí X na argumenty zkonstruované konstrukcemi $Y_1, ..., Y_m$. Pokud konstrukce X v-konstruuje funkci f, všechny podkonstrukce $Y_1, ..., Y_m$ v-konstruují hodnotu, a je-li funkce f na daných argumentech definovaná, Kompozice v-konstruuje funkční hodnotu na těchto argumentech. V opačném případě je Kompozice v-nevlastní.

Uzávěr $\lambda x_1...x_mY$ je konstrukce v-konstruující funkci. $x_1, ...x_m$ musí být navzájem různé proměnné, Y musí být konstrukcí. Konstruce uzávěru je velmi podobná abstrakci v lambda kalkulu, ze které také vychází. Na rozdíl od lambda kalkulu však v TIL může Uzávěr konstruovat funkce s aritou vyšší než jedna. Uzávěr nemůže být nikdy nevlastní, může však konstruovat tzv. *degenerovanou funkci*, tedy funkci, která je nedefinovaná na celém definičním oboru.

Provedení ${}^1X$ je konstrukce v-konstruující objekt konstruovaný konstrukcí X. Pokud je konstrukce X v-nevlastní, je provedení ${}^1X$ také v-nevlastní. Jelikož je však provedení výchozím módem pro objekty, většinou se explicitně neuvádí. Provést lze pouze konstrukce. Objekty z báze (tedy čísla, individua, apod...) či funkce nelze provést, jejich provedení nekonstruuje nic. Proto je potřeba tyto objekty vždy trivializovat.

Dvojí provedení ${}^2X$ je poslední z výčtu konstrukcí. Je-li X konstrukcí v-konstruující konstrukci Y, a v-konstruuje-li konstrukce Y objekt Z, pak ${}^2X$ v-konstruuje Z. V opačném případě je Dvojí provedení ${}^2X$ v-nevlastní.

Jiné konstrukce v Transparentní intenzionální logice neexistují.

[1] Trivializaci lze považovat za ekvivalent funkce `QUOTE` z jazyka Lisp.

[2] V jazyce Lisp čísla konstruují sebe samotné, tedy provedením čísla získáme zpět prováděné číslo. `1` a `'1` jsou tedy v Lispu ekvivalentní výrazy. V TIL však není možné, aby objekt konstruoval sám sebe, viz rozvětvená hierarchie typů 2.5 – Rozvětvěná hierarchie typů.

2.3.1 Princip kompozicionality

Princip kompozicionality je důležitým rysem Transparentní intenzionální logiky. Princip kompozicionality říká, že význam výrazu je jednoznačně určen významy jeho podsložek.

Z principu kompozicionality vyplývá, že je-li libovolný konstituent konstrukce X v-nevlastní a pro danou valuaci v nekonstruuje žádnou hodnotu, pak je v-nevlastní i konstrukce X. Právě tento důsledek principu kompozicionality bude v této práci velmi důležitý.

2.4 Typy 1. řádu

Definice je skoro slovo od slova převzata z knihy *TIL jako procedurální logika – Průvodce zvídavého čtenáře Transparentní intensionální logikou*[4]. Tato sekce slouží jako krátké vysvětlení základů Transparentní intenzionální logiky. Pro podrobnosti a důkladnější vysvětlení se čtenář může obrátit například na tuto knihu.

Nechť B je báze. Pak:

i) Každá množina z báze B je atomický typ řádu 1 nad B.

ii) Nechť $\alpha, \beta_1, ..., \beta_m (m > 0)$ jsou typy řádu 1 nad B. Pak soubor všech m-árních parciálních funkcí $(\alpha\beta_1...\beta_m)$, tedy zobrazení z $\beta_1 \times ... \times \beta_m$ do α, je molekulární typ řádu 1 nad B.

iii) Nic jiného není typem řádu 1 nad bází B.

2.5 Rozvětvěná hierarchie typů

Definice je opět skoro slovo od slova převzata z Průvodce[4].

Nechť B je báze. Pak:

2.5.1 T_1 (typy řádu 1)

Viz sekce 2.4 – Typy 1. řádu.

2.5.2 C_n (konstrukce řádu n)

i) Nechť x je proměnná v-konstruující objekt typu řádu n. Pak x je *kontrukce řádu n nad B*.

ii) Nechť X je prvek typu řádu n. Pak trivializace ${}^0X$, provedení ${}^1X$ a dvojí provedení ${}^2X$ jsou *konstrukcemi řádu n nad B*.

iii) Nechť $X, Y_1, ...Y_m$ jsou konstrukce řádu n nad B. Pak kompozice $[XY_1...Y_m]$ je *konstrukce řádu n nad B*.

iv) Nechť $x_1, ..., x_m$ jsou vzájemně různé proměnné a X je konstrukce řádu n nad B. Pak uzávěr $[\lambda x_1 ... x_m X]$ je *konstrukce řádu n nad B.*

v) Nic jiného není konstrukcí řádu n nad bází B než dle i)-v).

2.5.3 T_{n+1} (typy řádu n+1)

Nechť $*_n$ je kolekce všech konstrukcí řádu n nad B.

i) $*_n$ a každý typ řádu n jsou *typy řádu n+1 nad B.*

ii) Jsou-li $\alpha, \beta_1, ..., \beta_m$ typy řádu $n+1$ nad B, pak $(\alpha\beta_1 ... \beta_m)$, tedy kolekce parciálních funkcí, je *typy řádu n+1 nad B.*

iii) Nic jiného není typ řádu $n+1$ nad B než dle i) a ii).

2.6 Analytické a empirické výrazy

Výrazy přirozeného jazyka lze dělit na dva typy výrazů, a to empirické a analytické.

Analytické výrazy jsou výrazy takové, které označují extenze nebo konstantní intenze. Jedná se například o matematické věty nebo věty vyjadřující relaci rekvizity mezi vlastnostmi (např. věta "Všechny velryby jsou savci" je analytická a nutně pravdivá nezávisle na stavu světa, neboť existuje-li individuum, které je velrybou, pak bude vždy také savcem.)

Empirické výrazy naopak označují intenze, jejichž hodnota na stavu světa závisí. Abychom určili hodnotu dané intenze, musíme empiricky zkoumat stav světa v daném časovém okamžiku. Empirické zkoumání světa ovšem již není záležitost logiky.

Kapitola 3

TIL-Script

Nyní konečně přišel čas představit TIL-Script. TIL-Script je interpretovaný funkcionální programovací jazyk, do znatelné míry inspirovaný jazyky jako Haskell nebo Lisp. Syntax jazyka TIL-Script by měla co nejvíce připomínat syntaktické prvky Transparentní intenzionální logiky, aby pouhá znalost TIL stačila k okamžitému pochopení jazyka TIL-Script. Sémantika TIL-Script konstrukcí poté musí odpovídat sémantice TIL.

Tato kapitola je rozdělena do tří sekcí. V první sekci jsou popsány důležité základní rysy jazyka TIL-Script. Druhá sekce popisuje již existující jazykové prvky, případně také dokumentuje změny oproti předchozím iteracím jazyka. Poslední sekce navrhuje rozšíření jazyka TIL-Script.

3.1 Charakteristické rysy jazyka TIL-Script

Tato sekce popisuje charakteristické rysy jazyka TIL-Script v takové podobě, jakou nabývá v této práci. Pokud se v některém bodě TIL-Script neshoduje s Transparentní intenzionální logikou či předchozími verzemi jazyka TIL-Script, je rozdíl náležitě popsán a vysvětlen.

3.1.1 Lambda kalkul parciálních funkcí

3.1.1.1 Shora neomezená arita funkcí

Na rozdíl od lambda kalkulu ve své tradiční podobě, nebo například jazyka Haskell, v Transparentní intezionální logice není arita funkce shora omezená, viz 2.2 – Funkce. TIL-Script musí tento fakt reflektovat. Proto tento jazyk umožňuje definici i aplikaci funkcí libovolné (samozřejmě kladné) arity. Nevyužíváme zde tedy rozvíjení funkcí (anglicky *currying*)[6]. Zatímco např. v Haskellu jsou funkce arity dvě nebo vyšší automaticky rozvinuty na sérii několika unárních funkcí, jejichž oborem hodnot jsou unární nebo nulární funkce, v jazyce TIL-Script není arita nijak omezená.

3.1.1.2 Parciální funkce a respektování principu kompozicionality

Jelikož v TIL můžou být funkce parciální, musí i TIL-Script počítat s parcialitou funkcí. Dále musí TIL-Script respektovat princip kompozicionality, základní rys Transparentní intenzionální logiky. Jedním z důsledků principu kompozicionality je, že konstrukce, jejíž přinejmenším jeden konstituent je v-nevlastní, bude také nutně v-nevlastní. Reprezentaci stavu, kdy je parciální funkce aplikována na argumenty, na kterých není definována, se věnuje podsekce 3.3.7 – Hodnota *Nil* této kapitoly.

Jedinou výjimkou je funkce `IsNil`, jež vrací pravdivostní hodnotu `True`, pokud je její jediný argument `Nil`, v opačném případě je jejím výsledkem `False`. Tato speciální sémantika funkce `IsNil` , ačkoliv porušuje princip kompozicionality a vyžaduje aplikaci unární funkce na *nic*, je zvolena jako doplněk k funkci $Improper/(o*_n)$ definované v Průvodci čtenáře[4], a jako kompromis mezi dodržením principů TIL a umožněním zpracování chybových stavů.

3.1.2 Okamžité vyhodnocování

Ačkoliv je TIL-Script funkcionální jazyk, vyhodnocování výrazů probíhá okamžitě (tzv. *Eager evaluation*). Okamžité vyhodnocování je potřeba k zajištění respektování principu kompozicionality. Pokud by vyhodnocování bylo líné, programátor implementující TIL-Script funkci v jazyce Java by si mohl vyžádat vyhodnocení argumentu během aplikace funkce, zjistit, že funkce neobdržela jeden z argumentů, ale přesto by mohl tento fakt ignorovat a vrátit ze své funkce validní hodnotu. Tím by však došlo k aplikaci funkce na chybějící argument a také k porušení principu kompozicionality (konstrukce by mohla mít význam, ačkoliv je jedna z jejich podkonstrukcí v-nevlastní).

3.1.3 Neměnnost argumentů funkcí a objektů (*Immutability*)

Jelikož je TIL-Script funkcionální jazyk, jsou objekty a argumenty funkcí neměnné (*immutable*). Zatímco v imperativních jazycích, jako je třeba jazyk C++, není problém v těle funkce modifikovat argument, který funkce obdržela, v jazyce TIL-Script nic takového provést nemůžeme. Dále například nemůžeme modifikovat existující seznam. Pokud chceme transformovat již existující seznam, musíme vytvořit nový seznam a požadované hodnoty uložit v novém seznamu[1]. Podobně nemůžeme změnit hodnoty n-tice, můžeme však vždy vytvořit novou n-tici.

Hodnotu volné proměnné změnit lze, abychom mohli zkoumat konstrukce při různých valuacích v. Typ proměnné však změnit nelze.

Obdobně nelze redefinovat funkce (změnit typ či předpis funkce) nebo změnit typ symbolické hodnoty (viz 3.2.15 – Symbolické hodnoty).

[1] Neměnnost a induktivní definice seznamů umožňuje, aby dva různé seznamy rozdílně začínaly, od určitého prvku však sdílely stejný zbytek (`Tail`). Díky této optimalizaci se můžeme vyhnout zbytečnému kopírování seznamů.

3.1.4 Definice a deklarace symbolů

TIL-Script nově rozlišuje mezi deklaracemi a definicemi proměnných a funkcí. Deklarace pouze uvědomí překladač o existenci proměnné nebo funkce, nijak ale nedefinuje valuaci proměnné nebo sémantiku funkce. Deklarace umožňuje funkci či proměnnou zmínit (např. v trivializaci, v uzávěru), neumožňuje nám však proměnnou provést nebo funkci aplikovat – jak také, když neznáme hodnotu proměnné, případně sémantiku dané funkce. Provedení deklarované, avšak nedefinované proměnné je chybou, při které interpret ohlásí chybu a běh programu je ukončen. Deklarovat jeden symbol lze vícekrát, deklarace však nesmí být konfliktní a lišit se typy.

```
$ java -jar interpreter/build/libs/tilscript.jar examples/undef-var.tils
** Error **
(4, 1): myVar.
    ~~~ ^ ~~~
        Variable 'myVar' is declared but undefined
$ java -jar interpreter/build/libs/tilscript.jar examples/undef-fn.tils
** Error **
(2, 1): MyFn/(Int Int Int).
    ~~~ ^ ~~~
        Function MyFn is declared but undefined, application is impossible
```

Ukázka 3.1: Hlášení chyby při chybějící definici

Definice přiřadí proměnné valuaci, funkci sémantiku. Proměnné s řádnou definicí lze provést a můžou tak být konstituentem prováděné konstrukce. Funkce s řádnou definicí lze aplikovat. Funkce i proměnné lze definovat pouze jednou. Opakovaná definice je chybou a vyústí v předčasné ukončení programu.

Symbolické hodnoty, viz 3.2.15 – Symbolické hodnoty, lze pouze deklarovat.

Deklarace jsou automaticky odvozeny z definic. Proto, pokud je známa definice, není třeba dodávat také deklaraci. K interpretaci deklarací automaticky dochází před interpretací definic, aby byla umožněna např. definice vzájemně rekurzivních funkcí. Definice jsou interpretovány v takovém pořadí, v jakém jsou uvedené ve zdrojovém kódu.

Deklarace bez řádných definic na první pohled můžou působit zbytečně. K čemu může sloužit funkce, kterou nelze aplikovat? Nesmíme však zapomenout, že konstrukce TIL samy vyjadřují význam, a nemusí nutně být pouze provedeny. Provedením konstrukce sice dostaneme její denotát, ten nás ale ne vždy zajímá. Představme si tedy případ, kdy provádíme analýzu výrazu přirozeného jazyka. Výraz analyzujeme pomocí Transparentní intenzionální logiky a získáme konstrukci. S danou konstrukcí chceme dále pracovat a chceme ji strojově zpracovat. Její denotát nás ovšem nezajímá, zajímá nás pouze význam konstrukce – procedura. Současně daná konstrukce obsahuje funkci, jejíž definici vůbec neznáme, nebo ji známe, ale nejsme schopni ji strojově vyjádřit, nebo

nás pouze nezajímá. Jelikož víme, že konstrukci nebudeme provádět, a tedy nebudeme ani aplikovat funkci v ní zmíněnou, nepotřebujeme znát její přesnou definici. Stačí nám znát pouze její název a typ. Právě tehdy provedeme deklaraci funkce, pomocí které funkci přiřadíme název a typ, ale již nijak nedefinujeme sémantiku funkce.

Pro představu uveďme několik příkladů, kdy stačí dodat deklaraci (název a typ), protože nepotřebujeme definici (název, typ i sémantiku).

Představme si situaci, kdy analyzujeme větu "V minulosti se pro výpočet převrácené hodnoty druhé odmocniny využívala funkce *Fast Inverse Square Root*". Jistě pro analýzu věty budeme nutně potřebovat deklarovat funkci `FastInvSqrt/(Real Real)`. Implementaci funkce *Fast Inverse Square Root* v jazyce C lze jednoduše nalézt na internetu. Tato funkce se však již roky nevyužívá, neboť moderní procesory implementují výpočet druhé odmocniny i její převrácené hodnoty hardwarově. Současně na první pohled vidíme, že funkci `FastInvSqrt` nebudeme potřebovat aplikovat. Není tedy třeba na internetu dohledávat existující specifikaci funkce *Fast Inverse Square Root* a definovat danou funkci v jazyce TIL-Script. Současně pro výpočet převrácené hodnoty druhé odmocniny čísla existují efektivnější metody, proto danou funkci nepotřebujeme aplikovat ani v případě, kdy potřebujeme provést matematické operace (za předpokladu, že např. nezkoumáme chybu této funkce). Rozlišení definic a deklarací zde slouží převážně k ušetření práce.

Jako jiný příklad lze uvést tzv. *Halting problem*, tedy algoritmicky nerozhodnutelný problém, zda program někdy zastaví. Jistě si lze představit matematickou funkci, která rozhodne, zda program zastaví. Můžeme například seřadit všechny syntakticky korektní programy dle abecedy, následně je očíslovat, a vytvořit nekonečnou tabulku, která každému programu přiřadí hodnotu 1, pokud program zastaví, v opačném případě pak hodnotu 0. Jelikož však počítače mají pouze omezené zdroje, nekonečnou tabulku sestavit nelze. A jelikož, jak dokázal již Alan Turing, je tento problém algoritmicky nerozhodnutelný, nemůžeme ani sestavit algoritmus, který obdrží na vstupu zdrojový kód, a na základě analýzy zdrojového kódu rozhodne, zda program zastaví[12]. Přesto však funkci `Zastaví/(Bool Text)`, případně `Halts/(Bool Text)` můžeme potřebovat například k analýze věty "Programátor zkoumá, zda jeho program někdy zastaví." V tomto případě deklaraci bez definice využijeme proto, že korektní definice funkce `Halts/(Bool Text)` neexistuje.

Názvosloví *deklarace*, *definice* je převzato z programovacího jazyka C, kde deklarace pouze uvědomí překladač o existenci symbolu, definice poté přiřadí symbolu konkrétní hodnotu. Počet deklarací je shora neomezený, naopak definice může existovat nanejvýš jedna. Deklarace nedefinovaného symbolu není chybou, ovšem snaha nedefinovaný symbol využít (např. volání funkce, přístup k proměnné) vyústí v chybu při procesu linkování.

3.2 TIL-Script jako výpočetní varianta TIL

Tato sekce popisuje základní výrazy a konstrukce jazyka TIL-Script, které existovaly již v předchozích verzích jazyka. Pokud práce tyto výrazy nějakým způsobem upravuje, je úprava náležitě

popsána a zdůvodněna.[7][8]

3.2.1 Věty jazyka TIL-Script

V jazyce TIL-Script za věty (*sentence*) považujeme výrazy na nejvyšší úrovni v programu. Větou je tedy například konstrukce taková, že není podkonstrukcí jiné konstrukce než sebe samotné, ale také definice funkce, proměnné, typu, apod. Každá věta musí být ukončena terminátorem. Roli terminátoru zastává znak . (tedy ASCII tečka).

3.2.2 Atomické datové typy

Atomické datové typy v jazyce TIL-Script vycházejí z výchozí báze využívané v Transparentní intenzionální logice k analýze přirozeného jazyka, tedy z množin o, ι, τ, ω. TIL-Script ovšem rozlišuje mezi časy a reálnými čísly, a pro tyto hodnoty definuje dva nekompatibilní typy, mezi kterými neexistuje implicitní konverze. Dále TIL-Script využívá typ ν představující celá čísla. Nakonec TIL-Script pro názvy typů nevyužívá řecká písmena, která nelze prakticky a jednoduše zapisovat na většině rozložení klávesnic, ale anglická slova nebo zkratky. Názvy typů vždy začínají velkým písmenem.

Typ o představující pravdivostní hodnoty TIL-Script pojmenovává `Bool`. Hodnotami typu `Bool` jsou poté hodnoty `True` a `False`.

Typ ι, v jazyce TIL-Script `Indiv`, reprezentuje množinu individuí. Individua v Transparentní intenzionální logice považujeme za *holá* – žádnou netriviální vlastnost nemají nutně[4]. Všechny netriviální vlastnosti individuí jsou určeny stavem světa. Individuum samotné nemá žádnou inherentní valuaci. Slouží pouze jako unikátní indentifikátor. Obdobně hodnoty `Indiv` v jazyce TIL-Script nemají žádnou konkrétní reprezentaci. Typ `Indiv` je využíván v konjunkci se symbolickými hodnotami, viz 3.2.15 – Symbolické hodnoty. Tímto TIL-Script umožňuje uživateli referovat na konkrétní individuum pouze pomocí symbolického identifikátoru, aniž by individuím musely být přiřazeny arbitrárně zvolené konkrétní hodnoty.

Reálná čísla TIL-Script reprezentuje typem `Real`. V implementaci překladače vytvořeném v rámci této práce jsou reálná čísla interně reprezentována typem `double`. TIL-Script samotný žádné omezení na reprezentaci reálných čísel nestanovuje, prakticky však reálná čísla v současné implementaci reprezentujeme pomocí 64bitové reprezentace dle standardu IEEE 754.

Celá čísla TIL-Script reprezentuje typem `Int`. Obdobně jako u typu `Real` neexistuje omezení pro reprezentaci celých čísel. Interně je využíván datový typ `long`, jedná se tedy o 64bitové znaménkové číslo reprezentované dvojkovým doplňkem.

Množinu možných časů modelujeme typem `Time`. Pro interní reprezentaci časových okamžiků byl v této práci zvolen datový typ `long`. Uživatel se sám může rozhodnout, jak bude tyto hodnoty interpretovat. Ve standardní knihovně lze však nalézt např. funkci `Now`, jejíž aplikací získáme počet milisekund uplynulých od 1. ledna 1970.

Typ konstrukcí, v TIL denotován $*$, byl v jazyce TIL-Script přejmenován na `Construction`. V dřívějších verzích jazyka se pro typ konstrukcí také využíval znak $*$[8]. Změna na `Construction` byla provedena z důvodu zachování konzistence s ostatními typy, které v jazyce TIL-Script taktéž pojmenováváme anglickými slovy, a také z důvodu omezení ambiguity, aby se typ konstrukce nepletl s funkcí násobení. Zatímco typograficky korektní reprezentací násobení je využití znaku $\times$, na počítači násobení značíme $*$. Ačkoliv lze mezi funkcí násobení a typem konstrukcí odlišit na základě kontextu, v jakém se znak objevuje, odlišení již v názvu umožňuje uživateli jednoduše rozeznat typ konstrukce od násobení čísel okamžitě a bez bližšího zkoumání kontextu.

Dále byl TIL-Script rozšířen o atomický typ sloužící k reprezentaci textu. Tento typ je podrobněji popsán v sekci o rozšířeních jazyka, viz 3.3.3 – Typ *Text*.

3.2.3 Generický typ *Any*

V Transparentní intenzionální logice není neobvyklé definovat typově polymorfní funkce. Zvykem je označovat předem neznámé typy řeckým písmenem α. Obdobně TIL-Script umožňuje definici typově polymorfních funkcí. Generické typy v jazyce TIL-Script značíme slovem `Any`, okamžitě následovaným indexem polymorfního typu. Index je libovolné číslo z rozsahu $\langle 0; 2^{32} - 1\rangle$. Nenulové indexy nesmí začínat číslem 0.

Generické typy lze použít pouze v typech funkcí. Má-li více argumentů funkce stejný generický typ, tedy typ `Any` se stejným indexem, překladač při procesu typové kontroly zajistí, že argumenty, na něž je funkce za běhu aplikována, jsou stejného konkrétního typu.

3.2.4 Výrazy *TypeDef*

Výrazy `TypeDef` umožňují přiřadit již existujícímu typu alternativní jméno. `TypeDef` nevytváří nový typ, jedná se pouze o alternativní název. V průběhu typové kontroly jsou tato jména rekurzivně přepisována, dokud překladač nezjistí původní typ. Teprve poté porovnává původní typy, nezávisle na jejich alternativních názvech. Typové aliasy můžeme využívat například pro zkrácení zápisu komplikovaných molekulárních typů. Obdobné zkratky v TIL využíváme běžně.

Korektní syntaktický zápis `TypeDef` výrazu začíná právě klíčovým slovem `TypeDef`, následovaným novým názvem, operátorem přiřazení `:=`, a nakonec původním typem. Název musí začínat velkým písmenem.

```
TypeDef Float := Real.
TypeDef Property := (((Bool Indiv) Time) World).
```

Ukázka 3.2: Výraz TypeDef

3.2.5 Funkce

V matematice známe funkce jako jednoznačné zobrazení z množiny možných argumentů (definiční obor) do množiny možných obrazů (obor hodnot). V programovacích jazycích konstrukce nazývané funkcemi často nemusí být nejen jednoznačné (výstup nemusí odpovídat pouze argumentům), ale dokonce nemusí být ani zobrazením (nevrací žádnou hodnotu, v takovém případě většinou modifikují stav světa, jako příklad lze uvést funkce s návratovou hodnotou `void` v jazyce C).

V jazyce TIL-Script, obdobně jako v TIL, jsou funkce vždy zobrazením do určitého oboru hodnot. Díky parcialitě může být funkce degenerovaná, a v takovém případě nebude vracet hodnotu pro žádnou kombinaci argumentů. V takovém případě se však jedná o určitou formu chybového stavu, spíše než záměr, jak by tomu bylo v případě např. jazyka C.

Implementací TIL-Script funkce v jazyce Java lze vytvořit TIL-Script funkci, která bude modifikovat stav světa např. zápisem do databáze. Ačkoliv tomuto nelze zabránit, je na zvážení uživatele, zda je TIL-Script, tedy nástroj pro logickou analýzu přirozeného jazyka, také vhodný nástroj pro plnění databáze.

Arita funkce musí být vždy alespoň jedna.

Pro zápis typu funkce využíváme podobnou notaci jako v Transparentní intenzionální logice. Typy funkcí denotujeme kulatými závorkami. Uvnitř závorek uvedeme nejprve obor hodnot funkce, poté uvádíme postupně typy argumentů. Jediný rozdíl oproti TIL spočívá v nutnosti zapsat mezeru mezi názvy jednotlivých typů. Tedy ekvivalentem k typu $(o\nu\tau)$ by v jazyce TIL-Script byl typ `(Bool Int Real)`, případně `(Bool Int Time)`.

Funkce lze deklarovat uvedením názvu funkce následovaným lomítkem a jejím typem. Deklarujeme-li více funkcí stejného typu, můžeme uvést více názvů oddělených čárkami. Deklarací funkci přiřadíme pouze název a typ, ne sémantiku.

Pro definici funkce, tedy přiřazení sémantiky, byla přidána nová syntax popsaná v sekci 3.3.2 – Definice pojmenovaných funkcí.

```
Add, Sub, Mult, Div/(Int Int Int).
```

Ukázka 3.3: Deklarace funkcí

3.2.6 Literály

Názvosloví *literál* je přejato z jiných programovacích jazyků. V jazyce TIL-Script literály myslíme nefunkce, tedy členy množin tvořících bázi. Pomocí literálů lze uvádět celá i reálná čísla, pravdivostní hodnoty, a také text (viz oddíl 3.3.3 – Typ *Text* věnující se typu `Text`). Individua zmiňujeme pouze za využití symbolických hodnot, viz Symbolické hodnoty 3.2.15 – Symbolické hodnoty. Literály nikdy nesmíme zapomenout trivializovat.

3.2.7 Trivializace

Trivializace v jazyce TIL-Script slouží ke stejnému účelu jako v Transparentní intezionální logice. Na rozdíl od TIL ovšem Trivializaci denotujeme apostrofem, namísto nuly zapsané jako levý horní index.

```
'1         -- Trivializace konstanty typu Int
'3.14159   -- Trivializace konstanty typu Real
'['+ '1 '2] -- Trivializace kompozice
```

Ukázka 3.4: Příklad Trivializace.

3.2.8 Proměnné

Proměnné v jazyce TIL-Script opět slouží stejnému účelu jako proměnné v TIL, tedy v-konstruují hodnotu v závislosti na valuaci v.

Volné proměnné lze deklarovat bez přiřazení valuace, ale lze je také definovat a přiřadit jim konkrétní hodnotu. Proměnné deklarujeme, pokud nás nezajímá konkrétní valuace. Přiřazení hodnoty využijeme například v případě, kdy si chceme uložit výsledek drahé operace (např. čtení z databáze), nebo si třeba jen chceme zkrátit zápis dlouhé konstrukce, nebo když chceme pracovat s konkrétní valuací v. Vždy je třeba uvést typ hodnoty, kterou proměnná v-konstruuje.

Syntax pro deklaraci proměnné již existovala. Syntax pro definici proměnné (přiřazení hodnoty) je nově zavedená. Deklarovat můžeme více proměnných najednou, ale pouze za předpokladu, že se jedná o proměnné stejného typu. Pak stačí názvy jednotlivých proměnných oddělit čárkou. Přiřazujeme-li však proměnné valuaci, můžeme uvést vždy pouze jednu proměnnou.

Pro změnu hodnoty volné proměnné použijeme stejnou syntax jako při úvodním přiřazení hodnoty. Přiřazení hodnoty proměnné však není konstrukce, ale metavýraz, proto lze hodnotu proměnné vždy změnit pouze na nejvyšší úrovni v programu. Typ proměnné změnit nelze.

Deklarace proměnných jsou vždy provedeny hned po spuštění programu, nezávisle na tom, kde v programu jsou uvedeny. Přiřazení hodnot proměnným není prováděno přednostně, aby byla umožněna korektní redefinice proměnné. Hodnota proměnné je tedy změněna pouze a přesně na místě, kde je tak uvedeno v programu. Deklarace jsou však z definic vždy dedukovány automaticky – proměnná je přednostně deklarována při spuštění programu, i když její deklarace není uvedena explicitně, abychom proměnnou mohli zmínit.

Název proměnné musí začínat malým písmenem.

```
x -> Int.            -- Deklarace proměnné v-konstruující hodnotu typu Int
y, z -> Int.         -- Deklarace více proměnných najednou
pi -> Real := '3.1415. -- Definice proměnné pi aproximující hodnotu pi
['* pi '2].          -- Využití proměnné pi jako konstituent konstrukce
```

```
long_varName123 -> Int.
```

Ukázka 3.5: Příklad využití proměnných

3.2.9 Provedení

Provedení se sémantikou opět nijak neliší od svého ekvivalentu v Transparentní intenzionální logice. Syntax ovšem musela být kvůli praktičnosti upravena, a proto bylo upuštěno od pravých horních indexů. Provedení denotujeme `^1`. Pro Dvojí Provedení poté využíváme `^2`. Dřívejší verze jazyka TIL-Script definovaly i trojí až deváté Provedení. Protože však trojí a vícenásobné provedení není v praxi skoro vůbec potřeba (dle Průvodce TIL taková potřeba nenastala), a protože limit devátého Provedení byl poněkud arbitrární (proč ne například desetinásobné Provedení), je tato práce konzervativní a drží se definice Provedení z Průvodce.

Jelikož jsou všechny konstrukce standardně v módu Provedení, není třeba explicitní Provedení využívat příliš často. Většinou tedy budeme používat Dvojí Provedení.

```
^1 x.
^2['GetComposition argument1 argument2].
```

Ukázka 3.6: Příklad využití Provedení

3.2.10 Kompozice

Kompozice umožňuje aplikovat funkce na argumenty. Kompozice využívají stejnou syntax jako v TIL. Protože arita funkce musí být alespoň jedna, musí i Kompozice obsahovat alespoň dvě podkonstrukce – konstrukci konstruující funkci a alespoň jeden její argument. Počet argumentů, na něž funkci aplikujeme, musí odpovídat počtu argumentů funkce. Dále nesmí být žádný argument v-nevlastní (s výjimkou funkcí `If` a `IsNil`). V opačném případě k aplikaci funkce vůbec nedojde, neboť nemáme argumenty, na které bychom funkci aplikovali, a Kompozice je tak v-nevlastní.

```
['* '2 '6].
```

Ukázka 3.7: Příklad využití Kompozice

3.2.11 Uzávěry

Sémantika Uzávěrů je opět stejná jako v Transparentní intenzionální logice, syntax ovšem byla upravena. Zatímco v TIL často vypouštíme hranaté závorky, v jazyce TIL-Script musíme závorky zapsat vždy. Řecké písmeno lambda nahradil v jazyce TIL-Script znak zpětného lomítka.

Zpětné lomítko následuje seznam argumentů funkce konstruované Uzávěrem. V dřívějších verzích jazyka TIL-Script bylo možné typ argumentu v některých případech vynechat – existovala-li volná

proměnná se stejným názvem jako λ-vázaná proměnná v Uzávěru, a nebyl-li typ λ-vázané proměnné uveden explicitně, byl typ λ-vázané proměnné automaticky dedukován podle typu stejnojmenné volné proměnné. Tato práce však od této automatické dedukce upouští. Jelikož byly do jazyka TIL-Script přidány výrazy `Import` umožňující importovat volné proměnné z jiného souboru, uživatel jazyka by se mohl dostat do situace, kdy musí procházet několik importovaných souborů, jen aby zjistil typ λ-vázané proměnné. V případě, kdy máme více λ-vázaných proměnných, použijeme čárku pro jejich oddělení.

Za seznamem argumentů může nově následovat explicitní specifikace oboru hodnot konstruované funkce. Specifikaci oboru hodnot denotujeme symbolem `->`, který následuje název existujícího typu. Explicitní specifikace je nepovinná, může však sloužit k zdůraznění úmyslu uživatele, aby čtenář zdrojového kódu na první pohled znal typ funkce. Dále explicitní specifikace typu může pomoct při typové kontrole.

Nakonec je třeba uvést konstrukci, nad kterou abstrakci provádíme.

```
[\x: Int -> Int ['+ x '2]].
[\x: Int, y: Int -> Int ['+ x y]].
[\x: Int, y: Int ['+ x y]].

[[\x: Int, y: Int -> Int ['+ x y]] '2 '3].
```

Ukázka 3.8: Příklad využití Uzávěrů

3.2.12 Zkrácený zápis typu intenzí a extenzionalizace

V TIL často využíváme zkráceného zápisu jak pro typy intenzí, tak pro jejich extenzionalizaci. Pro hodnoty typu α závislé na světamihu zkracujeme zápis $((\alpha\tau)\omega)$ na $(\alpha_{\tau\omega})$. Obdobně pro extenzionalizaci intenze a využíváme zkráceného zápisu a_{wt} ekvivalentnímu konstrukci $[[aw]t]$, kde a je konstrukce konstruující funkci (často se jedná o Trivializaci), a $w \rightarrow_v \omega$, $t \rightarrow_v \tau$. Jedná se však pouze o notační zkratku – o dohodu, ne součást TIL.

Ekvivalentní zkratky můžeme využívat také v jazyce TIL-Script. Chceme-li specifikovat typ intenze, můžeme využít notační zkratku `@tw`. Zkratku `@wt` využijeme k extenzionalizaci intenze. Zkratka `@wt` vždy extenzionalizuje za využití proměnných `w` a `t`. Proměnná `w -> World` je součástí standardní knihovny, proměnnou `t -> Time` musí uživatel definovat sám. Zkrácenou notaci nelze využít s proměnnými jiného názvu.

```
Rektor/(Indiv Indiv)@tw. -- intenze typu (((Indiv Indiv) Time) World)
['Rektor@wt 'VSB]. -- extenzionalizace funkce Rektor/(((Indiv Indiv) Time) World)
                   -- a následná aplikace na argument VSB.
```

Ukázka 3.9: Příklad využití zkrácené notace

3.2.13 Seznamy

Seznamy se v Transparentní intenzionální logice příliš neobjevují. Při strojové analýze a zpracování je však vhodné mít k dispozici způsob vyjádření kolekce dat, proto TIL-Script seznamy obsahuje. Seznam představuje homogenní seřazenou kolekci potenciálně neomezené délky. Seznamy můžou obsahovat duplicity.

V této práci jsou všechny seznamy neměnné. Seznamy jsou definovány induktivně. Seznam je buď prázdný seznam, nebo *cons cell* skládající se z hlavičky (známé jako *head* nebo *CAR* z jiných jazyků) – prvního prvku v seznamu, a z podseznamu reprezentujícího zbytek seznamu (*tail*, *CDR*). Z definice tedy vyplývá, že vytvoření nového seznamu vložením prvku na začátek lze provést v konstantním čase. Vytvoření nového seznamu přidáním prvku na konec jiného seznamu naopak bude lineární vzhledem k velikosti původního kolekce. Tato implementace seznamu je volena zejména proto, že umožňuje snadnou iteraci pomocí rekurzivních funkcí. Stejnou implementaci seznamů využívají například jazyky Lisp, Haskell, a další[13].

Typ seznamu denotujeme slovem `List` následovaným kulatými závorkami, v nichž je uveden typ prvku ukládaného v seznamu.

Standardní knihovna obsahuje řadu funkcí pro práci se seznamy. Tyto funkce jsou v této kapitole pouze nastíněny při ilustraci využití seznamů, podrobněji jsou však zdokumentovány v kapitole popisující standardní knihovnu, viz 5.3 – Standardní knihovna.

Dále interpret implementuje syntaktický cukr pro jednodušší vytváření seznamů. Pro vytvoření seznamu stačí v kompozici aplikovat funkci `ListOf` na nenulový, avšak shora neomezený počet argumentů. Během parsování bude tato kompozice korektně přepsána na sérii kompozic využívajících funkci `Cons` k vytvoření tzv.*cons cells* – buněk seznamu. K žádné aplikaci funkce na libovolný počet argumentů tedy nedochází. Při vytváření seznamu pomocí `ListOf` musíme dodat alespoň jeden prvek seznamu, aby bylo možné správně dedukovat typ prvků ukládaných v seznamu. `ListOf` je pouze syntaktický cukr překládaný při procesu parsování. Pokud bychom chtěli generovat konstrukce konstruující seznamy dynamicky za běhu programu, museli bychom zřetězit aplikace `Cons`.

```
['ListOf '1 '2 '3].                  -- vytvoření seznamu
['Cons '1 ['Cons '2 ['ListOfOne '3]]]. -- ekvivalent předchozího řádku
                                     -- a ukázka přepisu ListOf
['Head ['ListOf '1 '2 '3]]           -- 1
['Tail ['ListOf '1 '2 '3]]           -- ['ListOf '2 '3]
```

Ukázka 3.10: Příklad využití seznamů

3.2.14 N-tice

N-tice doznaly oproti předchozím verzím jazyka TIL-Script změny. Dříve byly n-tice homogenní kolekce nespecifikované, avšak konečné délky. Roli homogenní kolekce však již zastává `List`, a v jazyce

TIL-Script neexistoval způsob, jak seskupit více hodnot jiného typu dohromady.

V matematice využíváme n-tice například k reprezentaci prvků kartézského součinu. N-tice lze definovat více způsoby, vždy jsou však uspořádané, konečné, a můžou být heterogenní (například pokud provádíme kartézský součin nad neidentickými množinami). Obdobnou roli plní n-tice i v jazyce TIL-Script. Délka n-tice je pevně daná, stejně jako typy hodnot v n-tici. Celý typ n-tice je tedy dán uspořádaným výčtem typů všech hodnot obsažených v této n-tici. Syntaktický zápis typu n-tice je podobný typu seznamu. Typ n-tice denotujeme slovem `Tuple`, následovaným kulatými závorkami. V závorkách ovšem uvádíme výčet typů. Jednotlivé typy oddělujeme čárkami. Délka n-tice je určena počtem uvedených typů.

Práce s n-ticemi je do jisté míry podobná práci se seznamy. N-tici vytvoříme aplikací variadické funkce `MkTuple`. Stejně jako u funkce `ListOf` se však jedná pouze o syntaktický cukr. Funkce `MkTuple` skutečně existuje, je to ale funkce binární, nikoliv variadická, a vytváří dvojice z dodaných argumentů. Pokud ovšem funkci `MkTuple` dodáme více než dva argumenty, parser aplikaci `MkTuple` rozepíše na jednu aplikaci `MkTuple` a následné řetězení funkce `PrependToTuple`. Funkce pro práci s n-ticemi jsou podrobněji popsány v uživatelské dokumentaci, viz 5.3 – Standardní knihovna.

```
['MkTuple '1 '2.0 'True].                -- vytvoření n-tice
['PrependToTuple '1 ['MkTuple '2.0 'True]]. -- ekvivalent předchozího řádku

x -> Tuple(Int, Real, Int) := ['MkTuple '1 '3.14159 '10].
```

Ukázka 3.11: Příklad využití n-tic

3.2.15 Symbolické hodnoty

Název *symbolické hodnoty* je v kontextu jazyka TIL-Script nový, jedná se však pouze o praktický způsob implementace entit, které již v jazyce TIL-Script existovaly. Entitu lze specifikovat uvedením jejího názvu následovaného lomítkem a typem entity. Tímto byl názvu přiřazen typ, ale ne konkrétní hodnota. Entity v předchozích verzích jazyka ovšem mohly reprezentovat jak funkce, tak konstanty. V nové verzi jazyka TIL-Script jsou entity na úrovni implementace děleny na *symbolické hodnoty* a funkce. Funkce tedy není *symbolickou hodnotou*. Pro běžného uživatele toto odlišení nemá znatelný dopad, bude se však týkat například programátorů implementujících nové TIL-Script knihovny. Název *symbolická hodnota* je opět inspirován jazykem Lisp, kde podobný koncept symbolů, tedy jmen bez hodnoty, již dlouhou dobu existuje.

Symbolické hodnoty využíváme, když potřebujeme o daném objektu referovat jménem, ale ne hodnotou. Jako příklad lze uvést například individua. Individua nemají žádnou inherentní hodnotu. Objekt typu ι je pouze identifikátorem, unikátním jménem. Vlastnosti individuí jsou intenze – tedy zda dané individuum má či nemá konkrétní vlastnost závisí na stavu světa v konkrétním časovém okamžiku. Dále můžeme zmínit například číslo π. Číslo π sice bezpochyby hodnotu má,

jedná se o reálné číslo. Jelikož je však π číslo iracionální, tedy číslo s nekonečným neperiodickým desetinným rozvojem, bohužel jej nemůžeme s absolutní přesností reprezentovat v počítači. V praktické implementaci se tak musíme spokojit pouze s aproximací čísla π, nebo právě se symbolickou reprezentací.

Symbolické hodnoty nelze provést. Chceme-li je zmínit, musíme využít Trivializaci. Dále nad nimi nelze provádět všechny operace, které jdou provádět s nesymbolickými konkrétními hodnotami. Například jak bychom mohli přičíst k jinému číslu číslo π, když neznáme přesnou hodnotu čísla π? Kde to však dává smysl, můžeme naprogramovat dodatečnou podporu pro symbolické hodnoty. Funkce `Sin`, `Cos` v matematické knihovně mají zabudovanou kontrolu, zda obdržely jako svůj argument symbol `Pi/Real`. Pokud ano, funkce vrátí korektní hodnotu. Obdobně je například výsledkem aplikace přirozeného logaritmu na symbol `E/Real` číslo 1.

```
Pi/Real.
['Sin 'Pi]. -- 0
['Cos 'Pi]. -- -1
['+ '1 'Pi]. -- Nil - Výsledek existuje, na počítači jej však nelze spočítat
```

Ukázka 3.12: Příklad využití symbolických hodnot

3.3 Rozšíření jazyka TIL-Script

Tento oddíl nastiňuje nadstandardní rozšíření jazyka TIL-Script. Cílem těchto rozšíření je udělat z jazyka TIL-Script nástroj pro analýzu a práci s konstrukcemi Transparentní intenzionální logiky. Každá změna a rozšíření je zdůvodněna.

3.3.1 Komentáře

Komentáře v jazyce TIL-Script plní stejnou roli jako v jiných programovacích jazycích. Jedná se tedy o text, který je překladačem ignorován. Komentáře tak můžeme využít například k popisu kódu. Komentáře existují pouze v jednořádkové podobě a denotujeme je, podobně jako v jazycích Haskell nebo SQL, dvěma pomlčkami (`--`). Všechny znaky za pomlčkami až po první znak odřádkování jsou překladačem automaticky ignorovány.

```
-- Následující konstrukce spočítá a vypíše kosinus pi
['Println ['Cos 'Pi]]. -- Prints -1
```

Ukázka 3.13: Příklad využití komentářů

3.3.2 Definice pojmenovaných funkcí

Syntax jazyka TIL-Script byla rozšířena o možnost definice pojmenovaných funkcí. Jedná se o způsob, jak funkci přiřadit jméno a následně na danou funkci odkazovat jménem, namísto uvádění stejného uzávěru vícekrát ve zdrojovém kódu. Definice pojmenovaných funkcí byla v jazyce TIL-Scriptu již dříve přidána ale následně byla zase odstraněna[8]. Dokud TIL-Script sloužil pouze jako notace pro zápis TIL konstrukcí, byl přínos této funkcionality diskutabilní. Pokud však chceme experimentovat s možnostmi využití jazyka TIL-Script také jako nástroje pro analýzu TIL konstrukcí, či importovat funkce z jiných souborů, definice pojmenovaných funkcí je nezbytná.

Definici funkce denotujeme klíčovým slovem `Defn`. Následuje hlavička funkce, symbol přiřazení (`:=`), a nakonec konstrukce předepisující sémantiku funkce. Syntax je odlišná od notace pro zápis uzávěrů. Důvodem pro tento rozdíl je grafické odlišení hlavičky funkce specifikující název, argumenty a typ výsledku od těla funkce (konstrukce). Dále tato notace omezuje počet vnořených závorek a speciálních znaků v těle funkce. Vazba argumentů funkce je samozřejmě ekvivalentní λ-vázání

Název funkce musí začínat velkým písmenem. Dále může obsahovat malá i velká písmena, čísla a podtržítka. Povolena jsou všechna písmena české abecedy (tedy například české *č* je povoleno, avšak například katalánské *ç* povoleno není).

```
-- Definice binární funkce
Defn Add(x: Int, y: Int) -> Int := ['+ x y].

-- Příklad definice rekurzivní funkce
Defn Sum(list: List(Int)) -> Int :=
    ['If ['IsEmpty list]
        '0
        ['+ ['Head list] ['Sum ['Tail list]]]].
```

Ukázka 3.14: Příklad definice funkcí

3.3.3 Typ *Text*

Typ `Text` slouží k reprezentaci textu. Opět se jedná o typ, který není příliš potřeba pro analýzu přirozeného jazyka, neboť v přirozené řeči často vypovídáme o individuích, tedy třeba článcích nebo knihách, málokdy však o samotných korpusech textu. V programovacím jazyku je však forma reprezentace textu nezbytná. Typ `Text` využíváme jak k reprezentaci delších textů, tak k reprezentaci jednotlivých znaků. TIL-Script tedy nezná koncept typu pro samostatné znaky (jako např. `Char` v jazyce Haskell), a `Text` není `List` znaků. Jedná se o atomický typ. Pomocí funkcí standardní knihovny však lze z textů extrahovat podřetězce (včetně jednotlivých znaků), konkatenovat řetězce, apod.

Literály typu `Text` uvozujeme uvozovkami. Hodnoty typu text musíme trivializovat. V literálech textů můžeme používat tzv. *escape sekvence*. Texty podporují standard Unicode. Interně jsou reprezentovány objekty typu `java.lang.String` a tudíž využívají kódování UTF-16 a jsou internovány.

```
text -> Text := '"Můj text".
text2 -> Text := '"Text na\ndva řádky".
['Println '"Výpis textu"].
```

Ukázka 3.15: Příklad využití typu Text

3.3.4 Výrazy *Import*

Výrazy `Import` umožňují importovat symboly definované v jiném souboru nebo Java archivu (*.jar* souboru). Výrazy `Import` jsou vždy interpretovány jako první při interpretaci souboru. Každý soubor můžeme importovat nanejvýš jednou v rámci jednoho souboru. Obsahuje-li soubor se zdrojovým kódem jazyka TIL-Script více importů stejného souboru, případně Java třídy, daný soubor či třídu importujeme pouze jednou, následující importy jsou ignorovány.

Každý soubor je interpretován pouze jednou. Pokud tedy více souborů importuje jeden soubor, tato sdílená závislost bude interpretována pouze při první importu. Při první interpretaci si překladač uloží všechny symboly definované v daném souboru (symbolické hodnoty, funkce, proměnné). Při následných importech stejného souboru jsou poté pouze importovány tyto symboly. Tedy hodnoty proměnných jsou spočítány pouze jednou, hodnota proměnné bude vždy poslední hodnota, která byla dané proměnné přiřazena při interpretaci souboru. Obsahuje-li importovaný skript konstrukce na nejvyšší úrovni v programu (tedy takové, že nejsou podkonstrukcí jiné konstrukce než sebe samotné), jsou dané konstrukce provedeny pouze při prvním importu programu. Při importu TIL-Script souboru je nejprve interpretován importovaný soubor, teprve poté jsou symboly definované v daném souboru importovány do kontextu skriptu, který daný soubor importoval.

Pro import souboru uvedeme nejprve klíčové slovo `Import`, následované cestou k požadovanému souboru. Cesta může být absolutní nebo relativní vzhledem k umístění aktuálně překládaného souboru. Cesta musí být uvozena uvozovkami. Importy se můžou nacházet pouze na nejvyšší úrovni v programu, neboť se jedná o metavýrazy, ne konstrukce.

Symboly můžeme importovat také z Java archivu. V takovém případě musí být daný Java archiv načten již při spouštění Java prostředí společně s kódem překladače. Načítání *.jar* souborů za běhu je z bezpečnostních důvodů velmi komplikované. Dále musí daný Java archiv obsahovat tzv. *registrátor* symbolů implementující potřebné rozhraní. Pro import souborů za využití registrátoru stačí místo cesty k TIL-Script souboru uvést plně kvalifikovaný název třídy registrátoru s předponou `class://`. Právě tato předpona značí, že se jedná o import z *.jar* souboru. Překladač poté vytvoří instanci určeného registrátoru a naimportuje soubory definované tímto registrátorem. Rozhraní registrátorů je popsáno v uživatelské dokumentaci.

Importy nejsou tranzitivní. Tedy importujeme-li soubor, který sám importuje jiné soubory, budou importovány pouze symboly definované v importovaném souboru, ne však symboly definované v souborech jím importovaných. Importy nesmí obsahovat vzájemně konfliktní definice.

```
Import "dependency.tils".
Import "relative/path.tils".
Import "/usr/local/lib/tilscript/path.tils".
Import "class://org.fpeterek.tilscript.math.Registrar".
```

Ukázka 3.16: Příklad využití výrazů Import

3.3.5 Typ *Type*

Typ `Type` slouží k reprezentaci typů a je primárně metaprogramovacím prvkem. V Transparentní intenzionální logice o typech vypovídat nemůžeme. Typově polymorfní TIL není momentálně předmětem výzkumu. Pro účely metaprogramování je však možnost reprezentace typů nezbytná. Typ `Type` umožňuje pracovat s typy jako s validními hodnotami jazyka TIL-Script. Typové reference, tedy hodnoty typu `Type`, lze využít například při tvorbě typově polymorfních funkcí, kdy je potřeba, aby se chování funkce řídilo typem argumentu, nebo například při psaní programů, které opravují nebo generují programy.

Pokud by TIL-Script typ `Type` neobsahoval, musel by být za účelem analýzy jazyka TIL-Script implementován parser jazyka TIL-Script v jazyce TIL-Script, nebo by uživatelé museli využívat jiné programovací jazyky. Proto tato práce navrhuje rozšíření jazyka TIL-Script o metaprogramovací prvky za účelem unifikace a zjednodušení nástrojů pro práci s Transparentní intenzionální logikou.

```
intType -> Type := 'Int.
typeOf5 -> Type := ['TypeOf '5].

-- Volba funkce na základě typu argumentu typově polymorfní funkce
[\x: Any1 -> Any1 [['If ['= ['TypeOf x] 'Int] 'DiscreteLog 'Log10] x]].
```

Ukázka 3.17: Příklad využití typových referencí

3.3.6 Typ *DeviceState*

Typ `DeviceState` využíváme jako obor hodnot funkcí závisejících na stavu zařízení, na němž běží překladač. Standardní knihovna definuje proměnnou `deviceState -> DeviceState`, kterou uživatel může využívat kdekoliv, kde je hodnota typu `DeviceState` vyžadována. Objekty typu `DeviceState` nemají žádnou inherentní hodnotu, obdobně jako objekty typu `World`. Jako příklad funkcí závislých na stavu zařízení lze uvést například funkci `Now` vracející aktuální čas, nebo funkci `Random`, jejíž výsledkem je náhodné číslo (v aktuální implementaci číslo pseudonáhodné, nic však

nebrání využití např. hardwarových generátorů náhodných čísel nebo `/dev/random`, pokud je to na dané platformě možné).

```
['Now deviceState].
['Random deviceState].
RandFromDevRandom/(Int DeviceState).
```

Ukázka 3.18: Příklad funkcí závislých na stavu zařízení

3.3.7 Hodnota *Nil*

Hodnota `Nil` slouží k reprezentaci stavu, kdy konstrukce nekonstruuje žádnou hodnotu. Je-li jedním z argumentů funkce `Nil`, je výsledek automaticky opět `Nil`. Pro vyvolání chybového stavu je možné `Nil` zmínit přímo v TIL-Script konstrukci. Trivializace `Nil` však není povolená gramatikou, neboť Trivializace nemůžou nikdy být nevlastní[4].

Specifikem implementace je, že hodnota `Nil` obsahuje metadata popisující kde a proč došlo k selhání a konstrukce nezkonstruovala žádnou hodnotu. Pokud konstrukce na nejvyšší úrovni v programu nezkonstruuje hodnotu a výsledkem jejího provedení je `Nil`, program skončí neúspěchem a uživateli je nahlášena chyba.

Příklad využití hodnoty `Nil` lze vidět v ukázce 3.19 – Příklad využití Nil. V ukázce počítáme výsledek $(lna) \div b$. Protože je však logaritmus nedefinovaný pro nekladné argumenty, a protože dělit nulou nelze, musíme zajistit, aby pro $a \leq 0$ nebo $b = 0$ byla kompozice v-nevlastní[2].

V ukázce využíváme funkci `Cond`. Funkce `Cond` je pouze syntaktický cukr umožňující zkrácený zápis pro vnořené aplikace funkce `If`. Funkce `Cond` přijímá sudý počet argumentů. Argumentem na liché pozici je vždy podmínka, argumentem na sudé pozici je konstrukce, která se vyhodnotí, pokud je podmínka pravdivá. Parser jazyka TIL-Script aplikaci funkce `Cond` pouze přepíše na postupné aplikace funkce `If`. Funkce `Cond` je podrobněji popsána v uživatelské dokumentaci, viz 5.3.1.2. Pokud není ani jedna z podmínek pravdivá, funkce `Cond` vrací `Nil`.

V ukázce tedy nejprve zkontrolujeme, zda platí $a > 0$. Pokud ne, Kompozice je v-nevlastní, tedy konstruuje `Nil`. Poté se ujistíme, že argument b je nenulový. Pokud není, opět vrátíme `Nil`. Nakonec nám postačí tzv. *catch-all* podmínka, tedy podmínka, která bude pravdivá vždy, protože již víme, že výpočet provést můžeme. Proto je poslední podmínkou pouze Trivializace `True`.

```
-- Pomocí syntaktického cukru Cond můžeme zkontrolovat
-- více podmínek v jedné Kompozici
-- Jedná se pouze o jednoduchou ukázku, v praxi Ln i dělení
-- vrátí Nil automaticky pro argumenty, na kterých nejsou definované
```

[2] Jedná se samozřejmě pouze o příklad, který byl vybrán tak, aby byl na první pohled zřejmý a pochopitelný. Dělení ve standardní knihovně i přirozený logaritmus v knihovně matematické automaticky vrátí `Nil` pro argumenty, na kterých nejsou definované, proto v praxi bude stačit provést kompozici `['/ ['Ln a] b]`.

```
['Cond
    -- Pokud je hodnota proměnné a nekladná, vrátíme Nil
    ['Not ['> a '0]] Nil
    -- Pokud je hodnota proměnné b rovná nule, vrátíme Nil
    ['= b '0]      Nil
    -- Jinak provedeme výpočet
    'True          ['/ ['Ln a] b] ].
```

Ukázka 3.19: Příklad využití Nil

V ukázce 3.20 – Příklad hlášení chyby lze vidět hlášení chyby. V závorce je nejprve uvedeno, kde ve zdrojovém kódu došlo k aplikaci funkce na argumenty, na nichž není definována. Následně je uvedena samotná konstrukce a funkce je graficky zvýrazněna. V ukázce je nejprve hlášena konkrétní chyba, tedy že došlo k dělení nulou, poté je ohlášeno, že program byl ukončen, protože konstrukce na nejvyšší úrovni zkonstruovala hodnotu `Nil`.

```
(1, 17): ['* ['+ '1 '2] ['/ '6 '0]].
                    ~~~ ^ ~~~
        Division by zero
** Error **
(1, 17): ['* ['+ '1 '2] ['/ '6 '0]].
                    ~~~ ^ ~~~
        Nil constructed by a top level construction. Aborting execution.
```

Ukázka 3.20: Příklad hlášení chyby

3.3.8 Struktury

Struktury v jazyce TIL-Script nám umožňují definovat nový molekulární typ. Struktury, obdobně jako n-tice, umožňují shlukovat více hodnot různorodých typů. Hlavní rozdíl mezi strukturou a n-ticí je, že definicí struktury vytvoříme nový typ.

Na rozdíl od n-tic, jejichž členy indexujeme číselně, atributy struktur jsou pojmenované a přistupujeme k nim jménem. Dále můžou být struktury definovány rekurzivně. Tato charakteristika umožňuje například reprezentovat grafy. Atributu struktury nemůže být přiřazena hodnota `Nil`.

Během typové kontroly jsou porovnávány nejen typy atributů struktur, ale také názvy porovnávaných struktur. Liší-li se struktury názvem, ačkoliv jsou ve všech ostatních ohledech stejné, jsou považovány za rozdílné typy. Struktury nám tedy, na rozdíl od n-tic, umožňují lépe využívat typového systému jazyka TIL-Script – definicí nového typu můžeme například zabránit nechtěné záměně hodnot, neboť by tuto záměnu zachytil překladač během typové kontroly.

Pro představu uveďme následující příklad. Potřebujeme v jazyce TIL-Script reprezentovat souřadnice. Dále pracujeme s dvěma typy souřadnic – s kartézskými a s polárními souřadnicemi.

Využijeme-li typ `Tuple(Real Real)` k reprezentaci souřadnic, může jednoduše dojít k záměně kartézských a polárních souřadnic. Pokud ale definujeme za účelem reprezentace souřadnic struktury, viz ukázka 3.21, nechtěné záměně souřadnic dokáže překladač zabránit během typové kontroly, protože oba druhy souřadnic reprezentujeme jiným typem.

```
-- Definice typu Cart reprezentujícího kartézské souřadnice
Struct Cart {
    x: Real,
    y: Real,
}.

-- Definice typu Polar reprezentujícího polární souřadnice
Struct Polar {
    distance: Real,
    angle: Real,
}.
```

Ukázka 3.21: Definice struktur reprezentujících souřadnice

3.3.8.1 Definice struktur

Strukturu definujeme klíčovým slovem `Struct`. Dále uvedeme jméno struktury, a do složených závorek seznam atributů. Nejprve vždy uvedeme název atributu, za dvojtečku pak jeho typ. Jednotlivé atributy oddělujeme čárkami. Název struktury musí být validním názvem typu – musí začínat velkým písmenem. Při pojmenovávání atributů se pak řídíme pravidly pro jména proměnných a začínáme vždy malým písmenem. Za posledním atributem může ale nemusí být čárka – tzv. *trailing comma*.

```
Struct Tree {
    value: Int,
    subtrees: List(Tree),
}.
```

Ukázka 3.22: Příklad definice struktury

3.3.8.2 Konstruktory struktur

Konstruktory struktur umožňují vytvářet objekty uživatelem definovaného typu – struktury. Pro konstruktory byla vytvořena nová syntax. Do složených závorek uvedeme nejprve název struktury (název typu), následně pak uvádíme konstrukce, které budou provedeny jako konstituenty konstruktoru. Výsledky provedení poté budou přiřazeny jednotlivým atributům struktury. Argumenty

konstruktoru musíme uvést v pořadí odpovídající definici struktury. Konstruktory nevyužívají typové reference, název konstruovaného typu tedy netrivializujeme, jedná se pouze o syntaktický prvek. Jinak jsou konstruktory ekvivalentní kompozicím – pokud konstruktor neobdrží alespoň jeden z požadovaných argumentů, je konstruktor v-nevlastní, jinak konstruktor konstruuje hodnotu specifikovaného typu.

Konstruktory můžeme využít také ke konstrukci prázdných seznamů. Konstrukce seznamů pomocí `ListOf` využívá typovou inferenci pro určení typu seznamu. Prázdný seznam však pomocí kompozice nelze vytvořit. Dále využití konstruktorů pro konstrukci prázdných seznamů umožňuje vyhnout se využívání typových referencí. Neprázdný seznam pomocí konstruktoru vytvořit nelze.

```
{List(Tree)}. -- Konstrukce prazdného listu
{Tree '0 {List(Tree) }}. -- Konstrukce objektu typu Tree

myTree -> Tree := {Tree
    '1
    ['ListOf
        {Tree '2 {List(Tree) }}
        {Tree '3 ['ListOf {Tree '4 {List(Tree) } }]}]}.
```

Ukázka 3.23: Příklad definice struktury

3.3.8.3 Přístup k atributům

Za účelem přístupu k atributům byl přidán syntaktický konstrukt `::`. Za název objektu, k jehož atributu přistupujeme, zapíšeme sekvenci `::` následovanou názvem požadovaného atributu. Přístupy k atributům lze řetězit, potřebujeme-li přistoupit k atributu atributu. Při parsování je přístupu k atributu přiřazena vyšší priorita než Trivializaci. Tedy například konstrukce `'object::attribute ::attribute` značí Trivializaci atributu `object::attribute::attribute`.

Prakticky je atribut pouze proměnná, ačkoliv se jedná o proměnnou ve strukturovaném složeném objektu. Proto i sémantika provedení atributu je stejná jako sémantika provedení proměnné. Přístup k neexistujícímu atributu je chybou. Provedení existujícího atributu (proměnné) vždy zkonstruuje hodnotu, neboť atributy nemůžou být v-nevlastní.

V ukázce 3.24 – Příklad přístupu k atributům objektu lze vidět přístup k atributu instance struktury (objektu typu definovaného uživatelem) v praxi. V ukázce pouze vypisujeme podstromy stromu definovaného v předchozí ukázce (viz 3.23 – Příklad definice struktury).

```
['Println myTree::subtrees].
```

Ukázka 3.24: Příklad přístupu k atributům objektu

Kapitola 4

Implementace

V této kapitole nastíníme implementační detaily překladače. Nejprve zmíníme využité technologie, poté popíšeme architekturu projektu, nakonec pak uvedeme zajímavější problémy, které se objevily při implementaci překladače, a jejich řešení.

4.1 Zvolené technologie

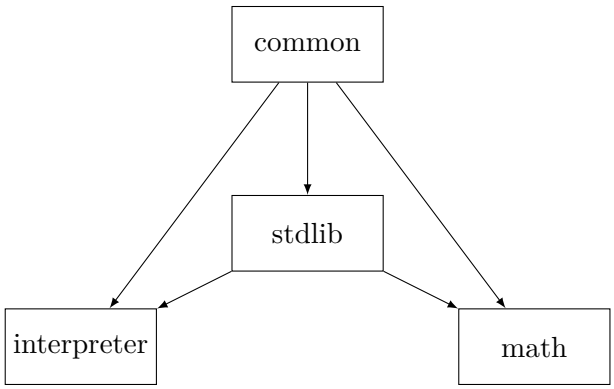
Celý projekt je implementován v jazyce **Kotlin** [1]. Jazyk Kotlin je staticky typovaný, multiparadigmatický jazyk kompilovaný do **JVM** bytekódu. Kotlin je vytvářen jako alternativa k jazyku Java, a nabízí plnou kompatibilitu s Javou. Využití Kotlinu umožňuje využívat veškeré výhody Java ekosystému, a také volat již existující Java kód. Tím můžeme využít například libovolnou již existující Java knihovnu.

Jazyk Kotlin byl pro implementaci překladače vybrán, protože umožňuje psát expresivnější kód, než by bylo možné v Javě. Dále jazyk Kotlin implementuje více statické analýzy, než jazyk Java. Null-safety a jazyková podpora pro algebraické datové typy tak umožňuje psát bezpečnější kód, než je možné v jazyce Java. Řadu chyb, kterých by bylo možné se dopustit v jazyce Java, dokáže kompilátor Kotlinu zachytit již během kompilace, a proto je vývoj projektu rychlejší a překladač obsahuje méně potenciálních problémů. Pro spuštění překladače je nutné mít na počítači nainstalované JRE.

Jako build systém byl zvolen projekt **Gradle**. Důvodem této volby je relativní jednoduchost konfigurace i využití systému Gradle, ale také přístup k Maven repozitáři s Java knihovnami. Dále využíváme několik Gradle pluginů nutných k sestavení projektu.

Parser generujeme za pomoci technologie **Antlr** ve verzi 4. Antlr je open-source generátor parserů podporující tvorbu parserů v řadě jazyků. V našem případě využíváme jako cílový jazyk Javu.

[1] Přestože je projekt psaný v Kotlinu, v práci často zmiňujeme např. *Java knihovny*, *Java objekty*, apod. Většinou tím myslíme Javu jako platformu. V takových případech poté nezáleží, zda používáme jazyk Java, Kotlin, nebo libovolný jiný jazyk kompilovaný pro platformu JVM. Mezi jazyky Kotlin a Java navíc existuje plná interoperabilita



Obrázek 4.1: Komponenty projektu

Interpret využívá knihovnu `org.apache.commons:commons-text` pro zpracování escape sekvencí. Tím výčet využitých technologií končí.

4.2 Architektura projektu

Celý projekt je rozdělen do čtyř komponent – společné knihovny (`common`), standardní knihovny (`stdlib`), překladače (`interpreter`) a matematické knihovny (`math`). Schéma projektu je znázorněno v obrázku 4.1, který vyjadřuje, jak na sobě jednotlivé komponenty projektu závisí.

4.2.1 Společná knihovna `common`

Knihovna `common` obsahuje kód společný pro zbytek projektu. Jedná se například o implementace tříd reprezentujících TIL konstrukce, definice společných rozhraní, reprezentaci typů, nebo pomocné funkce. Knihovna neobsahuje definice TIL-Script objektů, slouží ke sdílení kódu napříč jednotlivými komponentami. Využít ji tak může například programátor implementující novou TIL-Script knihovnu, konečného uživatele se však existence `common` nijak netýká.

Knihovna nemá žádné externí závislosti.

4.2.2 Standardní knihovna `stdlib`

Knihovna `stdlib` obsahuje implementaci standardní knihovny. `stdlib` se nedrží definice rozhraní, které musí implementovat TIL-Script knihovny implementované jako Java knihovny a distribuované jako Java archivy. Standardní knihovna je na úrovni překladače importována jiným způsobem, než ostatní knihovny. Standardní knihovnu ovšem není třeba importovat explicitně – import standardní knihovny implicitně zajistí překladač.

Standardní knihovna je nezávislá na použitém překladači jazyka TIL-Script. Interpret, jenž je součástí projektu, můžeme klidně nahradit novým překladačem (za předpokladu, že daný překladač implementuje potřebná rozhraní, např. `InterpreterInterface` definované v knihovně `common`).

Interpret samotný však na standardní knihovně závisí. Kvůli syntaktickému cukru (funkce `Cond`, `ListOf`, atd.), funkci `If`, jež musí být vyhodnocována líně, apod., musí překladač obsahovat speciální podporu pro standardní knihovnu.

Standardní knihovna definuje základní množinu funkcí, hodnot, typů a proměnných potřebnou pro práci s jazykem TIL-Script.

4.2.3 Matematická knihovna `math`

Matematická knihovna `math` slouží jako ukázková implementace TIL-Script knihovny v Kotlinu, případně v Javě. Dále je využívána k testování funkčnosti importování Java knihoven. Na rozdíl od standardní knihovny, překladač je naprosto nezávislý na knihovně `math`. `math` je třeba importovat explicitně pomocí výrazu `Import`.

Knihovnu nelze označit za extenzivní, obsahuje pouze malé množství funkcí, definice symbolických hodnot `E`, `Pi` a proměnných `e`, `pi`, aproximujících eulerovo číslo a číslo π.

4.2.4 Interpreter

Na rozdíl od předchozích komponent, které byly čistě Java knihovnami, Interpreter, tedy projekt obsahující interpret jazyka TIL-Script, je spustitelný Java program (tedy program s korektně definovanou funkcí `main`). Jedná se o referenční implementaci překladače jazyka TIL-Script. Překladač podporuje TIL-Script v takové podobě, v jaké je definován touto prací. Obsahuje základní nástroje pro hlášení chyb, aby ulehčil práci s jazykem TIL-Script. Typovou kontrolu provádí pouze za běhu programu.

4.3 Implementace překladače

V této sekci nejprve nastíníme fungování překladače. Poté popíšeme zajímavé problémy týkající se implementace překladače. Problémy můžou být zajímavé jak z hlediska řešeného problému, tak z hlediska budoucího rozvoje.

4.3.1 Vysokoúrovňový pohled na interpret

Překladač funguje čistě jako neinteraktivní textová aplikace. Programy, které potřebujeme interpretovat, předáváme překladači jako CLI argumenty. Pro překladač momentálně neexistuje žádné grafické rozhraní, ani REPL, který by umožnil interaktivní překlad TIL-Script výrazů. Pokud využíváme funkci `main` definovanou v JAR překladače, spustí se programy sekvenčně v takovém pořadí, v jakém je uživatel specifikoval. Pokud ale běh jednoho z programu skončí chybou, další programy se již nespouští. Pro každý program je však vytvořena nová instance překladače. Běhy jednotlivých programů jsou tedy na sobě nezávislé.

Pro strojovou práci se zdrojovým kódem musíme nejdříve daný kód převést ze sekvenční textové podoby do reprezentace, se kterou počítač umí pracovat lépe. Proto programy převádíme do stromové reprezentace nazývané abstraktní syntaktický strom (také AST – Abstract Syntax Tree). Tomuto převodu se říká *parsování*, případně anglicky *parsing*. Parsing (včetně tokenizace a lexikální analýzy) je v našem překladači delegován parseru automaticky vygenerovaném technologií Antlr. Pokud lexer nebo parser narazí na syntaktické chyby v programu, jsou dané chyby ohlášeny uživateli a překlad automaticky končí neúspěchem. Pokud je program syntakticky korektní, parser vytvoří abstraktní syntaktický strom.

Výsledné AST je ovšem pro naše potřeby příliš generické. Antlr je nástroj pro všeobecné použití, proto i pomocí Antleru vytvořené AST bývají všeobecné. Antlr však kromě parseru umí vygenerovat také abstraktní třídu `Visitor` sloužící k průchodu AST pomocí návrhového vzoru *visitor*. Pomocí visitoru převedeme výsledek parsování na dočasnou reprezantaci – mezivýsledek. Tento výsledek je, díky omezením automaticky vygenerované třídy, stále nedostatečný. V průběhu tohoto průchodu interpret nehledá chyby v programu – k tomu chybí překladači v současném momentě dostatečný kontext.

Proto proces parsování následuje ještě jeden průchod stromem a jeho konečný převod na přívětivou strukturu. Během tohoto průchodu překladač převede reprezentaci zdrojového kódu na objekty tříd definovaných v knihovně `common` – tedy na standardní objekty očekávané napříč projektem (překladačem, funkcemi, apod.). Dále v tomto průchodu překladač provede expanzi funkcí `Cond`, `ListOf`, `TupleOf`, `Progn` z variadických funkcí, jež v TILu neexistují, na sekvenci binárních funkcí. Během tohoto průchodu může interpret opět zachytávat chyby – většinou se jedná o chybné využití syntaktického cukru

Následně již dochází k interpretaci programu. Jednotlivé výrazy ovšem nejsou interpretovány přesně v takovém pořadí, v jakém jsou v programu uvedeny. Při překladu jsou nejprve provedeny výrazy `Import`. Pokud překladač narazí na výraz `Import`, je interpretace aktuálního souboru pozastavena a překladač přeloží importovaný soubor. Po přeložení importované závislosti jsou do kontextu současného souboru naimportovány symboly definované závislostí [2].

Dále dochází k přednostní deklaraci struktur, kterou následuje interpretace výrazů `TypeDef`. Teprve po interpretaci `TypeDef` dojde ke korektní definici struktur a jejich atributů. Využitím tohoto postupu umožníme, aby výrazy `TypeDef` mohly zmiňovat struktury, a aby v definicích struktur bylo možné zmiňovat typové aliasy.

Poté je potřeba provést deklarace proměnných a symbolických hodnot (viz 3.2.15), a současně deklarace a definice funkcí. V tomto kroku jsou deklarace proměnných automaticky vyvozeny z definic. Pokud program obsahuje definici proměnné, není nutné dodávat deklaraci. Dodání nekonfliktní deklarace ovšem není chybou.

[2]Symboly zde myslíme názvy – jména funkcí, proměnných, i symbolických hodnot. V daném kontextu slovo symbol nesouvisí se symbolickými hodnotami, využíváme jej stejně jako jej například využívají linkery kompilovaných programovacích jazyků.

Nakonec jsou interpretovány top-level konstrukce a přiřazení proměnným. Pokud je konstrukce na nejvyšší úrovni v-nevlastní, program končí chybou. Pokud však uživatel počítá s tím, že konstrukce může být v-nevlastní, může její výsledek přiřadit proměnné, nebo jej zpracovat pomocí funkce `IsNil`.

Vždy, když je vyhodnocována určitá skupina výrazů, ať už při přednostním vyhodnocování, nebo při vyhodnocování konstrukcí, pořadí výrazů v konkrétní skupině vždy odpovídá jejich relativnímu pořadí ve zdrojovém programu.

4.3.2 Detaily implementace

Tato podsekce popisuje detaily implementace, které mohou být důležité nebo zajímavé. Cílem je vyzdvihnout řešení, která jsou důležitá z hlediska budoucího rozvoje, řešení zajímavých problémů, ale také řešení, která jsou suboptimální a zasloužila by si další vývoj.

4.3.2.1 Interpretace konstrukcí, správa paměti a operace nad AST

Velkým designovým rozhodnutím, a zároveň nedostatkem, je rozhodnutí operovat přímo nad abstraktním syntaktickým stromem a interpretovat přímo TIL-Script konstrukce reprezentované Java objekty. Kód není překládán do bytekódu, který by byl následně interpretován.

Tento přístup k interpretaci má jednu velkou výhodu – nízkou náročnost na implementaci. Překlad do bytekódu by s sebou nesl značnou přidanou komplexitu. V první řadě by bylo potřeba implementovat vlastní mechanismus zásobníku. V současné době TIL-Script sdílí zásobník s Java prostředím. Aplikace funkcí je řešena rekurzivně. Při aplikaci TIL-Script funkce volá interpret interně metodu, která aplikaci provede. S voláním metody na JVM je vždy vytvořen nový rámec na zásobníku[3]. Na tomto rámci se poté uloží kontext aplikované TIL-Script funkce (argumenty funkce, apod.). Po dokončení interpretace TIL-Script funkce se interpret vrací z metody zpět na místo, odkud byla metoda zavolána. S návratem z funkce je spojena destrukce rámce na zásobníku – destrukce, která v současné implementaci obsahuje také zapomenutí kontextu funkce. Paměť využitá kontextem funkce je v případě potřeby uvolněna garbage collectorem během reklamačního cyklu.

Právě garbage collector je druhou výhodou současného přístupu. Interpretace AST a využití mechanismu zásobníku z JVM nám umožňuje využít také garbage collector z JVM pro automatickou správu paměti. Pokud bychom implementovali vlastní zásobník nezávislý na JVM, museli bychom naprogramovat také vlastní garbage collector.

Ve své podstatě však mechanismus zásobníku ani mark-and-sweep algoritmus[4] není složité naprogramovat.

[3] Tzv. *inlining funkcí* zde nebereme v potaz. Při aplikaci TIL-Script funkcí navíc musí překladač vždy zavolat virtuální metodu objektu, jenž funkci reprezentuje. Inlining virtuálních metod je možný, avšak složitější, a na podstatu řešení TIL-Script překladače nemá žádný vliv, neboť aplikace funkcí je řešena rekurzivně. Rekurzivní volání nelze inlinovat, a jelikož se nejedná o koncové volání, kompilátor nemůže aplikovat ani TCO.

[4] Alespoň tedy mark-and-sweep algoritmus ve své nejjednodušší podobě. Mark-and-sweep algoritmus se používal již v padesátých letech minulého století, a používá se dodnes, v moderních jazycích je však silně optimalizován.

Podstatně složitějším problémem by byla definice bytekódu, a také řešení obousměrného překladu mezi bytekódem a JVM objekty. Ve většině programovacích jazyků stačí implementovat překlad AST do interpretovatelného bytekódu. Dále poté překladač pouze interpretuje daný bytekód. TIL-Script ovšem umožňuje definovat funkce nejen pomocí konstrukcí Transparentní intenzionální logiky, ale také jako Java funkce. Tyto funkce psané v Javě přirozeně jako své arugmenty očekávají JVM objekty. Pokud bychom však překládali zdrojový kód do bytekódu, museli bychom před voláním Java funkce převést všechny její argumenty z bytekódu zpět na JVM objekty.

V závislosti na definici bytekódu a implementaci překladače by také mohlo být třeba implementovat vlastní haldu a alokátor paměti, neboť by nemuselo být možné využít haldu a alokátory JVM, které využívá současná implementace.

Největší nevýhodou současné implementace ovšem je chybějící optimalizace koncového volání (TCO). Optimalizací koncového volání rozumíme opakované využití jednoho rámce zásobníku pro více rekurzivních volání funkce. Při volání funkce je většinou potřeba vytvořit nový rámec na zásobníku. Moderní překladače ovšem dokáží v určitých situacích rekurzi optimalizovat, využít stejný rámec pro více po sobě jdoucích volání, a efektivně tak nahradit rekurzi cyklem. Tuto optimalizaci lze provést pouze pokud k rekurzivnímu volání dojde při opouštění funkce (proto *koncové* volání). Ačkoliv TCO implementuje také řada překladačů imperativních jazyků (např. GCC, Clang), nejčastěji se s touto optimalizací setkáváme právě ve funkcionálních jazycích, ve kterých nelze implementovat cyklus, a proto jsme odkázání čistě na rekurzi. V jazyce bez optimalizace koncové rekurze bohužel velmi rychle dojde k přetečení zásobníku.[5]

Z časových omezení se bohužel optimalizaci koncové rekurze, ani vše, co by s její implementací bylo nutně spojeno (vlastní zásobník, správu paměti, překladač a interpret bytekódu), nepodařilo implementovat. Čas byl věnován především rozvoji jazyka TIL-Script a možností jeho interpretace. Implementaci TCO je ovšem třeba do budoucna zvážit, a tato sekce tak může posloužit alespoň jako návod, jak pokračovat v rozvoji jazyka TIL-Script.

4.3.2.2 Java objekty jako TIL-Script funkce

TIL-Script nabízí uživatelům dva způsoby definice funkce. Prvním způsobem je abstrakce nad konstrukcí, ať už pomocí uzávěru, nebo pomocí syntaxe pro definici pojmenované funkce. Druhým způsobem je již dříve zmiňovaná definice funkce pomocí Java objektu.

Aby byl JVM objekt použitelný jako funkce jazyka TIL-Script, musí být daný objekt instancí třídy `DefaultFunction`. Daný objekt poté musí definovat jméno a typ funkce, a současně metodu `apply` volanou právě při aplikaci dané funkce. Argumenty této metody jsou instance rozhraní `InterpreterInterface`, představující aktuálně používaný interpret, seznam (`java.util.List`) všech argumentů funkce, a také kontext aplikace funkce. Kontext aplikace obsahuje pozici

[5] Zásobník JVM lze zvětšit pomocí CLI argumentu `-Xss`. Zvětšením zásobníku problém s jeho přetékáním můžeme částečně omezit.

ve zdrojovém kódu, kde k aplikaci funkce došlo, a slouží především k hlášení chyb. Návratovou hodnotou metody `apply` je výsledek aplikace funkce na dodané argumenty.

Metoda `apply` může vrátit `Nil`, nikdy však `Nil` nemůže obdržet jako jeden ze svých argumentů – pokud by alespoň jeden argument měl hodnotu `Nil`, k aplikaci funkce, a tedy ani volání metody `apply`, nikdy nedojde.

Definice TIL-Script funkcí pomocí Java objektů umožňuje využívat v jazyce TIL-Script již existující Java knihovny. Například místo implementace volání funkcí systému (tzv. *syscall*), a následné implementace komunikace po síti přímo nad operačním systémem v jazyce TIL-Script lze využít již existující knihovny napsané v jazyce Java. Pro komunikaci s databází by tedy stačilo využít již napsanou knihovnu pro platformu JVM, a pouze nad ní vytvořit jednoduché TIL-Script rozhraní.

Takto reprezentovanou funkci poté stačí importovat v TIL-Script programu za využití registrátoru. Registrátory jsou popsány hned v následující podsekci 4.3.2.3 – Import symbolů z Java archivu.

4.3.2.3 Import symbolů z Java archivu

Výhody umožnění implementace TIL-Script funkcí pomocí Java objektů již byly popsány. Výrazy `Import` byly popsány v sekci 3.3.4 – Výrazy *Import*. Nyní se budeme věnovat způsobu, jakým jsou symboly importovány z Java archivů.

Abychom vůbec mohli importovat definice z Java archivů, musí být daný JAR soubor načtený již při spouštění JRE. Java runtime ve snaze zabránit kyberútokům (například RCE útokům) značně ztěžuje načítání kódu za běhu. Způsoby neomezeného načítání kódu existují, liší se však napříč verzemi platformy Java, ale hlavně představuje potenciální velký bezpečnostní problém.

Abychom mohli bezpečně načíst kód za běhu, museli bychom znát nejen název třídy, jež potřebujeme načíst, ale také cestu k JAR souboru, ve kterém se třída nachází. Pokud bychom po uživateli jazyka TIL-Script chtěli, ať uvádí i Java archiv i plně kvalifikovaný název registrátoru, byly by `Import` výrazy příliš složité. Uvést pouze Java archiv nestačí, neboť nemusíme znát obsah daného archivu, případně alespoň názvy registrátorů. Abychom mohli uvést pouze registrátor, museli bychom znát JAR soubor dopředu. Poslední verze se zdá pro konečného uživatele jazyka TIL-Script nejjednodušší. Uživatel může JAR soubory uvést ručně, případně lze například automaticky načítat všechny skripty z adresáře na disku pomocí shell skriptu.

Zvolený způsob importu Java objektů dále obsahuje ještě jednu výhodu – umožňuje definovat více registrátorů v jednom JAR souboru, ale podle potřeby načítat a využívat pouze konkrétní registrátor. Není tedy třeba importovat více názvů, než je nezbytně nutné.

Vytvoření instance registrátoru a import symbolů jsou implementovány následovně. Překladač interpretuje výrazy `Import` přednostně hned při spuštění programu. Narazí-li na výraz `Import`, nejprve zkontroluje, zda parametr výrazu obsahuje předponu `class://`. Pokud ne, překladač automaticky považuje daný parametr za cestu k TIL-Script souboru (nezávisle na příponě, konvencí však je

používat příponu *.tils*). Pokud parametr začíná požadovanou přeponou, překladač přeponu odstraní a pokusí se načíst specifikovanou třídu. Následně vytvoří instanci této třídy pomocí konstruktoru bez parametrů (všechny registrátory tedy musí tento konstruktor definovat). Dále již překladač s vytvořenou instancí pracuje jako s objektem typu `SymbolRegistrar`. Všechny registrátory proto musí toto rozhraní implementovat.

Rozhraní `SymbolRegistrar` definuje šest metod – *getterů* vracejících Java seznamy funkcí (jeden getter využíváme pro deklarace, druhý getter pro definice – funkce s implementací), typových aliasů, definic symbolických hodnot, struktur, a proměnných.[6] Třídy implementující rozhraní `SymbolRegistrar` musí implementovat všech šest metod, jednotlivé metody ale můžou vracet prázdné seznamy.

Překladač při importu pomocí registrátoru zavolá všech šest metod definovaných rozhraním `SymbolRegistrar`, a v nespecifikovaném pořadí importuje všechna jména definovaná registrátorem. Pořadí je nespecifikované, neboť na něm nezáleží při importu pomocí registrátoru.

Po dokončení importu si překladač uloží plně kvalifikovaný název registrátoru do kontextu skriptu, ve kterém byl registrátor využit, aby se import symbolů pomocí jednoho registrátoru neprovedl zbytečně vícekrát.

4.3.2.4 Typová kontrola

Transparentní intenzionální logika je typovaným kalkulem, proto i v jazyce TIL-Script musíme zajistit typovou koherenci za využití typové kontroly. Typová kontrola probíhá pouze během provedení konstrukce, neprobíhá tedy například hned při spuštění skriptu ještě před provedením první konstrukce, nebo nad konstrukcemi, které nejsou nikdy provedeny. Narazí-li překladač během typové kontroly na typově inkoherentní výraz (např. konstrukci, přiřazení proměnné), program okamžitě končí chybou.

V Transparentní intenzionální logice je provedení typové kontroly problematičtější. Zejména pokud je jedním z konstituentů konstrukce dvojí provedení, není vždy možné provést typovou kontrolu strojově. Dále, díky hyperintenzionalitě, která nám umožňuje vytvářet nové konstrukce dynamicky za běhu programu (například pomocí substituční metody), vyvstává otázka, kdy typovou kontrolu provádět. Nabízí se možnost typovou kontrolu provést vždy, když je zkonstruována konstrukce, nezávisle na tom, zda bude konstrukce provedena. Tento problém s sebou nese výkonnostní implikace, neboť by bylo nutné provádět typovou kontrolu velmi často. Navíc by ji nebylo vždy možné provést. Proto typovou kontrolu překladač provádí pouze tehdy, kdy ji může provést s naprostou přesností – tedy při provedení konstrukce, kdy již pracuje s konkrétními objekty známého typu. Navíc tak překladač provede typovou kontrolu pouze jednou. Výhodou tohoto přístupu je vyšší výkon – překladač provádí méně operací, nevýhodou pak je, že překladač nedokáže chyby zachytit předem.

Ukázku hlášení chyby lze vidět například v ukázce 4.1 – Příklad hlášení chyby.

[6] V Kotlinu můžeme používat syntaktický cukr `get()` pro Java gettery.

```
  ** Error **
(2, 2): ['CatS '1 '"text"].
     ~~~ ^ ~~~
        Invalid argument type in application of function 'CatS' (expected: Text,
           received: Int)
```

Ukázka 4.1: Příklad hlášení chyby

Typová kontrola probíhá během provedení konstruktoru struktury, přiřazení hodnoty proměnné, při deklaraci funkce, symbolické hodnoty i proměnné, při definici typového aliasu, a také před i po aplikaci funkce.

Typová kontrola při provádění konstruktoru struktury zajistí, že jsou atributům struktury přiřazeny objekty správného typu. Obdobně pomocí typové kontroly zajistíme, že je proměnné přiřazena hodnota korektního typu. Při interpretaci definicí typových aliasů a struktur, ale také při interpretaci deklarací, potřebujeme zajistit, aby nebylo možné poskytnout konfliktní deklarace nebo definice – tedy deklarace, případně definice, které se shodují názvem, ale liší typem. Před aplikací funkce překladač provádí typovou kontrolu proto, aby zajistil, že funkce obdrží argumenty správného typu. Po aplikaci funkce provede překladač typovou kontrolu znova, a to za účelem zajištění, že funkce vrátila objekt správného typu.

Typová kontrola při přiřazení hodnoty proměnné, případně při detekci konfliktních deklarací nebo definicí typových aliasů, vyžaduje pouze porovnání dvou konkrétních negenerických typů.

Kontrola konfliktních definic struktur obsahuje kontrolu na úrovni atributů. Při porovnávání atributů jsou porovnávány názvy i typy i-tého atributu obou z porovnávaných struktur. Pokud se atribut liší v názvu nebo v typu, jsou struktury automaticky považovány za rozdílné typy. K porovnání atributů vůbec nedojde, pokud se počet atributů liší, nebo pokud se struktury liší v názvu. Z popisu typové kontroly struktur vyplývá, že záleží na pořadí atributů – ačkoliv by to ve vysokoúrovňovém skriptovacím jazyku nemuselo být třeba (např. protože neřešíme padding), cílem této volby je přinutit uživatele nevytvářet zbytečné definice struktur, zamezit případům, kdy se špatně navrhnutá struktura využívá k účelům, ke kterým nebyla zamýšlena, a také zjednodušení implementace.

Při typové kontrole argumentů a výsledků funkcí již není porovnávání jednotlivých typů nezávislé, neboť funkce můžou být typově polymorfní. Typová kontrola probíhá nejprve na argumentech funkce. Při porovnávání typů argumentů je postupováno po jednotlivých argumentech. Pokud funkce pro daný argument očekává objekt konkrétního typu, je pouze porovnán typ argumentu s požadovaným typem. Pokud však funkce pro daný argument neočekává konkrétní typ, můžou nastat dva případy. První případ nastane, narazíme-li na generický typ α_i[7] s indexem i poprvé v kontextu apli-

[7] V kontextu jazyka TIL-Script zde myslíme typ `Any`. TIL konvence je zde využita z důvodu úspornosti a jednodušší indexace

kace funkce. V takovém případě si překladač uloží k indexu i typ dodaného argumentu, k žádnému porovnání typů však nedochází – nemáme co porovnat. Druhý případ poté představuje situaci, kdy narazíme na generický typ α_i, ale generickému typu α_i jsme již přiřadili konkrétní typ v kontextu současné aplikace. Tehdy již dojde k porovnání očekávaného typu s typem dodaným. Po úspěšné aplikaci funkce je ještě zkontrolován typ objektu, jež funkce vrátila, aby překladač zajistil, že funkce vrátila objekt správného typu.

Při porovnávání jednotlivých typů jsou nejprve generické typy přepsány na typy konkrétní a typové aliasy jsou rekurzivně expandovány, dokud nezískáme původní typ, na který se alias odkazuje. Atomické typy následně stačí porovnat podle názvu. Molekulární typy (v TIL funkce, v jazyce TILScript navíc seznamy, n-tice a struktury), jsou porovnávány podle typů, z nichž se skládají. Struktury jsou navíc porovnávány podle názvu struktury i jejich atributů (viz dřívější odstavec této sekce). Pokud se dva molekulární typy skládají z různého počtu typů, k porovnávání jednotlivých typů samozřejmě nedochází.

4.3.2.5 Mechanismus zásobníku

Mechanismus sdíleného zásobníku (*callstack*) byl nastíněn v oddílu 4.3.2.1 – Interpretace konstrukcí, správa paměti a operace nad AST, zde se mu však budeme věnovat o trochu podrobněji. Pochopení tohoto mechanismu je navíc potřeba k pochopení implementace uzávěrů a uzávěrem konstruovaných funkcí.

Kromě JVM zásobníku si interpret udržuje také vlastní zásobník. Vlastní zásobník je však díky rekurzivní implementace aplikace funkcí svázán se zásobníkem platformy Java. Jednotlivé rámce zásobníku jsou využívány k ukládání proměnných. Dále pak má každý rámec nanejvýš jeden rodičovský rámec. Při provádění proměnné překladač zkontroluje nejprve rámec na vrcholu zásobníku. Pokud daný rámec neobsahuje požadovanou proměnnou, postupuje překladač rekurzivně a prohledává rodičovský rámec. Pokud rámec žádného rodiče nemá a proměnná nebyla nalezena, program končí chybou.

První rámec je vytvořen při spuštění skriptu. Na tomto rámci jsou ukládány proměnné definované na nejvyšší úrovni programu – proměnné, které budeme využívat jako volné proměnné v konstrukcích. Tento rámec nemá žádného rodiče.

Nový rámec zásobníku je vytvořen vždy při aplikaci funkce. Na tento rámec jsou umístěny všechny λ-vázané proměnné – argumenty funkce. Rodičem rámce pojmenované funkce je vždy rámec na začátku zásobníku, tedy rámec reprezentující globální stav.

Pokud však funkci konstruujeme pomocí uzávěru, situace se komplikuje. Takové funkce mají přístup nejen k vlastním argumentům a proměnným definovaným na nejvyšší úrovni v programu, ale také k λ-vázaným argumentům funkce, v níž byl uzávěr proveden a funkce zkonstruována. V takovém případě je k funkci konstruované uzávěrem uložen také rámec, na kterém byla daná funkce vytvořena. Tento rámec poté při aplikaci funkce vytvořené uzávěrem poslouží právě jako rodičovský

rámec. Nejnižší rámec v zásobníku je nastaven jako rodičovský pouze pokud uzávěr nebyl proveden ve funkci, ale na nejvyšší úrovni v programu.

Pro znázornění si představme následující konstrukci. Konstrukce konstruuje funkci následníka.

```
[[\x: Int -> (Int Int) [\y: Int -> Int ['+ x y]]] '1]
```

Ukázka 4.2: Funkce následníka pomocí vnořených uzávěrů

Funkci následníka získáme aplikací Schönfinkelovy redukce (*currying*) na funkci sčítání. Operace sčítání je na celých číslech totální funkcí, proto tuto redukci můžeme aplikovat.

```
[\x: Int -> (Int Int) [\y: Int -> Int ['+ x y]]]
```

Ukázka 4.3: Vnější uzávěr

Volná proměnná `x` je konstituentem vnitřního uzávěru. Zde chceme, aby proměnná `x` konstruovala hodnotu `1/Int`.

```
[\y: Int -> Int ['+ x y]]
```

Ukázka 4.4: Vnitřní uzávěr

Během konstruování funkce následníka nejprve provedeme vnější uzávěr. Tento uzávěr zkonstruuje unární funkci, jejímž oborem hodnot jsou unární funkce typu `(Int Int)`. Aplikací této funkce na argument `1` získáme funkci následníka. Aby volná proměnná `x` korektně konstruovala hodnotu `1`, musíme funkci následníka jako rodičovský rámec zásobníku nastavit právě rámec, na kterém byla daná funkce vytvořena, neboť právě tento rámec ukládá proměnnou `x` a její hodnotu.

Po aplikaci funkce je rámec odstraněn ze zásobníku. Pokud na rámec neodkazuje již žádný ukazatel, může být rámec smazán garbage collectorem. Pokud však na konkrétní rámec odkazuje funkce, pro niž je tento rámec rodičovský, rámec bude udržován v paměti společně s danou funkcí, dokud to bude potřeba.

4.3.2.6 Nezávislost knihoven na překladači

Projekt je koncipován tak, aby byly TIL-Script knihovny nezávislé na intepreteru, který uživatel použije. Implementovaný překladač je plně funkční, díky časovým omezením ovšem překladači chybí některé nutné prvky, jako třeba optimalizace koncového volání, jež je ve funkcionálních jazycích nezbytná, či překlad do bytekódu umožňující efektivnější překlad. V budoucnu se může stát, že bude potřeba nahradit současný překladač jinou implementací. V takovém případě ovšem není žádoucí, aby nastala potřeba přepsat nebo upravit také všechny již existující TIL-Script knihovny, standardní knihovnu, apod. Pro implementaci TIL-Script knihovny je však potřeba, aby např. TIL-Script funkce implementované v Javě měly určitý přístup k překladači, nebo alespoň kontextu, ve kterém jsou volány. Jinak by tyto funkce nemohly přistupovat k proměnným, k jiným funkcím, apod.

Za účelem odstranění závislosti na konkrétním překladači definuje knihovna `common` rozhraní `InterpreterInterface`. Toto rozhraní specifikuje základní funkce, které překladač jazyka TIL-Script musí implementovat. Při volání TIL-Script funkcí definovaných pomocí Java objektů je danému objektu předána instance překladače pomocí rozhraní `InterpreterInterface`.

Tento princip je v softwarovém inženýrství velmi známý, je však nutné, aby na modularitu systému bylo myšleno již od počátku a aby byl dobře zdokumentován. Tato práce umožňuje, aby bylo v budoucnu možno vytvořit drop-in náhradu za současný překladač bez jakékoliv úpravy existujícího kódu.

Kapitola 5

Uživatelská dokumentace

Tato kapitola je věnována uživatelské dokumentaci. Uživatelská dokumentace začíná ukázkou několika TIL-Script programů. Poté následuje návod, jak spustit překladač jazyka TIL-Script a program přeložit. Dále dokumentace popisuje standardní i matematickou knihovnu. Součástí dokumentace je například popis funkcí standardní i matematické knihovny. Uživatel tak může při práci s jazykem TIL-Script konzultovat tuto dokumentaci a vyhledávat si potřebné informace. Na konci dokumentace se nachází také ukázka implementace TIL-Script knihovny v jazyce Java, příloha práce však obsahuje také ukázku knihovny psané v jazyce Kotlin.

5.1 Ukázka TIL-Script programu

5.1.1 Hello, World!

V rámci zachování programátorské tradice začneme ukázkou implementace programu *Hello, World!*.

Náš program nejprve začíná komentáři. Zápis komentářů je stejný jako v jazyce Haskell. Obsah komentářů je překladačem ignorován. Komentáře následuje velmi jednoduchá Kompozice. Kompozicí aplikujeme funkci `Println` na argument – objekt, který chceme vypsat. Argument `Println` může být libovolného typu, zde vypisujeme textový řetězec. Funkci `Println` a literál textového řetězce nesmíme zapomenout trivializovat.

```
-- Toto je komentář
-- Následující řádek vypíše Hello, World!
['Println '"Hello, World!"].
```

Ukázka 5.1: Program Hello, World! v jazyce TIL-Script

5.1.2 Jednoduchá aritmetika

Nyní si ukážeme složitější Kompozice, ukázku matematických funkcí, a také přiřazení hodnoty proměnné, abychom nemuseli stejnou kompozici zbytečně provádět vícekrát. Abychom mohli využívat matematické funkce, nesmíme zapomenout importovat matematickou knihovnu.

```
-- Import matematické knihovny
Import "class://org.fpeterek.tilscript.math.Registrar".

-- Aplikace funkce
-- Nesmíme zapomenout, že definičním oborem logaritmu
-- jsou reálná čísla, proto musíme zapsat literál reálného čísla
-- Literály reálných čísel obsahují desetinnou tečku a alepoň jednu
-- číslici před i za desetinnou tečkou
['Log '27.0 '3.0].

-- Výsledek kompozice využijeme jako argument jiné funkce
['Log ['+ '20.0 '7.0] '3.0].

-- Předchozí výsledky jsme však zbytečně zahodili
-- Abychom mohli s výsledkem dále pracovat, uložíme jej do proměnné
-- Alternativně můžeme výsledek vypsat na výstup programu

x -> Real := ['Log ['+ '20.0 '7.0] '3.0].

-- Čísla můžeme sčítat, odčítat, násobit, dělit i porovnávat

-- Vypíše True
['Println ['= ['+ '15 '5] ['- '24 '4]]].

-- Vypíše False
['Println ['= ['* '2 '3] ['/ '42 '6]]].

-- Proměnnou můžeme provést
['Println x].

-- A využít jako argument funkce
['Println ['Cos x]].
```

```
-- Na výstup programu můžeme vypsat také konstrukci
-- Zde aplikujeme funkci Print na konstrukci, abychom ji vypsali
-- Využíváme Print, ne Println, abychom nevypsali sekvenci odřádkování
['Print '['Cos x]].

-- Oddělíme konstrukci od výsledku jejího provedení
['Print ': "].

-- A nakonec konstrukci provedeme a výsledek vypíšeme
['Println ['Cos x]].
```

Ukázka 5.2: Aritmetika v jazyce TIL-Script

5.1.3 Abstrakce a tvorba funkcí

Abstrakce je základním pravidlem jak lambda kalkulu, tak programování. Nyní si abstrakce v jazyce TIL-Script ukážeme prakticky.

Funkci lze vytvořit provedením uzávěru, nebo pomocí speciální syntaxe pro definici pojmenované funkce. Uzávěr je konstrukcí, proto jej můžeme využít jako podkonstrukci jiné konstrukce. Definice pojmenované funkce konstrukcí není. Definicí pojmenované funkce přiřadíme funkci jméno. Kdykoliv pak potřebujeme funkci zmínit, využijeme k tomu její jméno. Pojmenovanou funkci nesmíme zapomenout trivializovat.

V ukázce si také zkusíme vytvořit trochu zajímavější funkci. V předchozí ukázce jsme chtěli vypsat konstrukci i její výsledek. V nynější ukázce si nad tímto výpisem vytvoříme abstrakci. V ukázce využijeme funkci `Progn`. Funkce je podrobněji popsána v dokumentaci funkcí standardní knihovny, viz 5.3.1.2. Nyní pouze uvedeme, že se jedná o syntaktický cukr nad kombinátorem $\lambda xy.y$. Je-li alespoň jeden z argumentů `Progn` hodnota `Nil`, pak je nevlastní celá kompozice, jinak funkce `Progn` vrací svůj druhý argument. Pokud obdrží argumentů více, parser kompozici rozepíše na vnořené aplikace binární funkce `Progn`.

```
-- Funkci lze zkonstruovat provedením uzávěru
[\x: Real, y: Real -> Real ['Sqrt ['+ ['* x x] ['* y y]]]].

-- Takto zkonstruovanou funkci lze aplikovat
-- Pro přehlednost lze uzávěr a argumenty uvést na více řádcích
[[\x: Real, y: Real -> Real ['Sqrt ['+ ['* x x] ['* y y]]]]
    '3.0
    '4.0].
```

```
-- Pro zpřehlednění můžeme funkci přiřadit proměnné
hypotenuse -> (Real Real Real) :=
    [\x: Real, y: Real -> Real
        ['Sqrt ['+ ['* x x] ['* y y]]]].

-- Jelikož je hypotenuse proměnná, netrivializujeme ji
['Println [hypotenuse '3.0 '4.0]].

-- Nyní si vytvoříme funkci pro výpis konstrukce a výsledku jejího provedení
-- Funkci Progn využijeme, abychom provedli více výpisů na výstup programu
-- Proměnná cons v-konstruuje konstrukci nejprve vypíše danou konstrukci
-- Za konstrukci poté dopíšeme dvojtečku
-- Nakonec musíme využít dvojí provedení, abychom provedli konstrukci
-- konstruovanou proměnnou cons
Defn PrettyPrint(cons: Construction) -> Any1 :=
    ['Progn
        ['Print cons]
        ['Print ': "]
        ['Println ^2 cons]].

-- Nakonec stačí funkci aplikovat na konstrukci
['PrettyPrint '[hypotenuse '3.0 '4.0]].
```

Ukázka 5.3: Funkce a uzávěry

V praxi se klidně může stát, že bude funkce `PrettyPrint` aplikována na funkci, která je v-nevlastní pro aktuálně zkoumanou valuaci v. Aplikací funkce `Println` na hodnotu `Nil` dojde k výpisu `Nil` na výstup programu, přesto bude celá Kompozice v-nevlastní. Toto chování je v praxi pravděpodobně žádoucí, chtěli bychom vidět, že naše Kompozice byla v-nevlastní a nezkonstruovala žádnou hodnotu. Jako cvičení si však můžeme napsat také variaci funkce `PrettyPrint`, jež provede zápis na výstup programu pouze pokud obdržená konstrukce není v-nevlastní.

Za tímto účelem musíme vytvořit novou abstrakci nad aplikací funkce `Progn`. Abstrakci vytvoříme uzávěrem. Uzávěr bude konstruovat funkci, která přijímá jeden argument – výsledek provedení konstrukce, jež je argumentem `PrettyPrint`. Pokud bude tato konstrukce v-nevlastní, bude současně v-nevlastní také aplikace naší abstrakce nad `Progn`.

```
Defn PrettyPrint(cons: Construction) -> Any1 :=
    -- Abstrakce nad aplikací Progn
    [[\x: Any1 -> Any1
        ['Progn
```

```
['Print cons]
['Print ': "]
['Println x]]]
-- Argument funkce konstruované uzávěrem
^2 cons].
```

Ukázka 5.4: Funkce a uzávěry

5.1.4 Seznamy

Další praktický příklad se týká práce se seznamy. Předvedeme, jak idiomaticky pracovat se seznamy (tvorba, průchod, apod.). Z příkladu poté snad vyplynou také výhody induktivní implementace seznamů, jež se využívá ve funkcionálních programovacích jazycích.

Nejprve začneme tvorbou jednoduchého seznamu. Seznam vytvoříme aplikací funkce `ListOf`. Funkce `ListOf` slouží jako syntaktický cukr, který nám zjednoduší tvorbu seznamu.

Následně si zkusíme seznam projít a odstranit z něj prvky větší nebo rovné pěti. Začneme proto tvorbou predikátu, funkce `LessThanFive`. Dále vytvoříme rekurzivní funkci `GetSmall`, která vytvoří nový seznam obsahující pouze takové prvky, které jsou menší než pět. Funkce jako svůj argument obdrží seznam. Zde můžeme vidět využití funkce `If`.

Je-li seznam prázdný, vrátíme jej v nezměněné podobě – z prázdného seznamu již nemáme co odstraňovat. Argumenty funkce `If` jsou vyhodnocovány líně – je vždy vyhodnocena pouze jedna podkonstrukce. Pokud by došlo k vyhodnocení obou konstrukcí, došlo by k pokusu k přístupu prvního prvku seznamu – prázdný seznam ovšem nemá první prvek. Celá aplikace `GetSmall` by tak byla v-nevlastní pro všechny argumenty a ve všech valuacích (jednalo by se o funkci *degenerovanou*).

Pokud je seznam neprázdný, potřebujeme provést další aplikaci funkce `If`. Nejprve zkontrolujeme, jestli je první prvek seznamu menší než pět. Nezávisle na hodnotě aktuálního prvku – pomocí funkce `Tail` vezmeme seznam bez jeho prvního prvku, a rekurzivně aplikujeme `GetSmall`. Rekurzivní aplikací získáme zbytek seznamu, z něhož již byly odstraněny čísla větší než čtyři. Jediné, v čem se liší jednotlivé větve podmínky (argumenty funkce `If`), je, zda zachováme první prvek seznamu, nebo ne. Pokud je prvek seznamu menší než pět, pomocí funkce `Cons` vytvoříme nový seznam přidáním prvního prvku původního seznamu na začátek seznamu vytvořeného rekurzivní aplikací `GetSmall`. Pokud je hodnota prvku pět nebo vyšší, vrátíme pouze výsledek rekurzivní aplikace `GetSmall` na zbytek seznamu a první prvek ignorujeme.

Jelikož jsou seznamy neměnné (chceme-li seznam jakkoliv změnit, naší jedinou možností je vytvořit nový seznam), a protože jsou definovány induktivě (každý seznam je buď prázdný seznam, nebo buňka, *cons cell*, skládající se z prvního prvku seznamu, a podseznamu reprezentující zbytek seznamu), jsou operace `Cons`, `Head`, `Tail` (přesněji tedy operace, které aplikace těchto funkcí představuje), proveditelné v konstantním čase.

Dále budeme chtít projít seznam, a vybrat jen ty prvky, které jsou větší než čtyři. Ihned se nabízí možnost zkopírovat funkci `GetSmall` a jen ji upravit, aby využila jiný predikát. Snad není třeba uvádět, že tento postup je neideální, neboť v lambda kalkulu, a tedy i Transparentní intenzionální logice, lze využívat funkce jako argumenty jiných funkcí. Proto zkopírujeme funkci `GetSmall`, novou funkci pojmenujeme `Filter`, a přidáme nový argument, predikát, funkci typu `(Bool Int)`. Aplikace funkce `GetSmall` nahradíme rekurzivní aplikací `Filter`, a namísto aplikace `LessThanFive` aplikujeme funkci zkonstruovanou provedením argumentu `pred`.

Funkci `Filter` můžeme jako predikát předat funkci `LessThanFive`, a ověřit, že aplikace funkce `Filter` na seznam a predikát `LessThanFive` je ekvivalentní aplikaci `GetSmall` na ekvivalentní seznam.

Funkci `GetBig` poté definujeme pouze jako aplikaci funkce `Filter` na seznam a predikát `\x: Int -> Bool ['> x '4]`.

Dále se pokusíme napsat funkci, která transformuje jednotlivé prvky seznamu, ale zachová počet prvků. Takovou funkci tradičně ve funkcionálním programování nazýváme `Map`, proto se tohoto názvosloví budeme nyní držet. Funkce `Map` přijímá dva argumenty – seznam, a funkci, pomocí které transformujeme jeden prvek seznamu. Nyní nám postačí jen jediná podmínka. Pokud je seznam prázdný, vrátíme jej, jak jsme jej obdrželi – není co transformovat. Pokud je seznam neprázný, aplikujeme funkci reprezentovanou argumentem `transform` na první prvek seznamu, rekurzivně transformujeme zbytek seznamu, a nakonec vytvoříme nový seznam z transformovaného zbytku a prvního prvku seznamu pomocí funkce `Cons`. Poté stačí ověřit funkčnost funkce `Map` například tím, že si spočítáme druhé mocniny čísel v našem seznamu.

Poslední příklad jen ukazuje, že nemusíme definovat pouze seznamy čísel. Lze definovat také seznamy individuí, funkcí, konstrukcí, apod. Seznamy jsou ovšem homogenní, nemůžou tedy obsahovat objekty různého typu.

```
-- Nejprve si vytvoříme seznam, se kterým budeme pracovat
numbers -> List(Int) := ['ListOf '1 '6 '2 '5 '3 '4].

-- Pomocná funkce -- predikát, který později využijeme
Defn LessThanFive(num: Int) -> Bool := ['< num '5].

-- Funkce GetSmall vrátí nový seznam obsahující pouze čísla
-- menší než pět
Defn GetSmall(list: List(Int)) -> List(Int) :=
    ['If ['IsEmpty list]
        -- Pokud je seznam prázdný, můžeme jej přímo vrátit
        list
        -- Jinak zkontrolujeme, zda je první prvek seznamu
```

```
        -- menší než pět, a pokud ano, uložíme jej na začátek
        -- seznamu získaného rekurzivní aplikací GetSmall
        ['If ['LessThanFive ['Head list]]
            ['Cons ['Head list] ['GetSmall ['Tail list]]]
            ['GetSmall ['Tail list]]]].

['Println ['GetSmall numbers]].

-- Predikát lze parametrizovat
-- Funkce Filter přijímá jako jeden z argumentů predikát,
-- který je využit namísto konkrétní funkce
Defn Filter(list: List(Int), pred: (Bool Int)) -> List(Int) :=
    ['If ['IsEmpty list]
        list
        ['If [pred ['Head list]]
            ['Cons ['Head list] ['Filter ['Tail list] pred]]
            ['Filter ['Tail list] pred]]].

['Println ['Filter numbers 'LessThanFive]].

Defn GetBig(list: List(Int)) -> List(Int) :=
    ['Filter list [\x: Int -> Bool ['> x '4]]].

['Println ['GetBig numbers]].

-- Pomocí funkce Map můžeme transformovat jednotlivé prvky seznamu
Defn Map(list: List(Int), transform: (Int Int)) -> List(Int) :=
    ['If ['IsEmpty list]
        list
        ['Cons
            [transform ['Head list]]
            ['Map ['Tail list] transform]]].

['Println ['Map numbers [\x: Int -> Int ['* x x]]]].

Karel, Petr, Adela/Indiv.

-- Seznamy jsou vždy homogenní, můžeme však mít seznamy
```

```
-- objektů libovolného typu, zde vytváříme seznam individuí
['Println ['ListOf 'Karel 'Petr 'Adela]].
```

Ukázka 5.5: Funkce a uzávěry

5.1.5 N-tice

V další ukázce si vyzkoušíme práci s n-ticemi. N-tice jsou heterogenní kolekce přesně specifikované délky. Délka n-tice je dána jejím typem.

V ukázce bude opět nutné importovat matematickou knihovnu.

N-tici můžeme vytvořit aplikací funkce `MkTuple`. Funkce `MkTuple` je podrobněji popsáná v dokumentaci standardní knihovny, viz 5.3.1.2, zde pouze zmíníme, že se jedná o syntaktický cukr podobný například funkci `ListOf`, který nám umožňuje jednoduše vytvářet n-tice.

Dále definujeme funkce `Fst` a `Snd`. Funkce jsou typově polymorfní a jejich argument je typu `Any1`. Argument musí být typu `Any1`, aby bylo možné funkci aplikovat na n-tice různé délky. Pokud by oborem hodnot funkce byly pouze n-tice přesně dané délky n, mohli bychom jako typ argumentu explicitně uvést n-tici, a typ `Any` využít pouze pro prvky n-tice. Tedy například pro trojice bychom využili typ `Tuple(Any1, Any2, Any3)`.

Funkce `Fst` vrátí první prvek n-tice. Funkce `Snd` vrátí druhý prvek n-tice. Funkce `Get` je obsažena ve standardní knihovně jazyka TIL-Script a vrátí prvek n-tice na pozici i. Prvky n-tice jsou indexovány od nuly.

Pomocí funkce `PrependToTuple` můžeme vytvořit novou $(n+1)$-tici přidáním prvku na začátek již existující n-tice.

Nakonec si ukážeme možné reálné využití n-tic. Nejprve, pro zpřehlednění zápisu, pomocí výrazu `TypeDef` vytvoříme nové jméno, `Vector2`, pro typ `Tuple(Real, Real)`. Tímto typem, jak název napovídá, budeme reprezentovat dvoudimenzionální vektory. Dále definujeme funkci `DotProduct`, která spočítá skalární součin páru dvoudimenzionálních vektorů.

Poslední řádky ukázky ukazují funkcionalitu výrazu `TypeDef`. Tyto výrazy nevytvoří nový typ, pouze přiřadí jiné jméno již existujícímu typu. V našem programu vytvoříme dvě n-tice typu `Tuple (Real, Real)`, a dále jen aplikujeme funkci `DotProduct` na naše vektory.

```
Import "class://org.fpeterek.tilscript.math.Registrar".

-- Tvorba n-tice libovolné délky -- zde vytváříme trojici
triple -> Tuple(Int, Text, Construction) :=
    ['MkTuple ['+ '1 '3] ['IntToText '5] '['+ '3 '3]].

['Println triple].
```

```
-- Definice funkce přijímající n-tici libovolné délky
Defn Fst(tuple: Any1) -> Any2 :=
    ['Get tuple '0].

Defn Snd(tuple: Any1) -> Any2 :=
    ['Get tuple '1].

['Println ['Fst triple]].
['Println ['Snd triple]].

-- Vytvoření nové n-tice přidáním prvku na začátek již existující n-tice
quadruple -> Tuple(Real, Int, Text, Construction) :=
    ['PrependToTuple '3.0 triple].

['Println quadruple].

-- Vytvoření alternativního jména pro typ Tuple(Real, Real)
TypeDef Vector2 := Tuple(Real, Real).

-- Vytvoření funkce operující nad dvěma n-ticemi konkrétního typu
Defn DotProduct(v1: Vector2, v2: Vector2) -> Real :=
    ['+
        ['* ['Fst v1] ['Fst v2]]
        ['* ['Snd v1] ['Snd v2]]].

v1 -> Vector2 := ['MkTuple '2.0 '3.0].
v2 -> Vector2 := ['MkTuple '3.0 '2.0].

['Println ['DotProduct v1 v2]].
```

Ukázka 5.6: Práce s n-ticemi

5.1.6 Struktury

Struktury, viz 3.3.8 – Struktury, slouží k vytvoření nového typu. Struktury jsou složené typy, které nám umožňují rozlišit typy, které mají stejnou interní reprezentaci (tedy skládají se ze stejného množství objektů stejného typu uspořádaných ve stejném pořadí).

V ukázce opět importujeme matematickou knihovnu a definujeme funkci `PrettyPrint`.

Následně definujeme dva nové typy, `Polar`, sloužící k reprezentaci polárních souřadnic, a `Cart`, reprezentující souřadnice kartézské. Na ukázce dobře vidíme, že oba typy mají stejnou interní reprezentaci, tedy oba se skládají ze dvou objektů typu `Real`. Pokud bychom k reprezentaci kartézských i polárních souřadnic využili n-tice, oba druhy souřadnic by byly reprezentovány typem `Tuple(Real, Real)`. Poté by mohlo jednoduše dojít k záměně jednoho druhu souřadnic za druhý. Typový systém by této záměně nedokázal zabránit. Vytvoření nového typu nám umožňuje využít typovou kontrolu k zabránění chybné záměny souřadnic.

Abychom si předvedli práci se strukturami, vytvoříme si funkci pro převod kartézských souřadnic na polární. Za tímto účelem si ovšem nejprve definujeme funkci `GetAngle`. Argumenty této funkce jsou souřadnice `x`, `y` bodu a vzdálenost tohoto bodu od počátku v kartézském systému. V reálné implementaci by si funkce vzdálenost od počátku měla spočítat sama, jelikož se však jedná pouze o pomocnou funkci, vzdálenost předáme jako argument, abychom ji nemuseli počítat vícekrát. Funkce `GetAngle` spočítá úhel mezi osou x a úsečkou definovanou počátkem a bodem.

Funkce `CartToPolar` převede kartézské souřadnice na polární. Jelikož ve funkci potřebujeme využít vzdálenost bodu od počátku vícekrát, vytvoříme si pomocí uzávěru funkci, jejíž argumentem je právě tato vzdálenost. Funkci vytvořenou uzávěrem poté okamžitě aplikujeme na vypočítanou vzdálenost od počátku. V tomto případě uzávěr využíváme pouze ke zkrácení zápisu, abychom nemuseli vícekrát uvádět vzorec Pythagorovy věty. V praxi ovšem můžeme pomocí uzávěrů tímto způsobem omezit například vícenásobné provádění drahých operací (čtení z disku, komunikace po síti), nebo zabránit možným chybám (například kdy opakovanou aplikací funkce `Random` získáme různá náhodná čísla). Ve funkci využijeme konstruktor struktury pro vytvoření objektu typu `Polar`. Následně jen vytvoříme proměnnou v-konstruující objekt typu `Cart`, a převod mezi souřadnicemi otestujeme.

Druhou částí ukázky je definici rekurzivní struktury – v tomto případě stromu. Každý strom se skládá z hodnoty v kořenu stromu a seznamu podstromů. Po definici typu `Tree` si jeden strom ihned vytvoříme, abychom si mohli vyzkoušet práci se stromy.

Práci se stromy si vyzkoušíme implementací funkce `Depth`, která spočítá hloubku stromu. Hloubka stromu je rovna jedné, pokud strom nemá podstromy. Pokud má strom alespoň jeden podstrom, přičteme číslo jedna k hloubce nejhlubšího podstromu.

K implementaci funkce `Depth` potřebujeme také několik pomocných funkcí. Funkce `Depths` spočítá hloubky pro každý strom v seznamu stromů. Tuto funkci využijeme k vypočtení hloubek podstromu. Funkce `Max` vrátí největší hodnotu v seznamu celých čísel, interně však tato funkce slouží pouze ke zjednodušení aplikaci rekurzivní funkce `MaxInt`, která rekurzivně projde celý seznam a v rekurzivních aplikacích si předává nejvyšší nalezený prvek. Funkce `MaxOf` pouze vrátí větší ze dvou celých čísel.

Nakonec provedeme výpis stromu i jeho hloubky.

K přístupu k atributu instance struktury (v příkladu k objektům typu `Cart` a `Tree`) využíváme syntaktický konstrukt `::`.

```
Import "class://org.fpeterek.tilscript.math.Registrar".

Defn PrettyPrint(cons: Construction) -> Any1 :=
    ['Progn
        ['Print cons]
        ['Print '": "]
        ['Println ^2 cons]].

-- Definice typu reprezentujícího polární souřadnice
Struct Polar {
    dist: Real,
    angle: Real,
}.

-- Definice typu reprezentujícího kartézské souřadnice
Struct Cart {
    x: Real,
    y: Real,
}.

-- Výpočet úhlu mezi bodem a osou x
Defn GetAngle(x: Real, y: Real, len: Real) -> Real :=
    ['Cond
        ['= '0.0 len] '0.0
        ['> y '0.0] ['Acos ['/ x len]]
        'True ['- '360.0 ['Acos ['/ x len]]]].

-- Převod kartézských souřadnic na souřadnice polární
Defn CartToPolar(cart: Cart) -> Polar :=
    [[\dist: Real -> Polar
        {Polar dist ['GetAngle cart::x cart::y dist] }]
        ['Sqrt ['+ ['* cart::x cart::x] ['* cart::y cart::y]]] ].

cart -> Cart := {Cart '2.0 '2.0}.

['Println ['CartToPolar cart]].
```

```
-- Definice rekurzivní struktury
Struct Tree {
    value: Int,
    subtrees: List(Tree),
}.

-- Vytvoření stromu
myTree -> Tree := {Tree
    '1
    ['ListOf
        {Tree '2 {List(Tree) }}
        {Tree '3 ['ListOf {Tree '4 {List(Tree) } }]}]}.

-- Funkce vrátí větší ze dvou čísel
Defn MaxOf(fst: Int, snd: Int) -> Int :=
    ['If ['> fst snd] fst snd].

-- Funkce vrátí největší číslo v seznamu
-- Největší nalezené číslo je předáváno jako argument při rekurzivní aplikaci
Defn MaxInt(lst: List(Int), max: Int) -> Int :=
    ['If ['IsEmpty lst]
        max
        ['MaxInt ['Tail lst] ['MaxOf ['Head lst] max]]].


-- Funkce pouze rozloží seznam na první prvek a zbytek seznamu
-- a aplikuje rekurzivní funkci MaxInt
-- Funkce Max slouží ke zjednodušení aplikace MaxInt
Defn Max(lst: List(Int)) -> Int := ['MaxInt ['Tail lst] ['Head lst]].

-- Funkce Depths spočítá hloubky všech stromů v seznamu
Defn Depths(trees: List(Tree)) -> List(Int) :=
    ['If ['IsEmpty trees]
        {List(Int) }
        ['Cons
            ['Depth ['Head trees]]
            ['Depths ['Tail trees]]]].
```

```
-- Výpočet hloubky stromu
Defn Depth(tree: Tree) -> Int :=
    ['If ['IsEmpty tree::subtrees]
        '1
        ['+ '1 ['Max ['Depths tree::subtrees]]]].

['PrettyPrint 'myTree].
['PrettyPrint 'myTree::subtrees].
['PrettyPrint '['Depth myTree]].
```

Ukázka 5.7: Práce se strukturami

5.1.7 Intenze

Všechny příklady zatím pracovali pouze s extenzemi. Důvodem bylo převážně zjednodušení ukázek. Nyní si však ukážeme také příklad práce s intenzemi, neboť s nimi bezpochyby budeme chtít v praxi pracovat.

V ukázce implementujeme intenze v jazyce TIL-Script. Databázi definujeme pomocí tzv. aktuálních extenzí, tedy funkcí popisujících stav světa v konkrétních časovým okamžicích, případně intervalech, pokud funkce popisuje interval okamžiků. Dále definujeme intenze, které při extenzionalizaci podle dodaných argumentů správně vyberou aktuální extenzi popisující požadovaný světamih. Databáze bude přiložena v příloze.

Kromě intenzí v databázi, která je součástí příkladu, definujeme také individua, se kterými budeme pracovat. Konkrétně se jedná o individua `Adela`, `Karel` a `Vaclav`.

Definice intenzí v jazyce TIL-Script je ovšem zdlouhavá a pro větší databázi může být velmi nepřehledná. Pro praktické využití může být žádoucí využít databázové systémy pro uložení dat, a pomocí již existující Java knihovny pro daný databázový systém vytvořit TIL-Script knihovnu, která uživateli umožní s databází komunikovat.

V ukázce opět importujeme matematickou knihovnu. Dále importujeme také databázi obsahující potřebné intenze. Soubor definující intenze se jmenuje `intensions.tils`. Za účelem výpisu v ukázce také definujeme funkci `PrettyPrint`.

Jazyk TIL-Script definuje fyzickou reprezentaci časových okamžiků jako 64-bitové číslo, nijak však nepředepisuje, jak má uživatel dané číslo intepretovat. Proto pro zjednodušení budeme v ukázce využívat rozumně malá čísla, neboť využívání např. unixového času by vedlo ke špatně čitelné ukázce.

Proměnná `w -> World` je definována ve standardní knihovně. V ukázce musíme definovat pouze proměnnou `t -> Time`, pomocí které budeme určovat časové okamžiky, které zkoumáme. Hodnotu proměnné několikrát změníme, abychom mohli prozkoumat více časových okamžiků. Protože v ja-

zyce TIL-Script jsou časové okamžiky a čísla reprezentovány jiným typem, musíme literál celého čísla konvertovat na časový okamžik pomocí funkce `IntToTime`.

Také musíme deklarovat funkci `LastDigit`, která vrátí poslední číslici reálného čísla, pokud taková číslice existuje. Protože však funkci nepotřebujeme aplikovat, nemusíme ji definovat (tedy dodávat předpis funkce), stačí nám pouze uvést její jméno a typ.

V ukázce ve většině případů využíváme notaci `@wt`, která slouží pouze jako zkratka za aplikaci intenze nejprve na svět, a následně aplikaci získané funkce na časový okamžik. Tato notace je čistě syntaktická a parser jazyka TIL-Script ji přepíše na Kompozici během zpracování a analýzy syntaktického stromu.

Součástí databáze je individuová role `BestStudent/((Indiv Time)World)`, která specifikuje nejlepšího studenta v daném okamžiku (podle naší arbitrárně zvolené metriky), a funkce `IsComputing /(((Bool Indiv Construction)Time)World)`, která uvádí, zda uvedené individuum v daném světamihu počítá výraz představovaný konstrukcí.

Při výpisu nejlepšího studenta může na první pohled působit překvapivě dvojí trivializace. Zde je třeba si uvědomit, že zápis `@wt` slouží jako zkratka za Kompozici. Pravou trivializací zmíníme funkci `BestStudent`, kterou pomocí Kompozice extenzionalizujeme. Levá trivializace poté trivializuje právě konstrukci extenzionalizace funkce `BestStudent`. Jelikož se jedná o individuovou roli, extenzionalizací získáme konkrétní individuum, ne funkci. V příkladu s funkcí `IsComputing` již extenzionalizaci provádíme jako podkonstrukci jiné kompozice, jež získanou aktuální extenzi aplikuje na požadované argumenty.

Dále je již ukázka velmi přímočará. Pouze extenzionalizujeme žádané intenze v různých světamizích, a zkoumáme stavy světa. Sledujeme, kdo byl nejlepším studentem v konkrétních časových okamžicích, nebo jaké příklady studenti počítali. V předposlední větě programu pouze funkci `BestStudent` extenzionalizujeme pomocí explicitně uvedených Kompozic, abychom dokázali, že zápis `@wt` je čistě syntaktický a slouží pouze ke zkrácení zápisu.

```
Import "class://org.fpeterek.tilscript.math.Registrar".
Import "intensions.tils".

Defn PrettyPrint(cons: Construction) -> Any1 :=
    ['Progn
        ['Print cons]
        ['Print '": "]
        ['Println ^2cons]].

t -> Time := ['IntToTime '130].

LastDigit/(Int Real).
```

```
['PrettyPrint ''BestStudent@wt].

['PrettyPrint ['IsComputing@wt 'Adela ['Ln '14.0]]].
['PrettyPrint ['IsComputing@wt 'Karel ['/ '18 '6]]].

t -> Time := ['IntToTime '250].

['Println 'BestStudent@wt].

['PrettyPrint ['IsComputing@wt 'Adela ['Ln '14.0]]].
['PrettyPrint ['IsComputing@wt 'Adela ['Cos 'Pi]]].
['PrettyPrint ['IsComputing@wt 'Karel ['/ '18 '6]]].

t -> Time := ['IntToTime '600].

['Println [['BestStudent w] t]].

['PrettyPrint ['IsComputing@wt 'Vaclav ['LastDigit 'Pi]]].
```

Ukázka 5.8: Funkce a uzávěry

5.2 Překlad programu

Překladač byl psán pro platformu Java, proto pro spuštění překladače jazyka TIL-Script musíme mít nainstalované Java prostředí (JRE). Máme-li JRE nainstalované, překladač můžeme spustit ručně, nebo pomocí přiloženého pomocného skriptu.

Překladač spouštíme vždy z příkazové řádky, neboť pro něj momentálně neexistuje grafické rozhraní.

Při ručním spuštění je třeba manuálně spustit Java prostředí a specifikovat JAR soubor obsahující kód TIL-Script překladače. Překladači je potřeba předat jako argument název souborů, které chceme přeložit. Pokud je interpret TIL-Scriptu jediný Java archiv, který načítáme, není třeba specifikovat tzv. *Main Class*, tedy třídu obsahující statickou metodu `void main()` (neboť specifikace této třídy je součástí souboru `manifest` obsaženém v archivu).

```
java -jar tilscript.jar script.tils
```

Ukázka 5.9: Spuštění překladače

Pokud chceme kromě překladače načíst také TIL-Script knihovny, musíme uvést nejen všechny archivy, jenž potřebuje Java prostředí načíst, ale také hlavní třídu.

```
java -cp tilscript.jar:libs/math.jar org.fpeterek.tilscript.interpreter.MainKt
    script.tils
```

Ukázka 5.10: Spuštění překladače s načtením knihoven

5.2.1 tilscript.sh

Nejjednodušší způsob, jak překladač jazyka TIL-Script spustit, je využít pomocný skript `tilscript.sh`. Skript `tilscript.sh` využívá pouze funkcionalitu definovanou standardem POSIX, proto by tento skript měl fungovat korektně na všech operačních systémech splňujících standard POSIX. Dále se standardu POSIX musí držet také shell, který bude tento pomocný skript interpretovat[1].

Skript `tilscript.sh` předpokládá, že se nachází ve stejné složce jako soubor `tilscript.jar`, tedy archiv obsahující přeložený kód překladače. Dále tento skript předpokládá existenci adresáře `libs/`, opět ve stejné složce, jako skript samotný. Skript při spuštění automaticky načte všechny Java archivy ve složce `libs/`, spustí Java prostředí, zajistí načtení všech knihoven i TIL-Script překladače a korektně uvede hlavní třídu překladače. Všechny argumenty, které skript obdrží, poté automaticky předá TIL-Script překladači.

```
./tilscript.sh script.tils
```

Ukázka 5.11: Spuštění překladače za využití pomocného skriptu

5.3 Standardní knihovna

Standardní knihovna jazyka TIL-Script obsahuje základní funkce pro práci s objekty Transparentní intenzionální logiky. Dále obsahuje definice atomických typů a tří proměnných. Z důvodu náročnosti implementace některým funkcím chybí implementace, proto je můžeme pouze zmínit, nemůžeme však provést jejich aplikaci.

Nakonec je třeba uvést, že současný stav nemusí reprezentovat také konečný stav standardní knihovny. Na základě zpětné vazby uživatelů lze standardní knihovnu v budoucnu rozšiřovat.

5.3.1 Funkce

5.3.1.1 Deklarace

Zde uvedeným funkcím chybí implementace z důvodu její náročnosti a časové složitosti. Představme si například všeobecný kvantifikátor. Logickou pravdivost řady tvrzení či úsudků lze dokázat například pomocí důkazových kalkulů (rezoluční metoda, přirozená dedukce). Tyto metody dokazování

[1] Můžeme tedy používat například ZSH nebo Bash. Naopak shell Fish není kompatibilní se standardem POSIX, proto skript nebude schopen interpretovat.

jsou ovšem čistě syntaktické, nedokáží tedy dokázat pravdivost tvrzení jako například $\forall x[P(x)]$. V takových případech bychom se museli uchýlit k sémantické analýze predikátu P. Predikát ovšem může být netriviální a jeho analýza velmi složitá. Iterace přes celý obor hodnot predikátu je naopak nepraktická nebo nemožná. Pokud by predikát P byla funkce typu $(o\tau)$ (nezapomeňme, že v Transparentní intenzionální logice pracujeme pouze s funkcemi), iterovat přes obor hodnot by bylo nemožné, neboť množina reálných čísel je nespočetná. Množina všech validních hodnot 64-bitového čísla s plovoucí řádovou čárkou spočetná je, proto je možné přes tato čísla iterovat, není to však praktické, neboť by výpočet nemusel v skončit v rozumném čase.

Funkce: `ForAll`

Typ: `(Bool (Bool Any1))`
Všeobecný kvantifikátor

Funkce: `Exist`

Typ: `(Bool (Bool Any1))`
Existenční kvantifikátor

Funkce: `Sing`

Typ: `(Bool (Bool Any1))`
Singularizátor

Funkce: `Every`

Typ: `((Bool (Bool Any1))(Bool Any1))`
Omezený kvantifikátor

Funkce: `Some`

Typ: `((Bool (Bool Any1))(Bool Any1))`
Omezený kvantifikátor

Funkce: `No`

Typ: `((Bool (Bool Any1))(Bool Any1))`
Omezený kvantifikátor

Funkce: `Sub`

Typ: `(Construction Construction Construction Construction)`
Funkce *Sub* substituční metody

Funkce: `TrueC`

Typ: `(Bool Construction)`
Třída konstrukcí pravdivých ve všech valuacích v.

Funkce: `FalseC`

Typ: `(Bool Construction)`
Třída konstrukcí nepravdivých ve všech valuacích v.

Funkce: `ImproperC`

Typ: `(Bool Construction)`
Třída konstrukcí v-nevlastních pro všechny valuace v.

Funkce: `TrueC`

Typ: `(Bool ((Bool Time)World))`
Třída propozic pravdivých ve všech valuacích v.

Funkce: `FalseC`

Typ: `(Bool ((Bool Time)World))`
Třída propozic nepravdivých ve všech valuacích v.

Funkce: `TrueC`

Typ: `(Bool ((Bool Time)World))`
Třída propozic v-nevlastních ve všech valuacích v.

5.3.1.2 Definice

Následující funkce již jsou definovány a lze provést jejich aplikaci na argumenty. Ke každé funkci je přiložena ukázka jejího využití.

Funkce: `+, -, *, /`

Typ: `(Int Int Int)`, `(Real Real Real)`
Aritmetické operace. Tyto funkce jsou definovány jak pro reálná, tak pro celá čísla, oba argumenty však musí být stejného typu.

```
['+ '1 '2].
['* '3.14159 '2.71828].
```

Ukázka 5.12: Ukázka využití aritmetických operací

Funkce: =

Typ: (Bool Any1 Any1)

Funkce = slouží k porovnávání objektů. Porovnání objektů je automaticky definováno pro všechny typy, včetně seznamů, n-tic, i uživatelem definovaných struktur. Porovnávání objektů molekulárních typů probíhá prvek po prvku.

```
['= ['ListOf '1 '2 '3] ['Cons '1 ['ListOf '2 '3]]].
```

Ukázka 5.13: Ukázka využití funkce =

Funkce: ListOf

ListOf slouží pouze jako syntaktický cukr pro tvorbu seznamů. ListOf aplikujeme na alespoň jeden argument, počet argumentů je ale shora neomezený. Jediným omezením je, že všechny argumenty musí být stejného typu. Parser jazyka TIL-Script poté aplikaci ListOf převede na korektní sestavení seznamu pomocí funkcí ListOfOne a Cons.

```
-- Následující dvě konstrukce jsou ekvivalentní
['ListOf '1 '2 '3 '4].
['Cons '1 ['Cons '2 ['Cons '3 ['ListOfOne '4]]]].
```

Ukázka 5.14: Ukázka využití ListOf

Funkce: ListOfOne

Typ: (List(Any1)Any1)

Funkce ListOfOne vytvoří seznam obsahující jediný prvek.

```
['ListOfOne 'Pi].
```

Ukázka 5.15: Ukázka využití ListOfOne

Funkce: Cons

Typ: (List(Any1)Any1 List(Any1))

Funkce Cons vytvoří nový seznam vložením prvku na začátek již existujícího seznamu. Jelikož jsou seznamy definovány induktivně, a zároveň jsou neměnné, není třeba již existující seznam kopírovat. Proto lze tuto operaci provést v konstantním čase.

```
['Cons 'Pi ['ListOf '1 '2 '3]].
```

Ukázka 5.16: Ukázka využití Cons

Funkce: `Head`

Typ: `(Any1 List(Any1))`

Funkce `Head` vrátí první prvek seznamu. Seznam musí být neprázdný, v opačném případě funkce nevrací nic (vrací `Nil`).

```
['Head ['ListOf '1 '2 '3]]. -- Konstruuje 1
```

Ukázka 5.17: Ukázka využití Head

Funkce: `Tail`

Typ: `(List(Any1)List(Any1))`

Funkce `Tail` vrátí seznam bez jeho prvního prvku. Funkce je nedefinovaná pro prázdné seznamy.

```
['Tail ['ListOf '1 '2 '3]]. -- Konstruuje seznam [2, 3]
```

Ukázka 5.18: Ukázka využití Head

Funkce: `IsEmpty`

Typ: `(Bool List(Any1))`

Aplikací `IsEmpty` na prázdný seznam získáme hodnotu `True`. Aplikací na neprázdný seznam získáme `False`.

```
['IsEmpty ['ListOf '1 '2 '3]]. -- False
['IsEmpty ['Tail ['ListOfOne '1]]]. -- True
```

Ukázka 5.19: Ukázka využití IsEmpty

Funkce: `EmptyListOf`

Typ: `(List(Any1)Type)`

Funkce `EmptyListOf` jako svůj jediný vstup přijímá objekt typu `Type`. Výsledkem aplikace na typ je poté prázdný seznam objektů specifikovaného typu.

```
['IsEmptyOf 'Int].
```

Ukázka 5.20: Ukázka využití EmptyListOf

Funkce: `Print`, `Println`

Typ: `(Any1 Any1)`

Funkce `Print`, `Println` zajistí výpis svého argumentu na standardní výstup programu a poté vrátí svůj jediný argument. Funkce `Println` vypíše také oddělovač řádků (v systému GNU/Linux znak LF, v systému Windows sekvenci CRLF). Funkce `Print` tento oddělovač nevypisuje.

```
['Println '['+ '1 '2]]. -- Vypíše konstrukci ['+ '1 '2]
['Print ['+ '1 '2]]. -- Vypíše konstrukci ['+ '1 '2]
```

Ukázka 5.21: Ukázka využití Print, Println

Funkce: `If`

Typ: `(Any1 Bool Any1 Any1)`

Funkce `If` je, na rozdíl od všech ostatních funkcí, prováděna líně. Funkce `If` přijímá tři argumenty. První argument je objekt typu `Bool`. Druhým argumentem je hodnota, kterou funkce vrátí, je-li první argument `True`. Jinak funkce `If` vrátí svůj druhý argument. Konstrukce, která konstruuje argument funkce `If`, je ovšem provedena až poté, co překladač zkontroluje hodnotu prvního argumentu, aby nedošlo ke zbytečnému provedení konstrukce a následnému zahození výsledku.

```
['If ['> x y]
    ['Println '"x is greater than y"]
    ['Println '"y is greater than or equal to x"]].
```

Ukázka 5.22: Ukázka využití If

Funkce: `Cond`

`Cond` slouží pouze jako syntaktický cukr pro funkci `If`. `Cond` tedy není funkcí sama o sobě, překladačem je ovšem při procesu tvorby AST přeložena na sérii vnořených aplikací funkce `If`.

Máme-li pouze jednu podmínku se dvěma větvemi, *if* a *else*, je využití funkce `If` jednoduché a čitelné. Máme-li však větví více, musíme aplikace funkce `If` zanořit – větví *else* tak musí být další aplikace `If`.

Syntaktický cukr `Cond` nám umožňuje zjednodušit zápis funkce `If` a vyhnout se zanořování. `Cond` přijímá nespecifikovaný, avšak sudý počet argumentů. Lichým argumentem je vždy podmínka. Sudým argumentem je poté konstrukce, která se provede, je-li podmínka pravdivá. Překladač během překladu přeloží aplikaci `Cond` na zanořené aplikace funkce `If`. Podmínky se tedy nevyhodnocují všechny najednou, ale jedna po druhé, dokud není nalezena první pravdivá podmínka. Není-li pravdivá ani jedna podmínka, `Cond` vrací `Nil`.

```
['Cond
    ['= x '2] ['Log2 y]
    ['= x '10] ['Log10 y]
    'True ['Ln y]]. -- catch-all podmínka
```

Ukázka 5.23: Ukázka využití Cond

Funkce: Progn

Typ: (Any2 Any1 Any2)

Funkce Progn přijímá dva argumenty. První argument ignoruje, druhý argument vrátí, jak jej dostala. Funkce Progn je tedy ekvivalentem funkce *False* lambda kalkulu ($\lambda x \lambda y.y$, případně $\lambda x, yy$ v TIL). Díky principu kompozicionality je aplikace Progn v-nevlastní, neobdrží-li první argument. V takovém případě tedy vůbec nedojde k vyhodnocení druhého argumentu.

Funkci Progn využijeme například při výpisu do souboru nebo na standardní výstup. Pro analýzu přirozeného jazyka většinou není potřeba. Kódovat přirozená čísla pomocí Churchova kódování v Transparentní intenzionální logice také nepotřebujeme.

Funkce Progn je binární funkcí, existuje pro ni ovšem podobný syntaktický cukr, jako pro Cond nebo ListOf. Předáme-li funkci Progn více než dva argumenty, překladač aplikaci Progn na více argumentů rozepíše na vnořené aplikace binární funkce Progn. Na rozdíl od Cond, Progn je tedy skutečnou funkcí.

Název Progn je převzat z jazyka Lisp.

```
-- Vypíše 'x + y = ' a součet x a y
-- Vrátí součet x+y, protože Println vrátí svůj argument
['Progn
    ['Print '"x + y = "]
    ['Println ['+ x y]]].
```

Ukázka 5.24: Ukázka využití Progn

Funkce: Tr

Typ: (Construction Any1)

Trivializuje svůj argument.

```
['Tr ['+ '1 '2]]. -- Kompozice konstruuje '3
```

Ukázka 5.25: Ukázka využití Tr

Funkce: `TypeOf`

Typ: `(Type Any1)`

Vrátí typ svého argumentu. Může být užitečné například v typově polymorfních funkcích, potřebujeme-li provést rozhodnutí na základě typu argumentu.

```
-- Volba funkce na základě typu argumentu typově polymorfní funkce
[\x: Any1 -> Any1 [['If ['= ['TypeOf x] 'Int] 'DiscreteLog 'Log10] x]].
```

Ukázka 5.26: Ukázka využití TypeOf

Funkce: `IsNil`

Typ: `(Bool Any1)`

Vrátí `True`, neobdrží-li žádný argument (tedy je-li argumentem `Nil`). Tato funkce porušuje princip kompozicionality, je ovšem potřeba např. k ošetření chyb.

```
x -> Real = ['/ a b].
['If ['IsNil x]
    -- Vypíšeme chybu a ukončíme program
    ['Progn
        ['Println '"Program obdržel nesprávný vstup"]
        ['Exit '-1]]
    -- Program obdržel validní vstup, hodnota 1 bude ignorována
    '1].
```

Ukázka 5.27: Ukázka využití IsNil

Funkce: `Exit`

Aplikací funkce `Exit` ukončíme překlad programu. Argument funkce určuje návratovou hodnotu programu.

```
x -> Real = ['/ a b].
['If ['IsNil x]
    -- Vypíšeme chybu a ukončíme program
    ['Progn
        ['Println '"Program obdržel nesprávný vstup"]
        ['Exit '-1]]
    -- Program obdržel validní vstup, hodnota 1 bude ignorována
    '1].
```

Ukázka 5.28: Ukázka využití Exit

Funkce: `RandomInt`

Typ: `(Int DeviceState)`

Funkce `RandomInt` vrací náhodné celé číslo v intervalu $\langle 0; 2^{64} - 1 \rangle$.

```
['RandomInt deviceState].
```

Ukázka 5.29: Ukázka využití RandomInt

Funkce: `Random`

Typ: `(Real DeviceState)`

Funkce `Random` vrací náhodné reálné číslo v intervalu $\langle 0; 1 \rangle$.

```
['* ['Random deviceState] '100]. -- náhodné číslo v rozsahu 0-100
```

Ukázka 5.30: Ukázka využití Random

Funkce: `Not`

Typ: `(Bool Bool)`

Logická negace.

```
['Not ['= x y]].
```

Ukázka 5.31: Ukázka využití Not

Funkce: `And`

Typ: `(Bool Bool Bool)`

Logická konjunkce.

```
[And ['= x y] ['= x z]].
```

Ukázka 5.32: Ukázka využití And

Funkce: `Or`

Typ: `(Bool Bool Bool)`

Logická disjunkce.

```
[Or ['= x y] ['= x z]].
```

Ukázka 5.33: Ukázka využití Or

Funkce: `Implies`

Typ: `(Bool Bool Bool)`

Implikace.

```
['Implies ['And ['= x y] ['= x z]] ['= y z]].
```

Ukázka 5.34: Ukázka využití Implies

Funkce: `OneTuple`

Typ: `(Tuple(Any1)Any1)`

Vytvoří n-tici obsahující právě jeden prvek – obdržený argument. Lze využít například při rekurzivní tvorbě n-tice.

```
['OneTuple '1].
```

Ukázka 5.35: Ukázka využití OneTuple

Funkce: `MkTuple`

`MkTuple` slouží jako syntaktický cukr pro tvorbu n-tic. `MkTuple` přijímá nespecifikovaný počet argumentů, překladačem je poté přeložena na aplikaci `OneTuple` a následné vložení prvků na začátek n-tice.

```
['MkTuple '1 '3.5 'True].
```

Ukázka 5.36: Ukázka využití OneTuple

Funkce: `PrependToTuple`

Typ: `(Any1 Any2 Any3)`

`PrependToTuple` přijímá dva argumenty. Prvním argumentem je libovolná hodnota. Druhý argument musí být vždy n-tice. `PrependToTuple` vytvoří novou (n+1)-tici vložením prvního argumentu na začátek n-tice specifikované druhým argumentem.

```
['PrependToTuple '4 ['MkTuple '1 '3.5 'True]]. -- Vytvoří n-tici (4, 1, 3.5, True)
```

Ukázka 5.37: Ukázka využití PrependToTuple

Funkce: `Get`

Typ: `(Any1 Any2 Int)`

Funkce `Get` umožňuje získat prvek n-tice na požadované pozici. Prvním argumentem je n-tice, druhým je index prvku. Prvky indexujeme od nuly. Funkce `Get` je nedefinovaná, je-li index větší nebo roven velikosti n-tice.

```
['Get '1 ['MkTuple 'E 'Pi]]. -- Pi
```

Ukázka 5.38: Ukázka využití Get

Funkce: `TupleLen`

Typ: `(Int Any1)`

Funkce `TupleLen` vrátí délku n-tice. Využití funkce je převážně v typově polymorfních funkcích, neboť není-li funkce typově polymorfní, museli jsme typ explicitně uvést, a tedy délku n-tice známe. Argumentem musí být n-tice, funkce je typově polymorfní proto, aby argumentem mohly být n-tice libovolné délky.

```
-- vynásobíme první prvek n-tice délkou n-tice
[\x: Any1 -> Int ['* ['TupleLen x] ['Get x '0]]].
```

Ukázka 5.39: Ukázka využití TupleLen

Funkce: `Char`

Typ: `(Text Text Int)`

Funkce `Char` vrátí znak textového řetězce na požadované pozici. Prvky indexujeme od nuly. Funkce `Char` je nedefinovaná, je-li index větší nebo roven velikosti řetězce.

```
['Char '"TIL" '1]. -- "I"
```

Ukázka 5.40: Ukázka využití Char

Funkce: `CatS`

Typ: `(Text Text Text)`

Funkce `CatS` spojí dva textové řetězce.

```
-- Kompozice konstruuje řetězec "Transparentní intenzionální logika"
['CatS '"Transparentní intenzionální" '" logika"].
```

Ukázka 5.41: Ukázka využití Char

Funkce: `HeadS`

Typ: `(Text Text)`

Funkce `HeadS` vrátí první znak řetězce. Funkce je nedefinovaná pro prázdné řetězce.

```
['HeadS '"TIL"]. -- "T"
```

Ukázka 5.42: Ukázka využití HeadS

Funkce: `TailS`

Typ: `(Text Text)`

Funkce `HeadS` vrátí řetězec bez prvního znaku. Funkce je nedefinovaná pro prázdné řetězce.

```
[TailS '"TIL"]. -- "IL"
```

Ukázka 5.43: Ukázka využití TailS

Funkce: `LenS`

Typ: `(Int Text)`

Funkce `LenS` vrátí délku textového řetězce.

```
[LenS '"TIL"]. -- 3
```

Ukázka 5.44: Ukázka využití LenS

Funkce: `IsBefore`, `IsBeforeOrEq`, `IsAfter`, `IsAfterOrEq`

Typ: `(Bool Time)`

Funkce slouží k porovnávání časových okamžiků. Pro obyčejnou rovnost lze využít funkci `=`.

```
['IsBefore t1 t2].
```

Ukázka 5.45: Ukázka porovnávání časových okamžiků

Funkce: `Now`

Typ: `(Time DeviceState)`

Funkce `Now` vrátí aktuální systémový čas.

```
[['PrezidentCR ['Now deviceState]] w].
```

Ukázka 5.46: Ukázka využití Now

Funkce: `IntToText`, `RealToText`, `IntToTime`, `TextToInt`, `TextToReal`, `TimeToInt`, `IntToReal`, `RealToInt`

Funkce slouží k převodu mezi typy. Při převodu z reálných čísel na čísla celá dochází k ořezání čísla (zaokrouhlení směrem k nule).

```
['Log ['IntToReal ['+ '2 '3]]].
['CatS '"Log 10 = " ['RealToText ['Log '10]]].
```

Ukázka 5.47: Ukázka využití konverzí

Funkce: `IsVariable`, `IsComposition`, `IsClosure`, `IsExecution`, `IsFunction`, `IsTrivialization`, `IsSymbol`, `IsList`, `IsValue`, `IsTuple`, `IsConstruction`, `IsStruct`

Typ: `(Bool Any1)`

Tyto funkce umožňují ověřit, zda je argument konkrétní hodnotou (`IsValue`), symbolickou hodnotou (`IsSymbol`), seznamem, n-ticí, strukturou nebo konstrukcí. Také nám umožňují určit, o jaký typ konstrukce se jedná.

```
[\x: Int -> Int
    ['If ['IsSymbol x]
        Nil -- neumíme umocnit symbolickou hodnotu
        ['* x x]]]. -- konkrétní hodnotu umocníme
```

Ukázka 5.48: Ukázka využití IsSymbol

Funkce: `NilAt`

Typ: `(Int Text Tuple(Int Int Text))`

Funkce `NilAt` je nedefinována na všech argumentech. Funkce umožňuje zkonstruovat hodnotu `Nil`. `Nil` jde také zmínit přímo, `NilAt` nám ovšem umožňuje vnitřní reprezentaci `Nil` obohatit také o důvod, proč byla konstrukce v-nevlastní, a pozici, kde byla zavolána funkce, jež `Nil` vrátila. Tímto způsobem můžeme dosáhnout lepšího hlášení chyb.

Argumentem funkce je důvod, proč byla vrácena hodnota `Nil`, a pozice, kde došlo k aplikaci funkci na argumenty, na kterých není definována. Často zde budeme chtít použít proměnnou `callsite`, viz 5.3.3.3 – `callsite -> Tuple(Text Int Int)`.

Ukázka je schválně jednoduchá, logaritmus v matematické knihovně tuto situaci řeší sám, v praxi tedy není třeba kontrolovat argument logaritmu manuálně. Proměnná `callsite` konstruuje pozici ve zdrojovém kódu, kde došlo k aplikaci funkce konstruované uzávěrem.

```
[\x: Real -> Real
    ['If ['Not ['> x 0.0]]
```

```
['NilAt '"Argument logaritmu musí být kladný" 'callsite]
['Log x]]].
```

Ukázka 5.49: Ukázka využití NilAt

5.3.2 Typy

Standardní knihovna jazyka TIL-Script definuje atomické typy popsané v kapitole 3 – TIL-Script, tedy typy `Bool`, `Type`, `Int`, `Text`, `Indiv`, `Real`, `Time`, `World`, `DeviceState`.

Dále definuje seznamy a n-tice. Žádné struktury standardní knihovna nedefinuje.

5.3.3 Proměnné

Standardní knihovna definuje pouze tři proměnné.

5.3.3.1 `w -> World`

Proměnná `w` označuje současný svět. Objekty typu `World` nemají žádnou inherentní hodnotu, využívají se hlavně k označení intenzí. Proměnná `w` proto existuje, aby si ji uživatel nemusel explicitně vytvářet sám.

5.3.3.2 `deviceState -> DeviceState`

Proměnná `deviceState` existuje ze stejného důvodu, jako proměnná `w`. Objekty typu `DeviceState` nemají žádnou konkrétní hodnotu a jsou využívány pouze pro označení funkcí, jejichž hodnota závisí na stavu zařízení, na němž je program spuštěn.

5.3.3.3 `callsite -> Tuple(Text Int Int)`

Proměnná `callsite` je automaticky vytvářená proměnná. Proměnná `callsite` je vytvořena při aplikaci funkce, a obsahuje pozici ve zdrojovém kódu, kde byla aplikace funkce provedena. Proměnná slouží především k umožnění lepšího hlášení chyb.

Proměnná slouží převážně jako argument funkce `NilAt`, viz 5.3.1.2.

5.4 Matematická knihovna

Matematická knihovna slouží také jako ukázka tvorby TIL-Script knihoven v jazycích kompilovaných do JVM bytekódu, nebo jako test importu jmen z Java archivů. Přesto obsahuje několik užitečných definic. Na rozdíl od standardní knihovny, všechny funkce matematické knihovny jsou plně definované, a lze tedy provést jejich aplikaci na argumenty.

5.4.1 Funkce

Funkce: Sin

Typ: (Real Real)

Funkce sinus. Kromě konkrétních hodnot je definována také pro symbolickou hodnotu Pi/Real.

```
['Sin 'Pi]. -- 0
['Sin ['/ '3.14159 '2.0]]. -- přibližně 1
```

Ukázka 5.50: Ukázka využití Sin

Funkce: Asin

Typ: (Real Real)

Inverzní funkce k funkci sinus.

```
['Asin '0.5].
```

Ukázka 5.51: Ukázka využití Asin

Funkce: Cos

Typ: (Real Real)

Funkce cosinus. Kromě konkrétních hodnot je definována také pro symbolickou hodnotu Pi/Real.

```
['Cos 'Pi]. -- -1
['Cos ['/ '3.14159 '2.0]]. -- přibližně 0
```

Ukázka 5.52: Ukázka využití Cos

Funkce: Acos

Typ: (Real Real)

Inverzní funkce k funkci kosinus.

```
['Acos '0.5].
```

Ukázka 5.53: Ukázka využití Acos

Funkce: `Tan`

Typ: `(Real Real)`

Funkce tangens. Kromě konkrétních hodnot je definována také pro symbolickou hodnotu `Pi`/`Real`.

```
['Tan 'Pi]. -- 0
```

Ukázka 5.54: Ukázka využití Tan

Funkce: `Ln`, `Log2`, `Log10`

Typ: `(Real Real)`

Přirozený logaritmus, logaritmus o základu dvě a logaritmus o základu 10. Přirozený logaritmus je definován také pro symbolickou hodnotu `E`/`Real`.

```
['Ln ['* '2.71828 '2.71828]]. -- Přibližně 2
['Log2 '1024]. -- 10
['Log10 '1000]. -- 3
```

Ukázka 5.55: Ukázka využití Ln, Log2, Log10

Funkce: `Log`

Typ: `(Real Real Real)`

Logaritmus o libovolném základu. Prvním argumentem je číslo, jehož logaritmus chceme spočítat, druhým argumentem je základ logaritmu.

```
['Log '27.0 '3.0]. -- 3
```

Ukázka 5.56: Ukázka využití Log

Funkce: `Round`

Typ: `(Real Real)`

Funkce `Round` umožňuje zaokrouhlit reálné číslo na jednotky.

```
['Round '3.2]. -- 3
['Round '3.7]. -- 4
```

Ukázka 5.57: Ukázka využití Round

Funkce: `Sqrt`

Typ: `(Real Real)`

Funkce `Sqrt` vrátí druhou odmocninu svého argumentu.

```
['Round '3.2]. -- 3
['Round '3.7]. -- 4
```

Ukázka 5.58: Ukázka využití Round

5.4.2 Symbolické hodnoty

Hodnota `Pi`/`Real`

Hodnota `Pi` nám umožňuje symbolicky zmínit číslo π, viz 3.2.15 – Symbolické hodnoty.

```
['Sin 'Pi].
```

Ukázka 5.59: Ukázka využití Pi

Hodnota `E`/`Real`

Hodnota `E` nám umožňuje symbolicky zmínit Eulerovo číslo e, viz 3.2.15 – Symbolické hodnoty.

```
['Ln 'E].
```

Ukázka 5.60: Ukázka využití E

5.4.3 Proměnné

Proměnná `pi -> Real`

Proměnná `pi -> Real` aproximuje číslo π s přesností na 15 desetinných míst.

```
['* pi '2]. -- přibližně 6.28
```

Ukázka 5.61: Ukázka využití proměnné pi

Proměnná `e -> Real`

Proměnná `e -> Real` aproximuje Eulerovo číslo s přesností na 15 desetinných míst.

```
['Ln ['* e ['* e e]]]. -- 3
```

Ukázka 5.62: Ukázka využití proměnné e

5.5 Implementace knihovny

Nyní si ukážeme, jak implementovat vlastní TIL-Script knihovnu v jazyce Java. Ukázku implementace v jazyce Kotlin přiložíme za účelem porovnávání těchto dvou jazyků v příloze. V praxi lze využít libovolný jazyk kompilovaný pro platformu JVM, nejen Javu nebo Kotlin. Implementovat budeme funkci `InvSqrt/(Real Real)`, tedy převrácenou hodnotu druhé odmocniny.

5.5.1 Implementace funkce

Začneme vytvořením třídy `InvSqrt`. Tato třída musí být potomkem třídy `DefaultFunction`, abychom ji mohli použít jako TIL-Script funkci.

```
public class InvSqrt extends DefaultFunction {

}
```

Ukázka 5.63: Třída InvSqrt

Následně musíme implementovat konstruktor třídy `InvSqrt`. V konstruktoru musíme zavolat konstruktor předka – třídy `DefaultFunction`. Konstruktor `DefaultFunction` přijímá tři argumenty.

Prvním je jméno funkce – tímto jménem budeme funkci označovat v jazyce TIL-Script. Na jméně třídy, která danou funkci reprezentuje, nezáleží, kvůli přehlednosti je však vhodné třídu pojmenovat podle názvu funkce.

Druhým argumentem je obor hodnot – typ objektu, který funkce vrátí. Abychom si zkrátili zápis, vytváříme v kódu statickou proměnnou `real`.

Třetím argumentem je seznam proměnných – argumentů funkce. Zde nám postačí jeden argument. Opět vytvoříme pomocnou proměnnou, tentokrát pojmenovanou `arg`. Konstruktor třídy `Variable` přijímá pět argumentů. Prvním je název proměnné. Druhým argumentem je pozice ve zdrojovém kódu – tento argument se využívá v parseru jazyka TIL-Script a využívá se k hlášení chyb. Jelikož je funkce implementovaná v Javě, ne v jazyce TIL-Script, nastavíme pozici na hodnotu, která značí, že je pozice neznámá. Třetí argument konstruktoru určuje typ proměnné. Čtvrtým argumentem je seznam chyb hlášených při definici proměnné – jelikož nechceme zbytečně hlásit chyby pro vlastní proměnnou, vytvoříme prázdný seznam. Posledním argumentem je hodnota proměnné. Jelikož se ale jedná o argument funkce, není potřeba žádnou hodnotu dodávat. Při aplikaci funkce se vždy danému argumentu nastaví korektní hodnota. Nakonec tedy vytvoříme seznam obsahující tento jeden argument.

```
  private static AtomicType real = Primitives.INSTANCE.getReal();
```

```
  private static Variable arg = new Variable(
    "x",
    new SrcPosition(-1, -1, ""),
    real,
    new ArrayList<>(),
    null
  );

  public InvSqrt() {
    super(
      "InvSqrt",
      real,
      Collections.singletonList(arg)
      );
  }
```

Ukázka 5.64: Konstruktor InvSqrt

Nakonec musíme naprogramovat sémantiku funkce `InvSqrt`. Za tímto účelem musíme vytvořit metodu `apply`. Jelikož přepisujeme zděděnou abstraktní metodu, můžeme si ji nechat vygenerovat vývojovým prostředím nebo jazykovým serverem.

Jakmile máme vygenerovanou hlavičku funkce, stačí funkci naprogramovat. Funkce přijímá pouze jeden argument, proto si tento argument uložíme do pomocné proměnné, abychom nemuseli při každém přístupu k argumentu indexovat seznam argumentů.

Následně se ujistíme, že je argument Java objektem typu `Real`. Díky typové kontroly víme, že argument bude reálné číslo, pouze nevíme, zda se jedná o konkrétní číslo (Java objekt typu `Real` reprezentující reálná čísla), nebo o symbolickou hodnotu (Java objekt typu `Symbol`). Pokud argumentem není konkrétní hodnota, bohužel nevíme, jakou hodnotu potřebujeme odmocnit.

Následně potřebujeme získat hodnotu argumentu jako proměnnou typu `double`, abychom mohli provádět matematické operace v jazyku Java.

Dále se musíme ujistit, že funkce obdržela argument, na kterém je definovaná.

Poté už stačí jen spočítat výsledek a vrátit objekt typu `Real`, reprezentující reálné číslo v jazyce TIL-Script.

```
  @NotNull
  @Override
  public Construction apply(
    @NotNull InterpreterInterface interpreterInterface,
    @NotNull List<? extends Construction> args,
    @NotNull FnCallContext ctx) {
```

```
    final Construction arg = args.get(0);

    if (! (arg instanceof Real)) {
      return new Nil(
        ctx.getPosition(),
        new ArrayList<>(),
        "Argument of InvSqrt must not be symbolic"
      );
    }

    final double value = ((Real) arg).getValue();

    if (value <= 0.0) {
      return new Nil(
        ctx.getPosition(),
        new ArrayList<>(),
        "Argument of InvSqrt must be greater than zero"
      );
    }

    final double res = 1.0 / Math.sqrt(value);

    return new Real(res, ctx.getPosition(), new ArrayList<>());
  }
```

Ukázka 5.65: Konstruktor InvSqrt

Metoda `apply` přijímá tři argumenty. První argument – objekt typu `InterpreterInterface` slouží ke komunikaci s překladačem jazyka TIL-Script – v tomto případě jej nevyužijeme. Druhý argument je seznam argumentů, které obdržela funkce `InvSqrt`. Posledním argumentem je kontext aplikace funkce. Kontext obsahuje pozici ve zdrojovém kódu, kde k aplikaci funkce došlo. Kontext využíváme pro hlášení chyb – můžeme uživateli naší knihovny přesněji říct, kde aplikoval funkci na argumenty, na kterých není definována.

Konstruktor třídy `Nil` přijímá pozici ve zdrojovém kódu, kde tato hodnota vznikla, seznam hlášení, který opět bude prázdný, a nakonec důvod, proč je výsledkem aplikace funkce hodnota `Nil`. Seznam hlášení by v jazyce Kotlin nebylo třeba explicitně uvádět, neboť by byl vytvořen implicitním argumentem. Tento seznam slouží k hlášení chyb, které však nemůžou vzniknout při implementaci funkce v jazyce Java.

Konstruktor třídy `Real` přijímá reálné číslo, které daný objekt představuje, pozici ve zdrojovém kódu, kde byl daný objekt zkonstruován, a (opět prázdný) seznam hlášení.

5.5.2 Implementace registrátoru

Dále musíme implementovat vlastní registrátor – třídu, jejíž instance nám umožní importovat námi definovanou funkci. Registrátor musí implementovat rozhraní `SymbolRegistrar`. Rozhraní definuje řadu abstraktních metod – opět si je můžeme nechat vygenerovat vývojovým prostředím. Tyto metody nám umožňují registrovat např. struktury, typové aliasy (`TypeDef`), apod. Zde uvedeme pouze definici metody `getFunctions()`, jež slouží k registraci funkcí. Zbylé metody vrací pouze prázdný seznam (`java.util.ArrayList`) a neuvádíme je v ukázce za účelem zkrácení zápisu, v reálné implementaci by funkce pouze vraceli instanci prázného seznamu, nesměly by však chybět.

Metoda `getFunctions()` pouze vytvoří seznam obsahující instanci námi vytvořené funkce.

```
public class JavaMathRegistrar implements SymbolRegistrar {

  @NotNull
  @Override
  public List<FunctionInterface> getFunctions() {
    return Arrays.asList(
      new InvSqrt()
    );
  }

}
```

Ukázka 5.66: Java registrátor

5.5.3 Aplikace `InvSqrt` v TIL-Script programu

Po implementaci a sestavení naší Java knihovny již stačí pouze knihovnu importovat v TIL-Script programu a funkci aplikovat.

```
Import "class://org.fpeterek.tilscript.javamath.JavaMathRegistrar".

['Println ['InvSqrt '4.0]].
```

Ukázka 5.67: Aplikace InvSqrt

Následně program spustíme a ujistíme se, že program vypíše očekávanou hodnotu. První řádek výpisu obsahuje spuštění překladače. Druhý řádek je již výstup programu.

```
$ ./bin/tilscript.sh examples/javasqrt.tils
0.5
```

Ukázka 5.68: Aplikace InvSqrt

Kapitola 6

Závěr

Cíl práce – vytvořit funkční překladač jazyka TIL-Script, byl splněn. Byl implementován prototyp překladače, který dokáže překládat programy jazyka TIL-Script, chybí mu ovšem některé v praxi potřebné optimalizace (například optimalizace koncového volání). Nedostatky jsou však v textu zdokumentovány, aby bylo možné překladač dále rozšiřovat a vylepšovat.

Dále práce rozšířila jazyk TIL-Script o nové prvky, jako jsou například definice nových typů, komentáře, výrazy `Import`, nebo textové řetězce.

Implementována byla také standardní knihovna pro základní práci s jazykem TIL-Script, nebo matematická knihovna definující několik užitečných matematických funkcí. Standardní i matematická knihovna jsou zdokumentovány v tomto textu.

Překladač umožňuje implementovat TIL-Script funkce, které interně volají funkce jazyka Java – tím se jazyku TIL-Script otvírá také celý Java ekosystém, včetně knihoven pro jazyk Java.

Při tvorbě překladače bylo myšleno také na budoucí rozvoj. Současnou implementaci překladač je možné nahradit implementací novou. Pokud bude nový překladač implementovat potřebná rozhraní, bude tento překladač plně kompatibilní se standardní a matematickou knihovnou, případně s jakoukoliv jinou knihovnou psanou pro jazyk TIL-Script.

Nakonec byla práce také doplněna o řadu ukázek TIL-Script programů, ale také o ukázku tvorby TIL-Script knihoven pomocí jazyků Java a Kotlin. Do budoucna lze jazyk TIL-Script i překladač dále rozvíjet, kromě dříve zmíněných optimalizací by například bylo možné překladač doplnit o interaktivní prostředí, které umožní zadávat TIL-Script věty a konstrukce interaktivně a postupně je překládat. K interaktivnímu rozhraní by pak bylo možné vytvořit grafické prostředí, například skrze webovou aplikaci. Přesto je však překladač funkční a použitelný již v současné verzi, a snad tedy umožní další vývoj na poli logické analýzy přirozeného jazyka.

Zdroje

[1] Jurafsky, D., Martin, H.J.: *N-gram Language Models*
Dostupné z: `https://web.stanford.edu/~jurafsky/slp3/3.pdf`
Stanford University

[2] Mikolov, T., Chen, K., Corrado, G., Dean, J.: *Efficient Estimation of Word Representations in Vector Space*
Dostupné z: `https://arxiv.org/pdf/1301.3781.pdf`
Proceedings of Workshop at ICLR. 2013.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei: *Language models are few-shot learners*
Dostupné z: `https://arxiv.org/pdf/2005.14165.pdf`
In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20). Curran Associates Inc., Red Hook, NY, USA, Article 159, 1877–1901.

[4] Duží, M., Materna, P.: *TIL jako procedurální logika, (průvodce zvídavého čtenáře Transparentní intensionální logikou)*
Aleph Bratislava 2012, ISBN 978-80-89491-08-7

[5] Duží, M.: *Logika pro informatiky (a příbuzné obory)*
VŠB-TU Ostrava (2012), ISBN 978-80-248-2662-2

[6] Raclavský, J.: *On Partiality and Tichý's Transparent Intensional Logic*
Dostupné z: `https://www.phil.muni.cz/~raclavsky/texty/partiality_til.pdf`
Magyar Filozofiai Szemle 54 (4): 120-128.

[7] Ciprich, N., Duží, M., Košinár, M.: *TIL-Script: Functional Programming Based on Transparent Intensional Logic*
RASLAN 2007, Sojka, P., Horák, A., (Eds.), Masaryk University Brno, 2007, pp. 37–42.

[8] Vyletělek, P.: *Využití Transparentní intensionální logiky pro podporu realizace inferenčního stroje v multiagentovém systému*
VŠB-TU Ostrava (2009)

[9] Parr, T.: *ANTLR v4* [online, cit. 2022-06-09]
Dostupné z: `https://www.antlr.org`
Terence Parr

[10] Kotlin: *Kotlin Programming Language* [online, cit. 2022-06-09]
Dostupné z: `https://kotlinlang.org`
Jetbrains, s.r.o.

[11] Gradle: *Gradle Build Tool* [online, cit. 2022-06-09]
Dostupné z: `https://gradle.org`
Gradle Enterprise

[12] Turing, A. M.: *On Computable Numbers, With Application To Entscheidungsproblem*
Dostupné z: `https://doi.org/10.1112/plms/s2-42.1.230`
Proceedings of the London Mathematical Society, s2-42: 230-265.

[13] John McCarthy. 1960. *Recursive functions of symbolic expressions and their computation by machine, Part I.*
Dostupné z: `https://doi.org/10.1145/367177.367199`
Commun. ACM 3, 4 (April 1960), 184–195.

[14] Barendregt, H., Barendsen, E.: *Introduction to Lambda Calculus*
Dostupné z: `https://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf`
Chalmers University of Technology, March 2000

Příloha A

Ukázky zdrojových kódů

```
Import "class://org.fpeterek.tilscript.math.Registrar".

Karel, Vaclav, Adela/Indiv.

Defn BestStudent(w: World) -> (Indiv Time) :=
    [\t: Time -> Indiv
        ['Cond
            ['< ['TimeToInt t] '150] 'Vaclav
            ['< ['TimeToInt t] '300] 'Karel
            ['< ['TimeToInt t] '450] 'Vaclav
            'True 'Adela]].

Defn IsComputing150(i: Indiv, c: Construction) -> Bool :=
    ['Or
        ['And ['= i 'Adela] ['= c '['Ln '14.0]]]
        ['And ['= i 'Karel] ['= c '['* '4 '6]]]].

Defn IsComputing300(i: Indiv, c: Construction) -> Bool :=
    ['Or
        ['And ['= i 'Adela] ['= c '['Cos 'Pi]]]
        ['And ['= i 'Karel] ['= c '['/ '18 '6]]]].

Defn IsComputingRest(i: Indiv, c: Construction) -> Bool :=
    'False.

Defn IsComputing(w: World) -> ((Bool Indiv Construction) Time) :=
```

```
    [\t: Time -> (Bool Indiv Construction)
        ['Cond
            ['< ['TimeToInt t] '150] 'IsComputing150
            ['< ['TimeToInt t] '300] 'IsComputing300
            'True 'IsComputingRest]].
```

Ukázka A.1: Ukázka definice intenzí.

```
package org.fpeterek.tilscript.math

import org.fpeterek.tilscript.common.interpreterinterface.SymbolRegistrar
import org.fpeterek.tilscript.common.sentence.TilFunction
import org.fpeterek.tilscript.common.types.StructType
import org.fpeterek.tilscript.common.types.TypeAlias
import org.fpeterek.tilscript.javamath.InvSqrt

class Registrar : SymbolRegistrar {

    override val functions
        get() = listOf(
            Sin, Asin, Cos, Acos,
            Tan, Ln, Log, Log2,
            Log10, Round, Sqrt, InvSqrt()
        )

    override val aliases get() = emptyList<TypeAlias>()
    override val symbols get() = listOf(Pi, E)
    override val functionDeclarations get() = listOf<TilFunction>()
    override val structs get() = emptyList<StructType>()
    override val variables get() = listOf(e, pi)
}
```

Ukázka A.2: Ukázka implementace registrátoru v jazyce Kotlin

```
package org.fpeterek.tilscript.math

import org.fpeterek.tilscript.common.SrcPosition
import org.fpeterek.tilscript.common.interpreterinterface.DefaultFunction
import org.fpeterek.tilscript.common.interpreterinterface.FnCallContext
```

```
import org.fpeterek.tilscript.common.interpreterinterface.InterpreterInterface
import org.fpeterek.tilscript.common.sentence.*
import org.fpeterek.tilscript.common.types.Primitives
import kotlin.math.*

object Sin : DefaultFunction(
    "Sin",
    Primitives.Real,
    listOf(
        Variable("x", SrcPosition(-1, -1), Primitives.Real)
    )
) {

    override fun apply(
        interpreter: InterpreterInterface,
        args: List<Construction>,
        ctx: FnCallContext
    ): Construction {

        val x = args[0]

        if (x is Symbol && x.value == "Pi" &&
                interpreter.typesMatch(x.constructionType, Primitives.Real)) {
            return Real(value = 0.0, srcPos = ctx.position)
        }

        if (x is Symbol) {
            return Nil(
                ctx.position,
                reason="Cannot compute the sine of a symbolic value"
            )
        }

        x as Real

        return Real(sin(x.value), ctx.position)
    }
```

```
}
```

Ukázka A.3: Ukázka implementace TIL-Script funkce v jazyce Kotlin