VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLAD MEZI VYŠŠÍMI PROGRAMOVACÍMI JAZYKY

TRANSLATION AMONG HIGHER PROGRAMMING LANGUAGES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JAN KNAPOVSKÝ**
AUTHOR

VEDOUCÍ PRÁCE **prof. RNDr. ALEXANDR MEDUNA, CSc.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

144057

Ústav:	Ústav informačních systémů (UIFS)
Student:	**Knapovský Jan**
Program:	Informační technologie
Specializace:	Informační technologie
Název:	**Překlad mezi vyššími programovacími jazyky**
Kategorie:	Překladače
Akademický rok:	2022/23

Zadání:

1. Dle instrukcí vedoucího prostudujte překladače vybraných vyšších programovacích jazyků. Zaměřte se na jejich přední části (*front ends*).
2. Dle instrukcí vedoucího prostudujte vnitřní kódy (*intermediate codes*), které tyto překladače používají.
3. Využijte poznatků získaných v 1) a 2) k návrhu překladače mezi vyššími programovacími jazyky.
4. Implementujte navržený překladač. Demonstrujte užití implemenovaného překladače pro překlad mezi vybranými jazyky, např. z PERL do JavaScript.
5. Zhodnoťte dosažené výsledky a diskutujte další možný vývoj práce.

Literatura:

Meduna, A.: *Elements of Compiler Design*. ⁋Auerbach Publications; 1st edition (December 3, 2007).

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Meduna Alexandr, prof. RNDr., CSc.**
Vedoucí ústavu:	Kolář Dušan, doc. Dr. Ing.
Datum zadání:	1.11.2022
Termín pro odevzdání:	10.5.2023
Datum schválení:	26.10.2022

Abstrakt

S vývojem nových technologií, jazyků a jejich prostředků vyvstává čím dál tím urgentnější potřeba aktualizace již existujících programových základen, za účelem využití nových jazykových prostředků a technologií a zachováním udržitelnosti těchto systémů. Tato práce navrhuje tento proces automatizovat pomocí použití automatizovaného prostředku – transpilátoru.

Abstract

With the development of new technologies, programming languages and their devices the need arises for existing codebases to be updated and upgraded to use such devices and technologies, to preserve the maintainability and sustainability of such systems. This thesis proposes the use of automatised means to aid such efforts – a transpiler.

Klíčová slova

Transpilátor, Kompilátor, Interpret, Programovací jazyk, Syntaktická analýza, Statická analýza, Abstraktní syntaktický strom, Algoritmus, Algoritmus stoupání precedencí, JavaScript, Perl

Keywords

Transpiler, Compiler, Interpreter, Programming language, Syntax analysis, Static analysis, Abstract syntax tree, Algorithm, Precedence-climbing algroithm, JavaScript, Perl

Citace

KNAPOVSKÝ, Jan. *Překlad mezi vyššími programovacími jazyky.* Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. RNDr. Alexandr Meduna, CSc.

Překlad mezi vyššími programovacími jazyky

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením prof. RNDr. Alexandra Meduny CSc.. Další informace mi pak poskytl ing. Martin Šárfy. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jan Knapovský
5. května 2023

Poděkování

Děkuji panu profesoru Alexandru Medunovi, za jeho vstřícnost a ochotu při vedení této práce. Taktéž děkuji pánům Martinu Šárfymu a Romanu Bulgurisovi, že mi umožnili pracovat na této práci a za veškerou podporu, kterou mi v mém snažení poskytli. V neposlední řadě bych chtěl poděkovat paní Evě Míčkové, své babičce, za neskonalou lásku a morální podporu v době, kdy jsem si nevěděl rady.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Tato práce prozkoumává možnosti překladu zdrojového kódu mezi vybranými vyššími programovacími jazyky, za účelem automatizace přepisu již existujících programů a aktualizace již existující programové základny.

Přestože se většina aktivně používaných jazyků stále vyvíjí, vznikají novější jazyky, jež se poučují z vlastností a neduhů programovacích jazyků existujících. Tyto nověji vznikající jazyky poté krom úprav neblahých vlastností svých předchůdců, či úprav konceptů a jazykových prostředků, přináší často i modernější přístupy k řešení problémů. I když tyto vlastnosti jsou často následně také reflektovány v jazycích předcházejících, ať už pomocí knihovny, nebo pomocí rozšíření a aktualizace samotného jazyka, zpravidla nedosahují stejné efektivity, rozšiřitelnosti, rozšířenosti a rychlosti, jako jsou schopny dosáhnout jazyky vytvářené s těmito novými koncepty na mysli.

Dalším důvodem pro snahu o posun stávajících programových základen směrem k jejich reimplementaci v novějších a modernějších jazycích jsou také často negativní vlastnosti pramenící ze vzniku samotných jazyků. V dnešní době moderní jazyky vznikají zpravidla nejprve jako formální specifikace jazyka, s následnou implementací jeho překladu nebo interpretace, která se snaží dodržet danou specifikaci, příkladem může být specifikace *ECMA-262* jazyka JavaScript. Starší jazyky, nazvěme je jazyky *empirickými*, vznikaly často obráceným způsobem – jakožto popis fungování daného nástroje, jež používal určitý jazyk jako předpis pro svoji operaci. Dobrými příklady takovéhoto vývoje jsou třeba různé formy *shellů*, jako jsou *Bash* nebo *Fish*, nebo také v této práci popisovaný jazyk Perl.

Odhlédneme-li od nevýhod či výhod samotných jazyků, za považenou také stojí kvalifikovanost a zkušenosti programátorů s daným jazykem. Existuje-li novější jazyk, jež danou sadu úloh, pro kterou byl daný empirický jazyk v konkrétním případě využíván, zvládá efektivněji, přehledněji či spolehlivěji, lze předpokládat, že méně zkušení programátoři budou spíše seznámeni s fungováním onoho novějšího jazyka. Tato skutečnost má potenciál způsobit zastarání programové základny do takové míry, že nebude dostupný nikdo, kdo by takovou programovou základnu byl schopný efektivně udržovat, což ji eventuelně učiní nepoužitelnou.

Stejně tak s nástupem nových technologií, se dá předpokládat, že podpora těchto technologií bude dostupná v novějších programovacích jazycích dříve, ať už díky jejich popularitě a tudíž i komunitě, či díky samotnému výrobci potažmo tvůrci dané technologie, který sám zpřístupní nástroje pro její použití v moderních programovacích jazycích za účelem jejího rychlejšího rozšíření do praktického použití.

Možných motivací pro implementaci takového transpilátoru mez vyššími programovacími jazyky je tedy vícero. Od aspektů pohodlnosti vývoje, dostupnosti vývojových nástrojů či inherentní bezpečnosti samotného jazkya. Přes kompatibilitu s nově příchozími systémy a technologiemi. Po aspekty čistě personální a finanční, aby nedošlo k zastarání programové základny po možném ukončení spolupráce s potenciálně těžko nahraditelnými klíčovými osobami.

Jelikož mnoho společností v dnešní době stojí na základech tvořených z velké části IT infrastrukturou, je v jejich zájmu, aby tato infrastruktura byla a zůstala v průběhu let udržovatelná, rozšiřovatelná a stabilní. Zastarání této infrastruktury by totiž mohlo vést na takřka astronomické náklady ať už na její provoz, z důvodu potřeby extrémně kvalifikované pracovní síly, potřebné pro údržbu daného systému, nebo čistě náklady plynoucí z neefektivního provozu takové infrastruktury, nebo náklady vzniklé z nutnosti zpětného znovuvytvoření, reimplementace a znovunasazení takové infrastruktury, měla-li by se stávající implementace stát příliš obtížnou na údržbu a provoz.

Reimplementace existující programové základny je ovšem také vysoce kvalifikovaná činnost. Vyžaduje totiž nejenom důvěrnou znalost původního jazyka a jazyka ve kterém má k reimplementaci dojít, nýbrž i detailní znalost daného systému jako celku, jeho fungování a způsobu jeho implementace. To značně omezuje množinu osob schopných efektivně provést takovouto reimplementaci na zpravidla programátory zkušené přímo s daným software. Jelikož tyto osoby však bývají klíčové pro každodenní chod takovýchto systémů, je jsou jejich možnosti provést celkovou reimplementaci značně omezeny.

Tato práce se tedy zabývá možností využití automatizované reimplementace, chceteli překladu, pomocí transpilátoru dané dvojce vyšších programovacích jazyků, za účelem urychlení a zjednodušení tohoto procesu.

Podcílem této práce je navrhnout takovou implementaci, která by umožnila co nejfunkčnější překlad a tím pádem co nejrychlejší implementaci a integraci původní programové základny se základnou nově vznikající v cílovém jazyce.

Tato práce nejprve analyzuje dvojici empirického jazyka, konkrétně jazyka Perl, a jazyka moderního, konkrétně se jedná o jazyk JavaScript. Porovnává jejich vlastnosti a navrhuje řešení pro transpilaci jazyka vstupního do jazyka výstupního, kterou následně implementuje a používá k představení problémů pramenících jak ze samotných nedokonalostí překladu a omezení možností analýzy vstupního jazyka, tak z rozdílů jazyka vstupního a výstupního. Zabývá se také vlastnostmi generovaného kódu z pohledu udržovatelnosti, čitelnosti a optimálnosti.

Závěrem pak polemizuje jak nad užitečností, efektivitou a použitelností takto narvženého a implementovatelného transpilátoru, tak nad potřebnými náklady pro vytvoření takového transpilátoru. Zvažuje i další možnosti, které by mohly být úspěšné či úspěšnější než v této práci navržený přístup.

Kapitola 2

Definice základních pojmů

Informace v této kapitole byly přejaty z [2], [7], [8], [6] a [9], není-li uvedeno jinak.

2.1 Kompilátor

Transpilátory a kompilátory jsou obecně považovány za podmnožinu systémového software [9]. Jedná se o software, který narozdíl od aplikačního software, poksytujícího služby přímo uživateli, nekomunikuje přímo s uživatelem, nýbrž umožňují fungování a běh jiných programů a aplikací.

Kompilátorem pak rozumíme program, který je navržen za účelem převodu jednoho jazyka, zpravidla nazývaného jazykem zdrojovým, do jazyka druhého, zpravidla zvaného cílový, přičemž jakékoliv chyby v průběhu převodu – překladu – ohlásí. U kompilátorů se předpokládá, že cílový jazyk bude jazykem nižším, zpravidla bývá jazykem symbolických adres. Výsledný program, zapsaný v jazyce symbolických adres, je následně za pomoci assembleru, programu pro převod jazyka symbolických adres do binární podoby strojového kódu, přeložen přímo pro danou platformu.

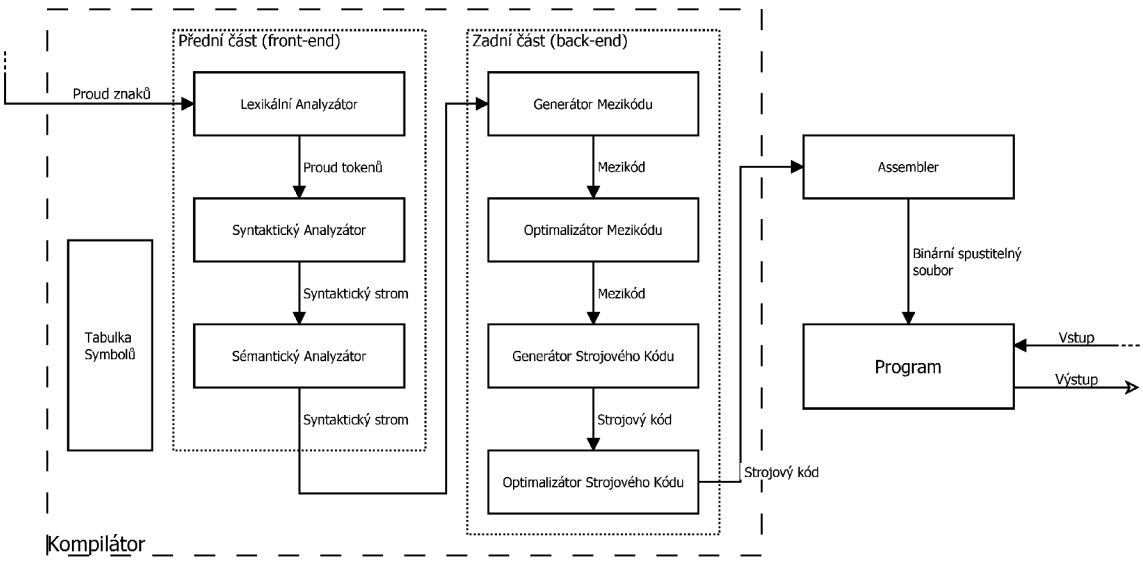
Výhodou kompilátorů je ten fakt, že výsledný program v binární podobě bývá optimalizovaný pro běh přímo na hardwaru a tudíž mohou využít jeho plný potenciál, čímž pádem jsou o mono rychlejší.

2.1.1 Princip fungování kompilátoru

Kompilace je proces skládající se z vícero kroků (viz obrázek č. 2.1). Na vstupu kompilátoru se nachází zdrojový kód programu ve formě proudu znaků. Tento zdrojový kód je předmětem překladu a bude postupně analyzován jednotlivými analyzátory.

Prvním analyzátorem, začínající přední část kompilátoru, je analyzátor lexikální. Ten má za úkol rozdělit spojitý vstupní proud znaků na skupiny znaků s určitým (lexikálním) významem – *lexémy*. Lexémy jsou tedy nejmenší jednotkou vstupního jazyka. Lexikální analyzátor tyto lexémy kontroluje a kategorizuje. Úspěšně přečtený lexém následně předává syntaktickému analyzátoru ke zpracování ve formě *tokenu*, který se zpravidla skládá z daného lexému, typu onoho lexému a dalších informací o daném lexému, například pokud se jedná o symbol, i odkazu do tabulky symbolů. Takovéto tokeny tedy postoupí syntaktickému analyzátoru.

Druhou fází procesu kompilace je syntaktická analýza. Syntaktický analyzátor, často také zvaný parser, používá tokeny postoupené lexikálním analyzátorem k vytvoření stromové struktury, reprezentující gramatickou strukturu vstupního proudu tokenů. Takto vy-



Obrázek 2.1: Diagram částí kompilátoru a jeho okolí

tvořený *abstraktní syntaktický strom* (AST, z anglického *abstract syntax tree*), kde uzly reprezentují operace, jejich potomci pak reprezentují jejich operandy či argumenty a uzly listové reprezentují konkrétní hodnoty či symboly nebo jiné zdroje hodnot.

Třetí fází, a poslední fází přední části kompilátoru, je analýza sémantická. Sémantický analyzátor, ve spojení s tabulkou symbolů, kontroluje sémantickou správnost zdrojového kódu z pohledu definice jazyka a jeho operátorů. Také sám plní tabulku symbolů nalezenými symboly a informacemi o nich. Jeho nejdůležitější úlohou je však typová kontrola, tzn. kontroluje, zda každý operátor má operandy správného typu, pokud ne, může buď nařídit vygenerování kódu pro převedení hodnoty za běhu, pokud to kombinace daného a vyžadovaného datového typu umožňuje (například převod celého čísla na číslo s plovoucí desetinnou čárkou), nebo ohlásit chybu. Takovýto implicitní převod datového typu se často nazýva „donucení“ či „vynucení“ typové konverze, z anglického *coercion*. Toto však také znamená, že datové typy používaných proměnných či návratových hodnot funkcí musí být známé či odvoditelné již během překladu a za běhu programu se nesmí měnit, jelikož sémantická analýza probíhá pouze jednou, a to právě při překladu.

Následující, a první částí takzvané zadní části kompilátoru, je generátor mezikódu. Není sice podmínkou překladu, ale přesto je často používán pro optimalizaci překládaného programu. Tento generátor prochází AST a na jeho základě generuje mezikód. Tento mezikód může mít mnoho podob. Mezi příklady, které stojí za zmínku patři například zásobníkový kód, ukládající postupně operandy na zásobník, s tím, že operátory následně pracují implicitně se zásobníkem tak, že nejprve vysunou sobě potřebný počet operandů ze zásobníku a následně nasunou svůj výsledek zpět. Jinou možností je například tříadresný kód, který dostal svůj název podle své struktury – každá instrukce má jednu adresu destinace, a jednu až dvě adresy operandů, dle potřeby. Takto vygenerovaný, na platformě nezávislý, mezikód bude předán následujícímu bloku.

Následujícím blokem je pak, opět volitelná, optimalizace. Tříadresný i zásobníkový kód umožňují jednoduchou implementaci mnoha optimalizačních technik, jako je optimalizace mrtvých proměnných zdola nahoru, která redukuje zbytečné výpočty nikdy nevyužitých hodnot, nebo optimalizace seskládání konstant, kdy výrazy vypočítatelné již při překladu

nebudou zahrnuty do výsledného programu, nýbrž budou nahrazeny svou konstantní hodnotou. Takto optimalizovaný kód může být následně předán generátoru kódu cílového.

Generátor cílového kódu, často také zvaný generátor kódu strojového, je takřka jedinou povinnou součástí zadní části kompilátoru. Jeho úlohou je na základě svého vstupu, ať už se jedná o AST v nepřítomnosti mezikódu, či samotný mezikód, generovat kód určený přímo pro cílovou platformu, která bude následně hostovat daný program. Výsledný, pro platformu specifický kód může být buď výsledkem celé kompilace, nebo ještě projít poslední vlnou optimalizací.

V poslední, taktéž volitelné části kompilačního procesu může dojít k optimalizacím specifickým pro danou platformu. Kde předchozí vlna optimalizací na platformě nezávyslého kódu je zpravidla společná pro všechny platformy podporované daným kompilátorem, tato část optimalizací je mířena přímo pro danou platformu. V této fázi je možno dělat optimalizace které vyžaduje přímo daná platforma, jako je reorganizace operací za účelem snížení potřebné paměti, či optimalizace použití registrů u registrových architektur platforem za účelem efektivnější práce s přístupy do paměti.

2.2 Interpret

Druhým častým typem jazykových procesorů je interpret. Ten, místo toho aby produkoval nativní kód pro cílovou platformu, sám na cílové platformě běží a na základě vstupního zdrojového kódu programu sám tento program vykonává, mapujíc vstupy daného programu na jeho výstupy. Toto mu dává odlišné vlastnosti. Jelikož výsledný program neběží přímo na cílové platformě, nemůžeme logicky očekávat stejnou efektivitu, ovšem výměnou za to můžeme získat větší flexibilitu či kompatibilní rozhraní nezávislé na platformě.

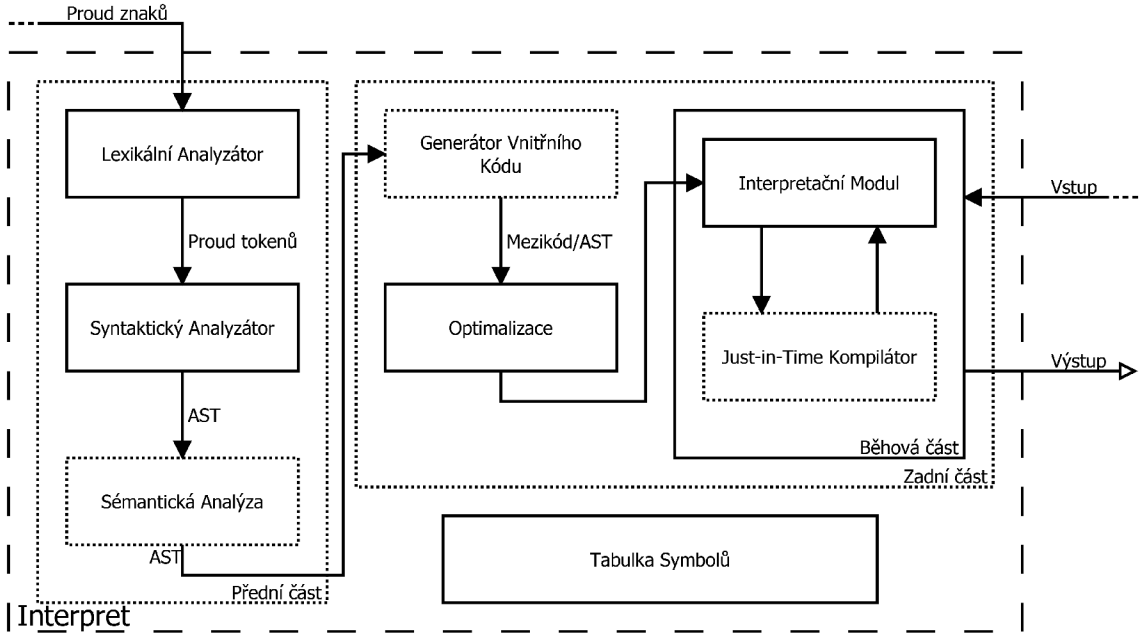
2.2.1 Princip fungování interpretu

Interpret sdílí nemálo principů s kompilátory. Jejich úloha je velmi podobná kompilátorům, což dcokazuje i značné množství sdílené funkcionality s nimi, jak je patrné z obrázku č. 2.2.

Přední část interprety takřka sdílí s kompilátory. Tato část je závislá na překládaném jazyce, nikoliv na cíli překladu, ať už je jím interpret, virtuální stroj, či přímo cílová platforma. Zadní část je naopak velmi rozdílná od kompilátorů. Nezabývá se totiž generováním kódu pro hostující či jinou cílovou platformu. Tato část je sama spuštěna na hostující platformě a na základě svého vstupu, kterým může být přímý výstup ze syntaktického analyzátoru (AST), či mezikód.

Sémantická analýza může proběhnout před spouštěním programu, avšak může být odložena až do času běhu. Takovéto chování může značně zvýšit flexibilitu oproti kompilátorům, umožňujíc kupříkladu dynamické typování proměnných. Toto však může vést k chybám ve vstupním programu, které nebudou odhaleny dříve než za běhu. Možná je i kombinace obou přístupů, jako u jazyka Perl, který nejprve provede statickou sémantickou analýzu, na základě zjištěných informací generuje mezikód, jej optimalizuje, za běhu poté probíhá sémantická analýza pouze těch částí, které nebylo možné analyzovat při překladu do mezikódu.

Implementace běhových částí se mnohou značně lišit. Rekurzivní implementace interpretu používá přímo AST k provádění vstupního programu, implementujíc rutiny pro jednotlivé operace. Takovýto interpret poté prochází AST shora dolů, pro každý uzel volajíc rutinu,



Obrázek 2.2: Ukázkový diagram částí interpretu

která vyhodnotí daný uzel. Tyto rutiny následně rekurzivně volají rutiny pro vyhodnocení podstromů operací k jejichž vyhodnocení byly použity.

Iterativní interprety se více podobají architektuře dnešních CPU, skládají se zpravidla z jedné smyčky, která iterativně provádí buď instrukce mezikódu, nebo méně často přímo uzly AST v pořadí průchodu do hloubky zleva do prava, udržujíc si ukazatel na aktuálně zpracovávanou instrukci/uzel. Po dokončení vykonávání aktuální instrukce dojde ke změně tohoto ukazatele tak, aby ukazoval na následující instrukci/uzel. Informace či přímo ukazatel na následující instrukci/uzel často bývá součástí aktuální instrukce, čímž tvoří řetězec instrukcí. Tato informace či hodnota ukazatele samozřejmě může být vypočítávána dynamicky, čímž umožní podmíněné skoky či volání funkcí a rutin. Takovýto interpret může k vykonávání svého vstupního používat různé architektury, jako je například architektura registrová, takový interpret bude mít v rámci svých vnitřních struktur virtuální registry, argumenty operátorů bude dle instrukcí umisťovat do těchto registrů, zatímco operátory budou používat hodnot v těchto registrech jako argumentů a své výsledky budou umisťovat zpět.

Daší možnou klasifikací běhových částí interpretu je podle formátu mezikódu neboli instrukcí mezikódu, které daný interpret zpracovává. Paul Klint například ve svém článku *Interpretation techniques* rozlišuje interprety na tři hlavní typy [8]:

1. Klasický interpret
2. Interpret s přímo provázaným kódem
3. Interpret s nepřímo provázaným kódem

Klasický interpret používá instrucke formátem podobné objektovému kódu, kdy způsob či rutina obsluhující danou instrukci je dána záznamem v tabulce operací pod klíčem shodným s kódem operace dané instrukce.

Instrukce pro interpret s přímo provázaným kódem nenesou kódy operace, nýbrž přímo ukazatele na rutiny obsluhující danou operaci. Interpret následně pouze volá tyto rutiny z adresy v paměti s argumenty z aktuální instrukce.

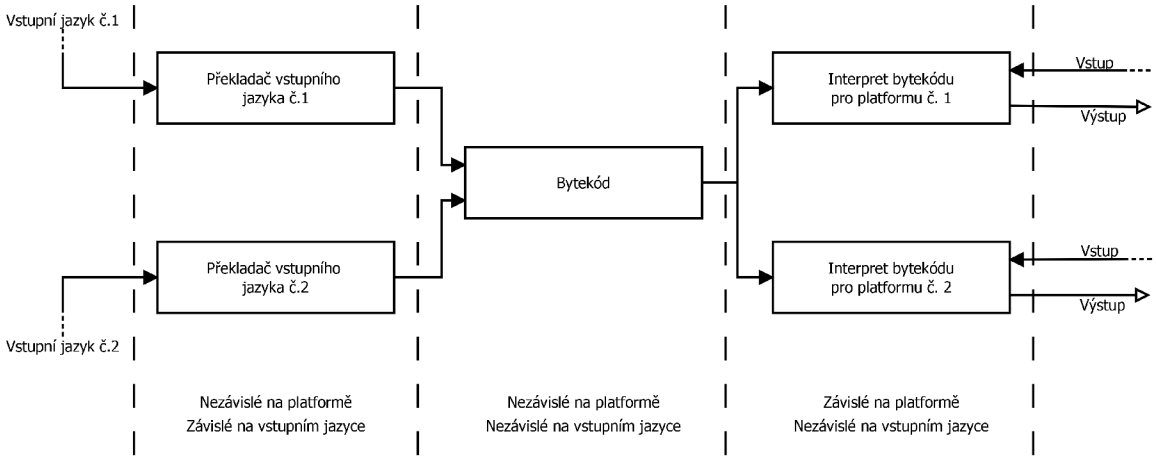
Interpret s nepřímo provázaným kódem potom podle Klintovy kategorizace tyto přístupy spojuje. Každá instrukce takového interpretu odkazje svým kódem operace do tabulky operací, tam se však nenachází vstupní bod rutiny obsluhující danou instrukci, nýbrž odkaz na něj. Výhodou tohoto přístupu je, že oproti klasickému interpretu se statickou tabulkou kódů instrukcí, tabulka tohoto typu interpretu může být dynamicky upravována za běhu programu zaměňováním ukazatelů na obslužné funkce.

Implementace interpretů mohou pro zrychlení spouštění programů implementovat takzvaný just-in-time (JIT) kompilátor. Jeho úkolem je kompilovat často používaný kód, který by byl za normálních okolností spouštěn běhovou částí interpretu, do kódu nativního hostitelské platformě a tak umožňuje rychlejší běh programu.

2.3 Hybridní překladač

Některé jazyky kombinují tyto přístupy dohromady. Zdrojový jazyk je nejprve přeložen překladačem, nebo chcete-li kompilátorem, do mezikódu, často také zvaného bytekód. Tento bytekód je uložen jakožto výstup překladače. Pro spuštění takto přeloženého programu je posléze využit interpret.

Toto umožňuje implementovat překladače různých jazyků do stejného bytekódu, což může umožnit interoperabilitu mezi těmito jazyky, zatímco na platformě nezávislý bytekód zaručuje přenositelnost mezi platformami. Jediné, co je potřeba pro přenesení takto přeloženého programu, je implementace interpretu takového bytekódu a základní knihovny pro komunikaci s hostujícím systémem na cílové platformě. Mezi nejznámější využití této techniky patří platforma *Java Virtual Machine*.



Obrázek 2.3: Diagram hybridní implementace překladače při použití s rozdílnými jazyky na rozdílných platformách

2.4 Transpilátor

Třetím z typů programu zpracovávajících programovací jazyky je transpilátor. Transpilátor je v mnohém podobný kompilátoru. Stejně jako kompilátory či interprety, i transpilátory implementují přední část velmi podobně. Shodně také jako vstup přijímají zdrojový kód programu ve zdrojovém, vyšším, programovacím jazyce. Narozdíl však od jejich již zmíněných protějšků tento kód ani sami neprovádějí, ani netransformují do jazyka symbolických instrukcí či jiného nižšího jazyka. Jejich cílovým, stejně jako zdrojovým, jazykem jsou jazyky vyšší.

Transpilátory je možno využít například pro implementaci nástaveb a aktualizací jazyka v prostředí, kde není možno kontrolovat či aktualizovat interpret nebo kompilátor, který bude daný program spouštět či kompilovat. Takovéto použití můžeme vidět například u nástorje *Babel.js*[1]. Tento nástroj je určený pro převod programů psaných pro moderní verze jazyka JavaScript, standardu *ECMAScript2015* a novějších, do podoby kompatibilní se staršími prohlížeči, které neimplementují novější standardy. Toto umožňuje použití nových vlastností jazyka JavaScript ve webových aplikacích, aniž by bylo nutno čekat na podporu ze strany prohlížečů, či se obávat nefunkčnosti stránek na zastaralých či pozdě aktualizovaných prohlížečích.

Dalším příkladem využití transpilátoru je nástavba taktéž jazyka JavaScript od společnosti Microsoft – TypeScript[2]. Ta rozšiřuje vlastnosti jazyka JavaScript o statickou typovou kontrolu a poskytuje tak prostředky pro zajištění vyšší typové bezpečnosti v jinak velice typově dynamickém jazyce JavaScript, který se používá nejen v prohlížečích. Programy psané v rozšíření TypeScript jsou pomocí transpilátoru nejprve kontrolovány, zda je dodržena typová bezpečnost, a následně jsou překládány do jazyka JavaScript. Toto umožňuje použití jazyka/rozšíření TypeScript všude, kde je jinak možno použít jazyk JavaScript, aniž by vznikla nutnost jakkoliv rozšiřovat či modifikovat existující interprety.

[1] babeljs.io
[2] typescriptlang.org/docs

Kapitola 3

Popis vybraného zdrojového jazyka

Tato kapitola stručně popisuje vybraný zdrojový jazyk Perl, a jeho interpret `perl`.

3.1 Popis programovacího jazyka Perl

Perl, konkrétně jeho pátá verze Perl5, se obecně považuje za jeden z velmi rozšířených skriptovacích jazyků. Dle dat zveřejněných v článku *Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects* má dvanácté nejvyšší zastoupení, co se týče počtu řádek zdrojového kódu, v projektech s otevřeným zdrojovým kódem na službě GitHub.[3]

Jedná se o vysokoúrovňový, všeobecný, dynamický, interpretovaný programovací jazyk. Perl podporuje mnoho paradigmat programování, od imperativního, přes objektově orientované, tak, s určitou knihovní podporou, i funkcionální programování[10]. Nedílnou součástí jazyka Perl je také velmi komplexní a široce využívaná vestavěná podpora regulárních výrazů, která ovlivňovala nejen vývoj jazyka Perl. Perl také obsahuje rozsáhlou podporu meziprocesové komunikace ať už prostřednictvím socketů velmi podobných BSD socketům, nebo pomocí UNIXových rour.

Jazyk Perl je také velmi rozšiřitelný a to ne jenom díky podpoře modularity a knihoven. Pro jazyk Perl byl komunitou vytvořen komprehenzivní repozitář veřejně přístupných skriptů a knihoven zvaný CPAN[1], alias *Comprehensive Perl Archive Network* vytvořený v roce 1995. Tento repozitář je v současnosti spravován komunitou a čítá přes dvěstě tisíc modulů. Umožňuje tak zjednodušení programování a zrychlení vývoje aplikací a skriptů, ať už pomocí předpřipravených programovacích rozhraní pro ovládání periferií, databází či webových API, nebo pomocí přímých rozšíření samotného jazyka či interpretu.

Samotný interpret jazyka Perl, zvaný `perl`, je sám velmi rozšiřitelný díky svému API pro jazyk C, ve kterém je sám napsán. Pro psaní rozšíření interpretu je k dispozici rozhraní a vývojové nástroje XS pro vývoj rozšíření Perl interpretu v jazyce C[14]. Takováto rozšíření mohou například zpřístupnit v prostředí jazyka Perl externí struktury a rozhraní dalších programů, jako jsou například nástorje pro objektově relační mapování přímo nad relačními databázemi, umožňující takřka přímé mapování databázových struktur na objekty jazyka Perl.

Perl původně vznikl jako nástroj pro dynamické zpracování arbitrárních textových či tabulkových dat, nikoliv jako obecný programovací jazyk. Tato skutečnost je místy velmi patrná, kdy se hranice mezi jazykem Perl a interpretem jazyka Perl ztrácí. Narozdíl na-

[1] cpan.org

příklad od jazyka JavaScript, který je formálně definován, a jeho interpretů, které se snaží tuto definici dodržet, by se dalo říct, že jazyk Perl je definován fungováním programu `perl` – interpretu jazyka Perl. Tento interpret je stále vyvíjen komunitou jazyka Perl a obsahuje značné množství volitelných či experimentálních funkcionalit, avšak jelikož se jedná o volitelná rozšíření samotného interpretu, která nemají dlouhou historii používání, nebudeme je pro účely této práce považovat za součást jádra jazyka Perl.

3.2 Základní vlastnosti syntax a sémantiky

Tato sekce shrnuje základní informace o lexikálních, sémantických a syntaktických prostředcích jazyka Perl. Ve zkratce se snaží přiblížit jeho nejzákladnější a nejpoužívanější prostředky. Informace jsou přejaty z [18], [4] a [14].

3.2.1 Proměnné a datové struktury

Proměnné v jazyce Perl mohou mít jeden ze tří vestavěných datových typů: skaláry, pole skalárních hodnot nebo asociativní pole skalárních hodnot, také známé jako „hash tabulky“. Skaláry jsou jednoduché hodnoty, ať už řetězec libovolné délky, omezený pouze dostupnou pamětí, číslo, nebo odkaz. Pole jsou uspořádané seznamy skalár indexované od nuly. Asociativní pole jsou pak neuspořádané kolekce skalár indexované pomocí asociovaných klíčů ve formě řetězce.

K získání hodnoty proměnné se běžně používá její jméno, nebo pojmenovaná reference. Prvním znakem takového jména je vždy jeden ze znaků `$`, `@` nebo `%` pro skalární hodnotu, pole či asociativní pole respektive.

Proměnné jsou v jazyce Perl rozlišovány nejen pomocí jejich jména, nýbž pomocí kombinace jména a rezolučního typu. Může tedy existovat zdánlivě více proměnných se stejným jménem ale různým typem, takže termy `$proměnná`, `$proměnná[0]` a `@proměnná` mohou, a s největší pravděpodobností budou, mít různé hodnoty. Interpret jazyka Perl pak tyto uchovává v tabulce symbolů pod stejným klíčem, jménem proměnné. Jednotlivé hodnoty jsou poté uchovávány v subtabulce rozdělené podle typu, jak je znázorněno na obrázku č. 3.1. Perl také podporuje speciální lexikální prostředek pro přímý přístup k záznamům v tabulce symbolů, pomocí takzvaných typových jmen (*typeglob*). Jedná se o přímé odkazy do tabulky symbolů. Použijeme-li proměnnou `$jméno1`, odkazujeme se na skalární hodnotu, která byla zařazena do tabulky typů, na níž odkazuje záznam `jméno1` v tabulce symbolů. Stejně tak se můžeme odkázat na celou tabulku typů tohoto jména pomocí termu `*jméno1`. Tímto získáme odkaz na všechny typy dat, uložené pod daným jménem. S tímto odkazem můžeme následně dále pracovat, jako například jej celý, nebo jenom některé jeho části, přiřadit – zkopírovat – pod jméno jiné (na obrázku `jméno2`). Tímto způsobem získáme alias pro dané jméno. Veškeré hodnoty uložené pod jménem prvním budou do konce aktuálního jmenného vazebného prostoru dostupné i pod jménem druhým. Nejedná se však o referenci ani kopii dat. Referencí se v Perlu myslí hodnota skalárního typu odkazující na hodnotu jinou, viz obrázek č 3.2.

Správu paměti řeší interpret programovacího jazyka Perl sám. To ovšem také implikuje přítomnost dealokátoru (garbage collector). Každá proměnná pak má počítadlo referencí. Opustí-li vykonávání programu jmenný vazebný prostor, v němž je proměnná definována, a neexistují-li už žádné další reference na její hodnotu (tzn. její počítadlo referencí dosáhlo počtu 0), bude paměť použita pro tuto proměnnou uvolněna ke znovuvyužití a její hodnota bude ztracena. Stejný přístup pak může platit i pro samotný kód. Byla-li při překladu

definována funkce ale spouštěním programu došlo k opuštění jmenného vazebného prostoru dané funkce, může být paměť použitá k uložení mezikódu, odpovídajícího dané funkci, uvolněna pro další použití, neexistuje-li jiná reference na tuto funkci.

Kontext

Výklad hodnoty proměnné často závisí na kontextu, tj. jaký typ hodnoty je očekáván nadřazenou operací. Nejčastějšími kontexty jsou kontext skalární a kontext pole. Operace mohou vracet odlišné hodnoty odlišného typu na základě kontextu, v jakém byly použity. I samotné vyvolání hodnoty proměnné může vrátit odlišnou hodnotu v závislosti na kontextu, v jakém bylo invokováno.

Stejným způsobem může každá operace uvrhnout všechny své operandy do nějakého kontextu, nebo pouze propagovat kontext nadřazený, a tím ovlivňovat výsledek vyhodnocení.

Operace přiřazení určuje kontext své pravé části dle své části levé. To jest, pokud je přiřazováno do skalární proměnné, bude pravá strana výrazu vyhodnocena ve skalárním kontextu, pokud je však přiřazováno do proměnné typu pole či asociativní pole, bude pravá strana výrazu vyhodnocena v kontextu pole.

Kromě již zmíněného skalárního kontextu a kontextu pole, disponuje Perl ještě dvěma kontexty, kterými jsou kontext nulový (*void*) a kontext booleovský. Nulový kontext se chová jako skalární, pouze s tím rozdílem, že výsledek bude zahozen. Nejčastěji se s ním setkáme při používání operátorů (funkcí) aniž bychom výsledek kamkoliv přiřadili. Druhý z nich, booleovský, se vyskytuje tam, kde interpret zajímá pravdivostní hodnota, to znamená například v podmínkovém výrazu podmíněného bloku. Jeho chování je však takřka identické s kontextem skalárním.

Skaláry

Veškerá data v Perlu jsou buď skaláry, pole skalár, nebo asociativní pole skalár. Jak již bylo zmíněno výše, za skalární hodnotu se považuje jedna ze tří následujících hodnot: textový řetězec, číselná hodnota, nebo reference. Narozdíl od odkazů, neboli referencí, je hranice mezi textovými řetězci a číselnými hodnotami velice tenká. Zpravidla lze říci, že je-li proměnné přiřazen jako hodnota takový řetězec, který by se dal intepretovat jako číslo, ať už celé nebo reálné, v dekadickém či jiném zápisu, bude možné s takovouto proměnnou nakládat jako s proměnnou s číselnou hodnotou. Stejně tak opačným způsobem je možno nakládat s numerickou hodnotou jako s řetězcem znaků. Naopak reference jsou silně typované, bez možnosti tento typ změnit (narozdíl např. od jazyka C).

V booleovském kontextu je skalární hodnota vyhodnocena jako nepravdiá pokud je nedefinovaná, rovna prázdnému řetězci nebo je numeriky rovna nule, to znamená číslo 0 nebo jeho rětězcový ekvivalent `"0"`.

Vyhodnocení skalární hodnoty v kontextu pole vyůstí v seznam s jedním jediným prvkem, jímž je daná skalární hodnota. Skalární hodnoty lze sdružovat do seznamů pomocí závorek a operátoru `,`. Narozdíl však od pole, seznam skalárních hodnot nebude ve skalárním kontextu vyhodnocen jako svá délka, ale jako hodnota posledního prvku a všechny předchozí prvky budou vyhodnoceny v nulovém kontextu.

Reference vzniká buď jako reference na pojmenovanou proměnnou, nebo pomocí konstruktorů anonymních referencí. (Nejedná se o konstruktor objektů!) Ty konstruují anonymní reference na pole nebo asociativní pole respektive. Takováto reference vyhodnocena ve skalárním kontextu vrátí samu sebe, tzn. kopii reference na daný objekt v pa-

měti (ne nutně objekt ve smyslu objektově orientovaného programování). Takovéto reference lze následně dereferencovat například pomocí operátoru `->` nebo pomocí konstrukce `?{$reference}`, kde znak `?` je nutno nahradit za znak odpovídající typu reference. Tento znak může být buď jeden ze znaků již představených v minulé sekci, tzn. `$` pro skalární hodnotu, `@` pro pole nebo `%` pro asociativní pole, nebo se může jednat o znak `&`, značící dereferenci na blok kódu, například, a ne výlučně, na potenciálně anonymní subrutinu. Takto uložená subrutina si uchovává svůj vlastní vnitřní kontext jmenného vazebného prostoru, stejně jako již uložené proměnné. Toto umožňuje tvorbu uzávěrů funkcí.

Řetězce znaků jsou další možnou hodnotou pro proměnné skalárního typu. Možností zápisů řetězcového literálu je více. Patří mezi ně jednoduché uvozovky (`'retezec'`), ty se dají nahradit také značkou `q/retezec/`, kde znak `/` můžeme nahradit za jakýkoliv pár závorek nebo některý z povolených znaků. V takto vymezeném řetězci nedochází k interpolaci proměnných do řečtězce ani k vyhodnocení escape sekvencí. Druhou možností jsou dvojité uvozovky (`"retezec"`) a jejich obdoba `qq/retezec/`. V rámci takto vymezených literálů řetězců dochází k interpolaci proměnných do obsahu řetězce. Nachází-li se v obsahu interpolovatelného řetězce subřetězec odpovídající invokaci, dereferenci či jinému použití proměnné, bude tento subřetězec nahrazen řetězcovou reprezentací hodnoty dané proměnné. Při interpolaci pole budou jeho prvky separovány hodnotou vestavěné proměnné *oddělovače poolí* (`$"`).

Pole

Pole skalárních hodnot jsou prvním složeným datovým typem v jazyce Perl. Jak již bylo popsáno, jedná se o uspořádaný seznam skalárních hodnot indexovaný celými čísly počínaje nulou. Velikost daného pole je omezena pouze dostupnou pamětí, interpret se stará o její alokaci a uvolnění

Přiřazení do proměnné typu pole uvrhne pravou stranu daného přiřazení do kontextu pole. Nejjednodušší možností, jak inicializovat hodnoty takovéto proměnné je použití *seznamu*, elementů oddělených čárkou, uzavřených v kulatých závorkách. *Seznam* uvrhne každý jeden element do kontextu pole. Pokud bude prvek vyhodnocen jako *seznam*, budou jeho prvky vloženy na pozici onoho prvku, tzn. *seznam* může být vždy jenom jednorozměrný.

Stejně jako *seznamy*, i pole mohou být pouze jednorozměrná. Perl však podporuje vkládáni podřízených polí pomocí referencí na pole, ať už anonymních či ne. Pro subskriptování a získávání hodnot takto vytvořených multidimenzionálních polí podporuje Perl speciální syntax, která přímo neodpovídá syntaxi pro dereferenci, viz 3.1.

Hodnoty jednotlivých položek lze z pole získat pomocí subskriptu vymezeného v hranatých závorkách.

	Kód	Význam
1	`@array`	Celé pole „array“, vyhodnocení závisí na kontextu.
2	`$array[0]`	První element (skalár) pole „array“.
3	`@{$array}[0]`	První prvek pole referencovaného skalární proměnnou „array“
4	`$array->[0]`	Další varianta případu č. 3
5	`$array[0][1]`	Druhý prvek pole referencovaného prvním prvkem pole „array“

Tabulka 3.1: Možnosti subskriptování proměnných typu pole v jazyce Perl

Za povšimnutí stojí případ č. 2. Kvůli znaku `$` na začátku výrazu by se mohlo zdát že se jedná o operaci se skalární proměnnou o jméně „array“, to však není pravda, zde spíše značí očekávaný typ získané hodnoty.

V případě č. 5 je znázorněna ona syntax pro multidimenzionální pole. Za povšimnutí stojí implicitní dereference potenciálně anonymní reference na pole uložené v poli pojmenovaném „array“.

Vyskytne-li se případ č. 1 ve skalárním kontextu (nebo booleovském, který se chová obdobně), získanou hodnotou bude délka pole. Z toho přímo vyplývá pravdivostní hodnota pole, tj. neobsahuje-li žádné prvky, pak je jeho pravdivostní hodnota nepravda, jinak je pravdivá.

Přiřazení do jednotlivých prvku kterýmkoliv ze způsobu 2 až 5 vede na vyhodnocení pravé strany přiřazení ve skalárním kontextu a nahrazení nebo vytvoření prvku na daném indexu.

Asociativní pole

Asociativní pole, na rozdíl od klasického pole, je neuspořádaná kolekce skalár indexovaná pomocí řetězců znaků. Stejně jako u klasických polí, není možno do asociativního pole vkládat jiné položky než skaláry, tudíž i reference. Taktéž velmi obdobně je k dispozici syntax pro subskriptaci obdobnými způsoby jako bylo popsáno v tabulce 3.1.

I u asociativního pole záleží na kontextu ve kterém je použita forma obdobná formě č. 1 z tabulky 3.1 (tj. `%hash`). Je-li tato forma vyhodnocena v booleovském kontextu, bude její hodnota pravdivá pouze pokud dané asociativní pole obsahuje alespoň jeden pár klíče a hodnoty. V kontextu pole bude vyhodnocen jako seznam hodnot, kdy každá lichá hodnota je klíč a každá sudá hodnota je danému klíči odpovídající hodnota.

Obdobně přiřazení do takovéto formy vyústí ve vyhodnocení pravé strany v kontextu pole, každý lichý prvek bude vyhodnocen jako klíč a každý sudý prvek jako jemu odpovídající hodnota.

Pro bližší práci s asociativními poli existují vestavěné operátory, například operátor `keys`, který v kontextu pole vrátí pole klíčů v nespecifikovaném pořadí, ve skalárním kontextu vrátí pouze jejich počet.

Objekt

V Perlu neexistuje separátní datový typ objekt; objektové chování emuluje speciálně upravená reference na asociativní pole, tzv. požehnaná reference. Takováto reference nese, kromě odkazu na zpravidla anonymní asociativní pole, i odkaz na balíček – modul – psaný v jazyce Perl. Tento modul/balíček tedy můžeme nazývat třídou. Dojde-li v kódu k dereferencování požehnané reference za účelem volání metody (`$objekt->metoda()`), interpret Perlu se pokusí najít v modulu, kterým byla daná reference požehnána, stejnojmennou subrutinu. Nalezne-li takovou, předá jí, jako první argument, onu požehnanou referenci, následovanou všemi dalšími argumenty. Nenalezne-li, jedná se o chybu.

Dědičnost řeší Perl pomocí deklarace nadřazených balíčků (modulů). Pokud není daná metoda nalezena přímo v modulu, jímž bylo referenci požehnáno, pokusí se interpret najít stejnojmennou funkci v balíčku nadřazeném. Toto dovolí aby jeden objekt byl instancí třídy s třídami nadřazenými, což umožňuje polymorfismus a (i vícenásobnou) dědičnost.

Atributy jsou ukládány přímo do konkrétního referencovaného asociativního pole. Do tohoto pole mají samozřejmě přístup nejen všechny metody dané třídy i tříd nadřazených, ale i veškerý kód v jehož rámci je daný objekt používán. Programovací jazyk Perl tudíž

ve své základní podobě nerozlišuje soukromé, chráněné či privátní atributy (ne bez dalšího programatického rozšíření[2]).

Třídy v programovacím jazyce Perl neumožňují jednoduše přetěžovat operátory, ne bez použití rozšíření, které je součástí standardní distribuce, ani nemusí znát speciální metody jako jsou konstruktory, destruktory a podobně. Přesto však je několik jmen metod , které se interpret může pokusit volat v průběhu programu, jako je například metoda `DESTROY`, jež se pokusí interpret zavolat u objektu, který má být zničen pomocí automatické správy paměti (garbage collectoru), pokud taková metoda u daného objeku existuje. Přestože v Perlu existuje syntax pro instancování jeho tříd, jedná se dle dokumentace o praxi nedoporučovanou. Většinou tedy probíhá vytváření objektů ručně pomocí volání dedikované metody z modulu, jež je zároveň třídou vytvářeného objektu.

Jazyk Perl mimo konvenčních objektů umožňuje vytváření také takzvaných „svázaných proměnných“. Pomocí operátoru `tie` můžeme svázat proměnnou s třídou, která implementuje chování dané proměnné. Tímto způsobem můžeme abstrahovat chování komplexních datových struktur pomocí jednoduchých proměnných kteréhokoliv typu. Dobrým příkladem abstrakce externí datové struktury pomocí implementace jednoduché proměnné Perlu je reprezentace tabulkové struktury UNIXové databázové knihovny *DBM*. Svázané proměnné umožňují třídám, respektive balíčkům/modulům jazyka Perl implementovat krom všech typů proměnných, tzn. skalárů, polí i asociativních polí, i chování pojmenovaných deskriptorů souboru. Tímto lze například abstrahovat přístup ke schránkám typu BSD.

Deskriptory souborů

Pojmenované sekriptory souborů jsou pojmenované reprezentace (nejen) souborů. Nejsou o mnoho odlišné od UNIXových popisovačů souborů. Perl zpřístupňuje vestavěné funkce pro práci s nimi, jako jsou `open` pro otevření souboru, či objektu souboru podobnému, a vytvoření deskriptoru. Deskriptory sdílí stejná pravidla ukládání do tabulky symbolů jako subrutiny a ostatní proměnné. To znamená, že jméno deskriptoru souboru může být stejné jako jméno jiné proměnné či subrutiny. Deskriptory souborů, přestože se podobají UNIXovým popisovačům, nemají samy o sobě žádnou hodnotu a nelze je tedy například používat jako návratové hodnoty subrutin. Tato limitace se však dá obejít pomocí referencí na deskriptory souborů, které lze také vytvářet, a ty už samy hodnotou jsou a jako takové je možné je jako hodnotu používat. Všechny vestavěné funkce pro operace nad deskriptory souborů přijímají jako argumenty i reference na tyto deskriptory bez rozdílu. Perl také poskytuje několik výchozích otevřených deskriptorů souborů, které jsou přístupné v každém programu jazyka Perl. Těmito jsou standardní vstup a výstup (`STDIN`, `STDOUT`), standardní chybový výstup (`STDERR`), ale také reprezentace vstupních souborů `ARGV`. Tento virtuální soubor reprezentuje soubory předané programu pomocí příkazové řádky. Čtení z tohoto souboru vede na postupné čtení ze vstupních souborů v pořadí, v jakém byly specifikovány v argumentech použitých ke spuštění programu. Po kompletím přečtením všech souborů jakýkoliv další pokus o čteí skončí neúspěchem z důvodu konce souboru (EOF). Virtuální deskriptor souboru je také výchozím operandem operátoru pro čtení ze souboru (`<>`).

Vestavěné proměnné

Podobně jako programovací jazyky, kterými byl jazyk Perl inspirován, používá Perl značné množství vestavěných proměnných. Tyto proměnné mívají rozličné názvy a úlohy. Počínaje

[2]Příklad implementace soukromých atributů v Perlu pomocí uzávěrů: perlmonks.org/?node_id=8251

tzv. *výchozí proměnnou* `$_`, která je používána nejen jako implicitní skalární argument některých vestavěných funkcí (není-li specifikovaný jiný), a za určiných okolností (většinou ve výrazech podmínky cyklů) do ní i přiřazuje vyhodnocené hodnoty, nejsou-li tyto hodnoty přiřazeny jinam (například čtecí operátor `<>`).

Obdobou *výchozí proměnné* je *výchozí pole* `@_`. Nachází-li se spouštění programu v hlavní části, obsahuje tato proměnná ve výchozím stavu argumenty zadané do příkazové řádky při spouštění skriptu. V subrutině poté obsahuje všechny argumenty v pořadí, v jakém byly funkci předány.

Vestavěné proměnné obecně ovlivňují detaily fungování vestavěných operátorů či samotného interpretu, nebo zpřístupňují informace o prostředí, interpretu, aktuálním balíčku či stavu programu.

3.2.2 Syntax

Výrazy a operátory

Výrazy v perlu se skládají z termů – proměnných a literálů, ať už řetězcových, numerických, nebo literálů typu seznam – a operátorů. V Perlu existuje mnoho operátorů, unárních, binárních i ternárních. Za zmínku však stojí absence funkcí jako takových. To co by bylo v jiných jazycích považováno za volání funkce, v perlu je považováno za tzv. *seznamový operátor*. To jest unární operátor, který po své pravé straně očekává *seznam*. Toto umožňuje jejich použití bez závorek, viz výpis programu č. 3.1.

```
1 print "Hello ", "World", "\n";
2 print("Hello ", "World", "\n");
```

Výpis 3.1: Možnosti zápisu *seznamových operátorů*

Perl také disponuje množstvím vestavěných unárních a *seznamových* operátorů, které na první pohled mohou vypadat a mohou se chovat obdobně jako funkce v jiných jazycích. Mezi tyto operátory, patří například unární operátor `eval`, často používaný pro zachytávání chyb, nebo *seznamový* operátor `print`

Co se asociativity týče, krom operátorů pravě či levě asociativních nebo i kompletně nonasociativních, zná jazyk Perl ještě takzvanou *řetězovou* asociativitu, která je velmi častá u porovnávacích operátorů. Výrazy ve výpise č. 3.2 jsou sémanticky identické, s tím rozdílem, že v prvním případě dojde k vyhodnocení hodnoty `$b` pouze jednou a získaná hodnota bude použita v obou porovnáních.

```
1 $a < $b < $c;
2 $a < $b && $b < $c;
```

Výpis 3.2: Příklad řetězové asociativity operátoru

Bloky a struktura programu

Základním stavebním blokem každého programu je výrok. Výroky mohou být složeny z výrazů a podvýrazů a zároveň mohou samy výrazem být. Základním výrokem je výrok složený z výrazu.

Krom samotného výrazu může výrok obsahovat právě jedno rozšiřující klíčové slovo, viz výpis č. 3.3.

```
1 if VYRAZ
2 unless VYRAZ
3 while VYRAZ
4 until VYRAZ
5 for SEZNAM
6 foreach SEZNAM
```

Výpis 3.3: Seznam výroky-rozšiřujících klíčových slov jazyka Perl

Toto modifikuje výrok a upravuje podmínky jeho spuštění. Pokud je výrok rozšířen slovem `if`, bude proveden pouze tehdy, když *`VÝRAZ`* bude pravdivý (v případě `unless` opačně). Obdobně `while` bude opakovaně výrok provádět, dokud daná podmínka platí, `until` bude opakovat provádění výroku dokud daná podmínka platit nezačne. Očekávatelně tedy i `for` a `foreach` uvrhnou následující výraz do kontextu pole a následně pro každý prvek získaného pole jednou provedou výrok (přičemž aktuálně zpracovávaný prvek je k dispozici v proměnné `$_`).

Z takovýchto výroků je možno následně skládat výroky složené. Nejjednodušším složeným výrokem je *blok*. Jedná se o seznam výroku, vymezující rozsah jmenné vazby programu. Blok může být vymezen souborem, ze kterého jsou výroky čteny (ať už se jedná o modul importovaný do již běžícího skriptu, nebo přímo spouštěný skript), nebo může být vyznačen pomocí složených závorek.

Prostý blok programu bude spuštěn nepodmíněně, pouze jednou, a to v době kdy interpret narazí na jeho začátek. Takový blok má sémantiku „smyčky“, která se má provést pouze jednou, což znamená, že v jeho těle je možné použít výroky jako je `last` pro ukončení smyčky, nebo `next` pro další iteraci. Oba zmíněné výroky budou mít v tomto případě stejný účinek.

Spuštění bloku však můžeme podmínit pomocí složených výroků, seznam významnějších naleznete ve výpise č. 3.4.[3]

```
1 if/unless (VYRAZ) BLOK
2 if/unless (VYRAZ) BLOK else BLOK
3 if/unless (VYRAZ) BLOK elsif (VYRAZ) BLOK ... else BLOK
4 ...
5 while/until (VYRAZ) BLOK
6 ...
7 for/foreach (VYRAZ; VYRAZ; VYRAZ) BLOK
8 ...
9 FAZE BLOK
```

Výpis 3.4: Vybrané složené výroky jazyka Perl

Kromě očekávaných podmínek typu `if` a jeho protějšku `unless` a smyček `while` a `for`, za pozornost stojí poslední vypsaný složený výrok.

Výrok složený z identifikátoru fáze překladua bloku kódu umožňuje spuštění kódu po určité fázi života programu v jazyce Perl, viz tabulka č. 3.2.

Tyto bloky mohou přímo ovlivňovat způsob, jakým interpret překládá kód, kontrolovat dostupné funkcionality interpretu a na základě toho například upravit, která část knihovny

[3]Kompletní seznam je k dispozici v dokumentaci[14] na webu perldoc.perl.org

bude přeložena, aby byla zachována kompatibilita s hostitelským systémem a interpretem, inicializovat databázová připojení před spuštěním hlavní části programu a podobně.

Subrutiny

Subrutinou v Perlu myslíme (pojmenovaný) blok kódu, jež je možno invokovat na vyžádání, a který se chová jako *seznamový operátor*. Jazyk Perl ve své základní podobě (tj. bez použití rozšíření) nepodporuje signatury subrutin. Subrutiny jsou tak velmi podobné například funkcím ve skriptech pro prostředí Bash.

Subrutiny, stejně jako jiné proměnné, lze referencovat a následně při jejich volání dereferencovat. Z toho tedy vyplývá, že v jazyce Perl jsou subrutiny *objektem první kategorie*[4].

Při invokaci subrutiny, jako *seznamového operátoru*, bude *seznam* argumentů přístupný v těle subrutiny ve vestavěné proměnné `@_`.

Pro návrat ze subrutiny je k dispozici unární operátor `return`. Tento operátor očekává jeden argument, který vyhodnotí v kontextu, ve kterém byla subrutina zavolána. Jeho použití je volitelné, interpret jej však implicitně provádí při opuštění bloku vymezující subrutinu. Nevrátí-li subrutina explicitně nějakou hodnotu, její návratová hodnota bude hodnota posledního vyhodnoceného výrazu.

Informace o kontextu, ve kterém byla subrutina zavolána je k dispozici pomocí operátoru `wantarray`, který neočekává žádné argumenty. Je tedy možné, že v závislosti na kontextu se bude jedna subrutina chovat zcela odlišně. Zároveň operátor return vyhodnotí svůj argument v kontextu, ve kterém byla funkce zavolána.

3.3 Popis interpretu programovacího jazyka Perl

Tato sekce popisuje základní fungování interpretu jazyka Perl, taktéž známého jako `perl`. Pokud není uvedeno jinak, informace v této sekci byly přejaty z [14].

Jak již bylo popsáno, Perl je jazykem interpretovaným. Jeho programy, taktéž zvané skripty, ke svému běhu potřebují interpret, který zajistí jejich přeložení a běh. Tento interpret za běhu dynamicky překládá kód do vnitřní reprezentace, kterou následně spouští.

3.3.1 Vnitřní reprezentace kódu interpretu jazyka Perl

Interpret jazyka Perl používá pro svou vnitřní reprezentaci a běh programu zásobníkový kód, který je po přeložení aktuálně spouštěné části programu vykonáván interpretem. K vykonání tohoto vnitřního kódu je použita implementace zásobníkového automatu s několika zásobníky. Na argumentový zásobník jsou ukládány hodnoty načtených proměnných a operátory z něj získávají své argumenty. K němu používá ještě doplňující zásobnk, který se stará o dynamický jmenný vazebný prostor. Jednotlivé instrukce vnitřního kódu, dokumentací často nazývaného bytekódem, přestože tento není přenositelný, jako jeho obdoba na platformě JVM, nesou, krom odkazu na své operandy, i referenci na rutinu interpretu obsluhující danou instrukci a odkaz na následující instrukci ve spouštěném pořadí, jedná se tedy o interpret s přímo provázaným kódem.

[4] Z anglického First-Class Object

3.3.2 Průběh překladu kódu jazyka Perl

Interpret překládá, optimalizuje a spouští kód po částech, takže se může stát, že část programu je vykonávána a vykonána, ještě před tím, než byl započat překlad jiné části programu. Tato posloupnost překladu a spouštění je pevně daná.

Prvně dojde k parsování vstupního kódu, program `perl` (interpret jazyka Perl), k tomu používá LR parser generovaný pomocí nástroje YACC. Tento parser velmi úzce spolupracuje s tokenizérem. Jazyk Perl je velmi závislý na kontextu, proto je tato komunikace klíčová. Často není možné určit přesný typ tokenu pouze na základě vstupu, proto se tokenizér dle potřeby rozhoduje podle toho, jaký typ tokenu parser zrovna očekává, a na základě toho klasifikuje přečtený token. Parser pak při průchodu programem generuje abstraktní syntaktický strom (AST) operací.

V průběhu parsování dochází k první vlně optimalizací. Jelikož LR parser funguje způsobem zdola nahoru, je možné v tomto kroku provádět i optimalizace zdola nahoru. V tomto kroku tak dojde k zjednodušení derivačního stromu vygenerovaného LR parserem a k seskládání konstant (tzn. výrazy s konstantní hodnotou budou vypočteny již při překladu).

Druhým průchodem dochází k propagaci kontextu. Kontext se v AST propaguje shora dolů, a tak i tento průchod AST je shora dolů. Každý uzel AST pak propaguje kontext který buď sám vyžaduje po svých následovnících nebo nebo ten, který po něm vyžaduje jeho předek. Speciálním případem je pak operace přiřazení, která na základě své levé strany určuje jaký kontext bude propagovat do strany pravé. K propagaci a evaluaci kontextu může však docházet i za běhu.

Třetí průchod již není průchod AST, nýbrž se jedná o průchod v pořadí spouštění instrukcí. Zde se odehrává poslední část optimalizací, jako jsou optimalizace pro lepší využití zásobníku a omezení přístupu do haldy.

Po dokončení třetího průchodu začíná spouštění vnitřního kódu. První instrukce je předána spouštěcí funkci, která tyto instrukce vykonává. Součástí každé instrukce je pak odkaz na instrukci následující.

3.3.3 Fáze spouštění programu jazyka Perl

Interpret jazyka Perl nepřekládá celý program najednou, nýbrž po částech. Tyto části mohou být specifické buď pro celý program, nebo pro jednotlivé překladové jednotky. Překladovou jednotkou je jak již zmíněný celý program, tak jednotlivé načítané moduly a kód spouštěný operátorem `eval`. Kód spouštěný operátorem `eval`, narozdíl od subrutin, není překládán společně s překladovou jednotkou, ve které se nachází, jelikož ještě ani nemusí existovat. Proto jsou výroky typu `eval` samostatnou překladovou jednotkou, která začne svůj překlad až v tu chvíli, kdy spouštění programu dojde k danému operátoru `eval`. Výčet jednotlivých fází překladu je k nalezení v tabulce č. 3.2.

`BEGIN`	Blok bude spuštěn ihned po svém překladu, tj. před kompilací zbytku překladové jednotky.
`CHECK`	Blok bude spuštěn po dokončení překladu celého programu.
`UNITCHECK`	Blok bude spuštěn po dokončení překladu aktuální překladové jednotky.
`INIT`	Blok bude spuštěn před začátkem spouštění programu.
`END`	Blok bude spuštěn po dokončení spouštění programu.

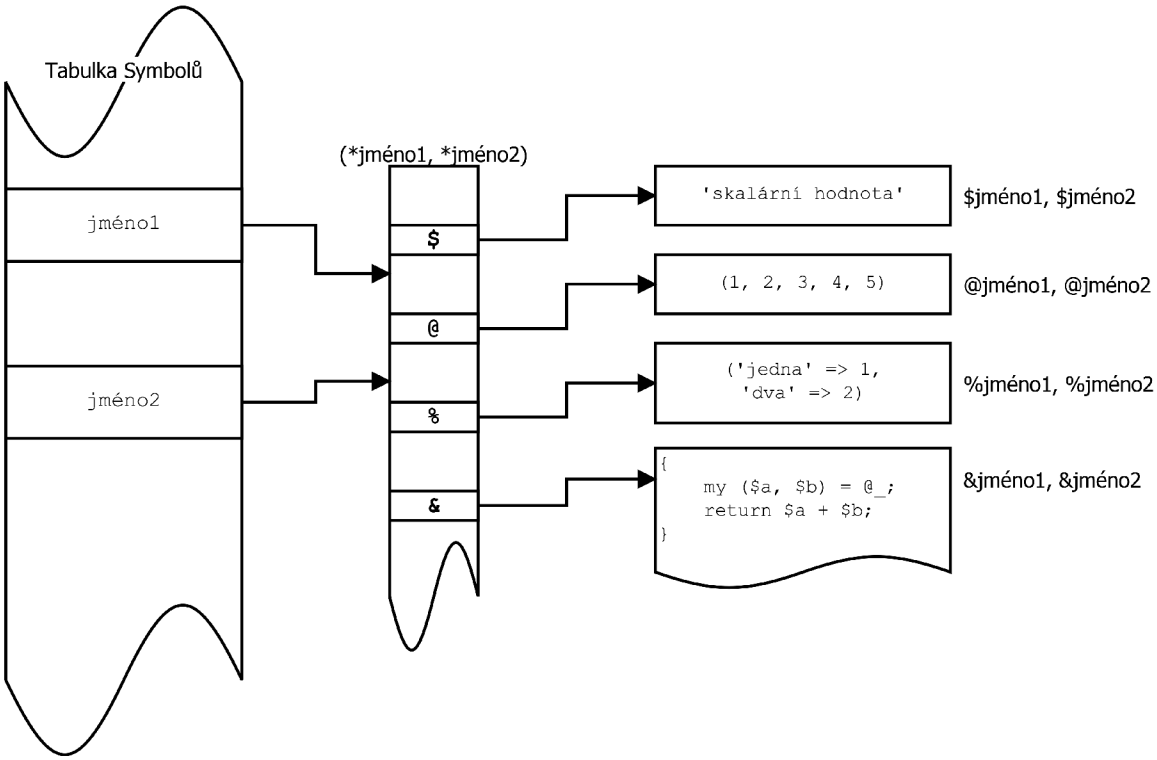
Tabulka 3.2: Výčet fází překladu skriptu jazyka Perl

3.4 Rozšiřitelnost jazyka Perl

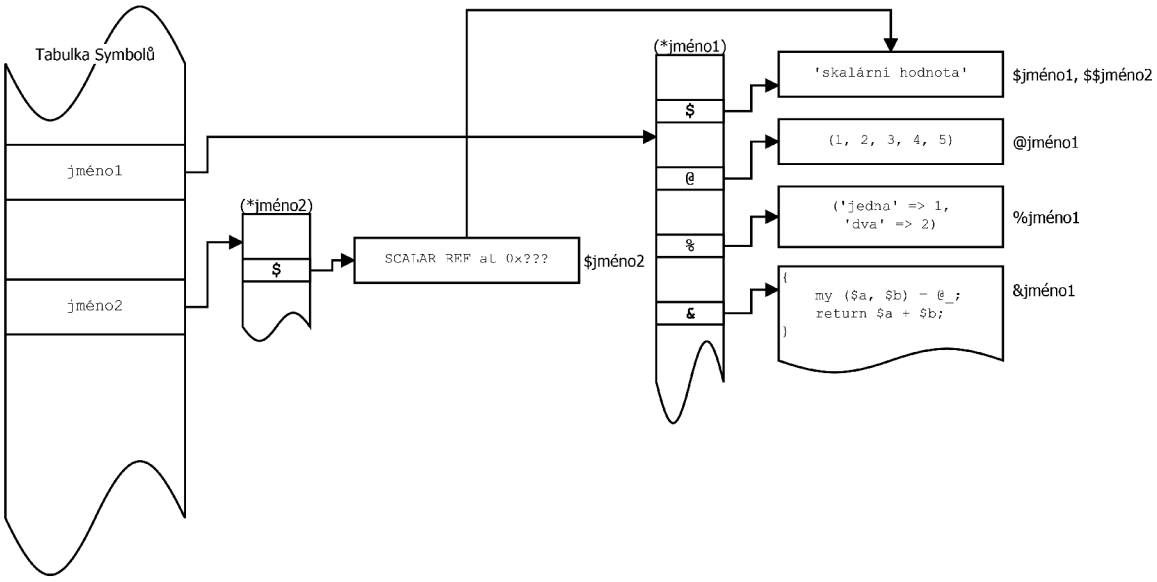
Jazyk Perl je velmi flexibilní a rozšiřitelný jazyk, jeho fungování je možno rozšiřovat jak pomocí modulů, které zpřístupní nové funkce, proměnné a objekty, ale i pomocí přímých zásuvných modulů interpretu, které přímo ovlivňují, jak interpret překládá kód. Takovýmto způsobem je možno značně změnit syntax a sémantiku celého jazyka Perl.

Zásuvné moduly interpretu mohou například zpřístupňovat nové (pseudo)vestavěné operátory, nová rozhraní, a nebo upravovat ta stávající, od kompletní funkcionality počínaje, po pouhé optimalizační detaily.

Moduly jazyka Perl potom mohou manipulovat přímo se vstupem samotného interpretu. Takovýto modul, nebo jeho část, se může chovat jako preprocesor či vstupní filtr a upravovat či rozšiřovat syntaktické prostředky jazyka.



Obrázek 3.1: Diagram tabulky symbolů interpretu jazyka Perl s aliasem proměnné



Obrázek 3.2: Diagram tabulky symbolů interpretu jazyka Perl s referencí na hodnotu

Kapitola 4

Popis vybraného cílového jazyka

Tato kapitola stručně popisuje vlastnosti vybraného cílového jazyka, jímž je JavaScript. Není-li uvedeno jinak, informace v této kapitole byly přejaty z [5] a [12].

4.1 Popis programovacího jazyka JavaScript

JavaScript, jazyk standardizovaný mezinárodní normou *ECMA-262*[5] a standardně nazývaný ECMAScript, je objektově založený, imperativní, interpretovaný, za běhu kompilovaný jazyk. Jeho nejrozšířenějším použitím jsou webové stránky, ale je používán i pro mnoho dalších účelů, jako jsou uživatelská rozhraní i serverové aplikace. Patří mezi nejrozšířenější jazyky, jak dokazuje studie popularity programovacích jazyků v projektech s otevřeným kódem[3].

4.2 Základní vlastnosti syntax a sémantiky

Jelikož se jedná o tolik rozšířený a standardizovaný jazyk, existuje mnoho implementací interpretu. Tato práce však uvažuje pouze prostředí *Node.js*, založené na JavaScriptovém enginu *V8* od společnosti Google, implementující standard ECMA-Script verze 6.

4.2.1 Proměnné a datové struktury

JavaScript je objektově založeným jazykem. To znamená, že všechny úlohy hostitelského interpretu včetně základních úloh jazyka jako takového jsou založeny na objektech a komunikaci s nimi. Objekt je tedy kolekce nula a více atributů. Atributy jsou potom místa, která mohou obsahovat ostatní objekty, *primitivní hodnoty*, nebo funkce. *Primitivní hodnota* je jedním z následujících vestavěných datových typů: *Nedefinované*, *Null*, *Booleovská hodnota*, *Řetězec* nebo *Symbol*. Objekt je instancí vestavěného typu *Objekt*, funkci spojenou s objektem prostřednictvím atributu daného objektu pak nazýváme *metodou*. Z toho také vyplývá fakt, že funkce jsou také objektem první kategorie a jako takové mohou být přiřazeny či navráceny jako návratová hodnota z funkce.

Krom *primitivních hodnot*, JavaScript poskytuje řadu dalších složených vestavěných typů, jako je například uspořádaná kolekce *Array*, nebo unikátní kolekce *Set*. Ovšem i vestavěný typ *Objekt* lze, díky jeho vlastnosti dynamicky přidávat nebo pozbývat atributy, použít jako asociativní pole.

V JavaScriptu můžeme delkarovat a používat proměnné. Lze je využít všude, kde lze vložit literál jakéhokoliv typu. Platnost proměnné může být omezena na aktuální dokument

(globální), na aktuální funkci (lokální), nebo pouze na aktuální blok kódu, ohraničený složenými závorkami. Velmi zajímavou vlastností proměnných v jazyce JavaScript jsou *vyzdvihnuté deklarace*. Proměnná deklarovaná jako lokální je definována již na začátku jmenného vazebného prostoru, ve kterém se nachází její deklarace. To znamená, že je možno přistupovat k proměnným ještě před tím, než provádení programu pokročilo na daný řádek, ve kterém se nachází deklarace proměnné. Toto se ovšem týká pouze deklarace, nikoliv inicializace, takže její hodnota bude *nedefinovaná*. Také se to netýká proměnných deklarovaných jako konstanta nebo proměnných s platností omezenou na aktuální blok. Deklarace neanonymních funkcí jsou *vyzdvihnuty* na začátek aktuálního jmenného vazebného prostoru vždy.

JavaScript je slabě typovaný jazyk, což znamená, že jeho proměnné nemají pevně stanovený typ a mohou ho libovolně měnit. Nezáleží tedy na tom, zda je do proměnné uložena hodnota *primitivní* či složená, či je v průběhu programu zaměněna.

Objekty jsou pak základním stavebním prvkem celého jazyka. Přestože je v JavaScriptu možné definovat a deklarovat třídy, JavaScript, dle své definice, třídy nezná. Kde v jazycích, s objekty založenými na třídách, by byly atributy vlastnosti objektu a metody vlastnosti třídy, v JavaScriptu jsou tyto všechny vlastnosti přímo daného objektu nebo objektu, který aktuální objekt vytvořil. Interpret potom bude hledat atributy nejprve v objektu aktuálním, pokud nenajde, bude pokračovat k objektu (prototypu), který je tvůrcem aktuálního objektu, čímž vzniká prototypový řetězec.

Program může komunikovat s vnějším prostředím pomocí objektů zpřístupněných samotnou implementací interpretu, ať už se jedná o tzv. globální objekt, zpravidla obsahující globálně definované proměnné, nebo objekty specializované, jako je například hojně používaný objekt `console`, zprostředkovávající přístup ke konzoli.

Jak již bylo zmíněno, všechny proměnné v JavaScriptu jsou objektem, nezávisle na tom, zda je to objekt jednoho z *primitivních* typů, nebo objekt složitý. Neexistuje zde však přetěžování operátorů. Standard JavaScriptu proto definuje sadu metod, které může objekt mít, aby byl převeditelný na *primitivní hodnotu*. Očekává-li nějaký operátor některou z primitivních hodnot, bude pak postupovat podle pravidel pro „donucení“ proměnné do jedné z *primitivních* hodnot. Například operátor `+`, je-li jedním z jeho operandů *řetězec*, bude se snažit „donutit“ i druhý operand do formy řetězce, a ty potom konkatenuje. Interpret se pokusí zavolat sekvenci metod donucovaného objektu, v tomto případě konrétně by volil sekvenci ve výpise č. 4.1. Objekt takto může podstoupit i sérii těchto „donucovacích“ sekvencí, například nejde-li převést přímo na řetězec, interpret se může pokusit ho převést na (libovolnou) *primitivní* hodnotu a tu posléze na řetězec.

```
1 objekt[@@toPrimitive]("string");
2 objekt.toString();
3 objekt.valueOf();
```

Výpis 4.1: Sekvence volaných metod pro „donucení“ objektu do tvaru řetězce v jazyce JavaScript

4.2.2 Syntax

Základním stavebním prvkem syntax jazyka JavaScript je výraz. Ten se může skládat z literálů jednotlivých primitivních typů, operátorů a proměnných. Tyto výrazy lze sdružovat do výroků.

Výroky jazyka JavaScript

Výrokem v jazyce JavaScript myslíme konstrukci skládající se z výrazů oddělených středníkem. Nejjednodušším z nich je *blok* skládající se z libovolného počtu výrazů a ohraničený složenými závorkami. Takovýto blok vymezuje platnost proměnných deklarovaných jako platných pouze v aktuálním bloku.

Z bloků pak následně pomocí klíčových slov jazyka JavaScript můžeme vytvářet výroky podmíněné či smyčky typu `for`, `foreach` či `while`, nebo také funkce pojmenované či nepojmenované, a tím pádem i jejich uzávěry. Krom těchto existují i výroky pro vyvolávání a zotavení se z výjimek. Výrok `throw VÝRAZ;` vyvolá výjimku s hodnotou, která bude výsledkem vyhodnocení výrazu. Vyvolání výjimky vede na okamžité přerušení vykonávání aktuálního bloku kódu. Pokud není tento blok přímo součástí výroku `try ...catch ...finally`, bude hodnota této výjimky propagována do bloku nadřazeného dokud buď nedojde k jejímu zpracování, nebo nedojde k ukončení programu z důvodu nezachycené výjimky. Hodnota výjimky může být jakéhokoliv typu, ať už *primitivního* nebo složitějšího. Dojde-li k vyvolání nebo k propagaci výjimky do bloku `try`, bude tato výjimka předána bloku `catch`, tento blok se následně může s výjimkou buď vypořádat, nebo ji znovu vyvolat, aby byla obsloužena blokem nadřazeným. Nezávisle na výsledku bloku `catch`, bude po jeho dokončení či opuštění spuštěn blok `finally`, pokud existuje.

Jazyk JavaScript má také vestavěnou podporu pro asynchroní, událostmi řízený kód. Ta je však mimo zaměření této práce a nebude detailněji popsána.

4.3 Popis interpretu V8 jazyka JavaScript

V8 je souhrnný název implementace interpretu jazyků JavaScript a WebAssembly společnosti Google. Tento interpret je používán mimo jiné v prostředí pro spouštění serverových skriptů *Node.js*, nebo prohlížeči *Google Chrome*.

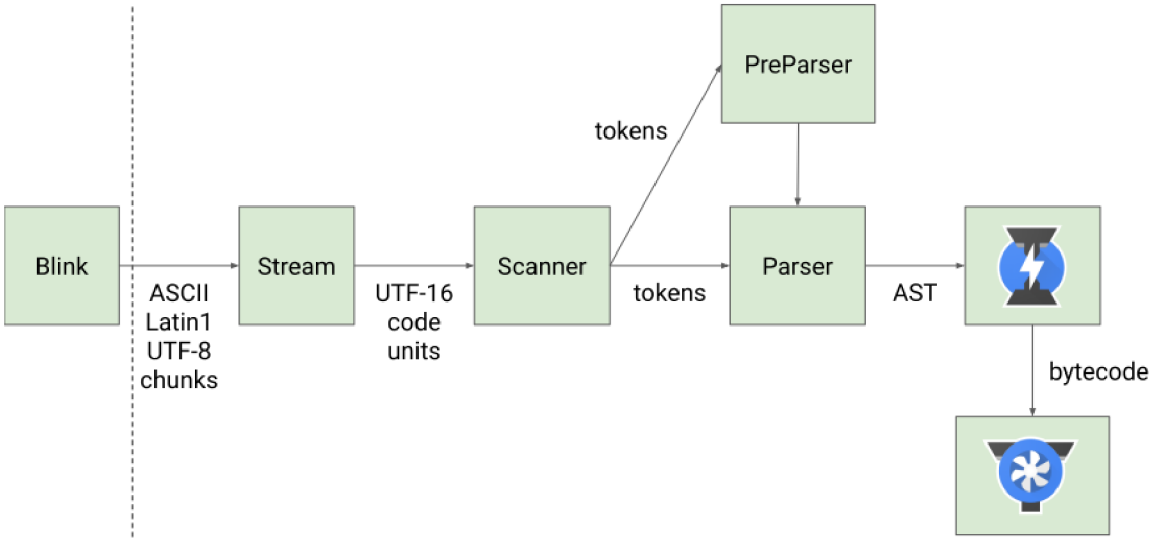
Interpret V8 jazyka JavaScript nejprve konvertuje vstupní proud znaků do kódování UTF-16, jak je znázorněno na obrázku č. 4.1. Takto konvertovaný proud znaků je následně podle dostupnosti předáván lexikálnímu analyzátoru – skeneru. Ten předává vytvořené tokeny parseru, potažmo preparseru. Jelikož V8 je interpret vytvořený pro použití v prohlížeči k překladu a spouštění kódu načítaných přes internet, je optimalizován tak, aby byl schopný začít vykonávat přijímaný program co nejrychleji, tak aby nebylo vykreslování stránky zdržováno překladem zdrojového kódu JavaScriptu. Proto byl vedle standardního syntaktického analyzátoru (parseru) implementován i takzvaný preparser. Jedná se o zjednodušenou verzi parseru, jejíž úloha je povrchně zkontrolovat validitu kódu, který není nutný k okamžitému spuštění analyzovaného programu. Kompletní analýza a generování AST je u těchto částí odloženo, dokud není nutné tento kód přeložit. Toto se jedná například definic funkcí. Preparser v takovém případě pouze zkontroluje zda je daná funkce syntakticky správná, vyhledá všechny informace o ní, které by mohly být třeba při parsování zbytku kódu, ale již neprodukuje AST, jak tomu je v případe parseru hlavního[16].

4.3.1 Popis vnitřní reprezentace kódu interpretu jazyka JavaScript V8

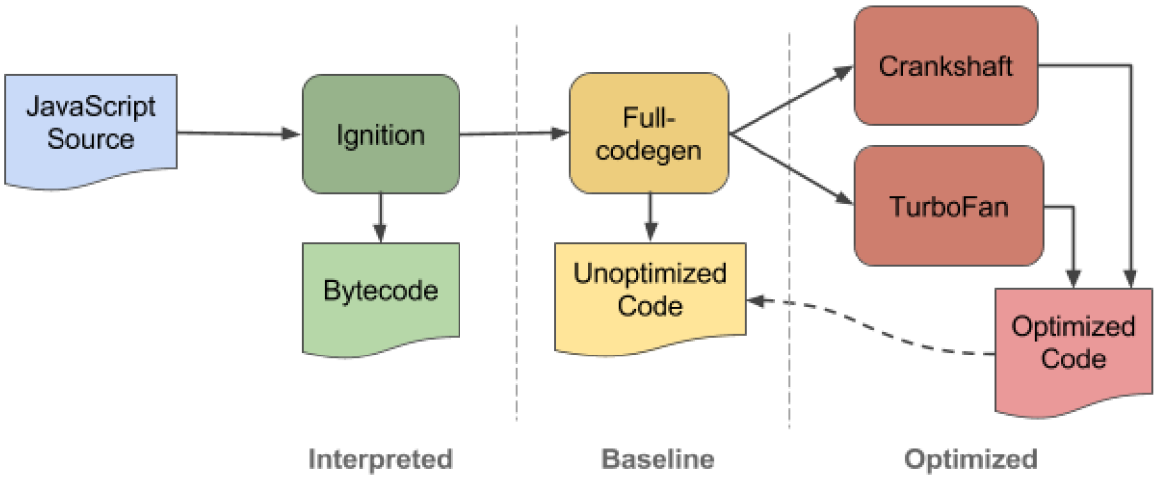
Interpret V8 používá pro interpretaci svého vnitřního kódu interpret zvaný *Ignition* [11]. Jedná se o virtuální stroj s akumulátorem, obecnými registry a zásobníkem. Tento virtuální procesor vykonává instrukce vnitřního kódu, též označovaného jako *bytekód*. Operandy jednotlivých instrukcí se zpravidla nacházejí v registrech a pracují obsahem registru aku-

mulátoru, zatímco zásobník je používán pro udržování lokálních proměnných, předávání argumentů volaným funkcím a ukládání návratové pozice. Tento interpret by se tedy dal kategorizovat jako interpret klasický, jelikož obsluha instrucke je dána jejím operačním kódem.

Interpret *Ignition* zpravidla spolupracuje s Just-In-Time (*JIT*) kompilátory, konkrétně s referenčním kompilátorem, který se nezabývá optimalizacemi – pouze kompiluje vnitřní kód do kódu strojového, a dvěma variantami optimalizujících kompilátorů, *Crankshaft* a *TurboFan* které při kompilaci vnitřního bytekódu na strojový kód provádějí další optimalizace. Touto kooperací vznikají lepší podmínky pro optimální běh programů. Často spouštěné úryvky kódu budou zkompilovány do kódu nativního, méně používaný kód však zůstane v podobě bytekódu, čímž sníží nároky běhu programu na paměť hostujícího systému.



Obrázek 4.1: Diagram přední části interpretu V8 jazyka JavaScript. Přejato z [17]



Obrázek 4.2: Schéma spolupráce interpretu *Ignition* s JIT kompilátory.[11]

Kapitola 5

Návrh překladače mezi vybranými jazyky

Na základě informací z předchozích kapitol nyní můžeme navrhnout možnou interpretaci transpilátoru jazyka Perl do jazyka JavaScript. Jelikož oba jazyky používají převážně imperativní paradigma, bude toto také hlavním zaměřením.

Prvním krokem implementace transpilátoru bude analýza vstupního jazyka, počítaje analýzou lexikální. Zde je možno využít již existující implementace tokenizérů jazyka Perl implementovaných v samotném jazyce Perl, jako je například balíček `PPI` se svým modulem `Tokenizer`[1], nebo jinou existující implementaci v jiném programovacím jazyce, jako je například implementace tokenizéru Perlu pro jazyk Python v rámci knihovny Pygments[2]. Takto získaný proud tokenů je potřeba podrobit analýze syntaktické.

Pro syntaktickou analýzu bude třeba využití parsovacího algoritmu. Jako varianty se nabízí LR syntaktické analyzátory a jejich generátory. Takovéto analyzátory však produkují velice složité stromy a návrh gramatik pro ně je netriviální záležitost. Jednodušší možností je precedenční analýza. Nakonec bylo rozhodnuto ve prospěch algoritmu soupání precedencí (precedence-climbing), zveřejněného poprvé Vaughan R. Prattem v roce 1973[15].

5.1 Algoritmus stoupání precedencí

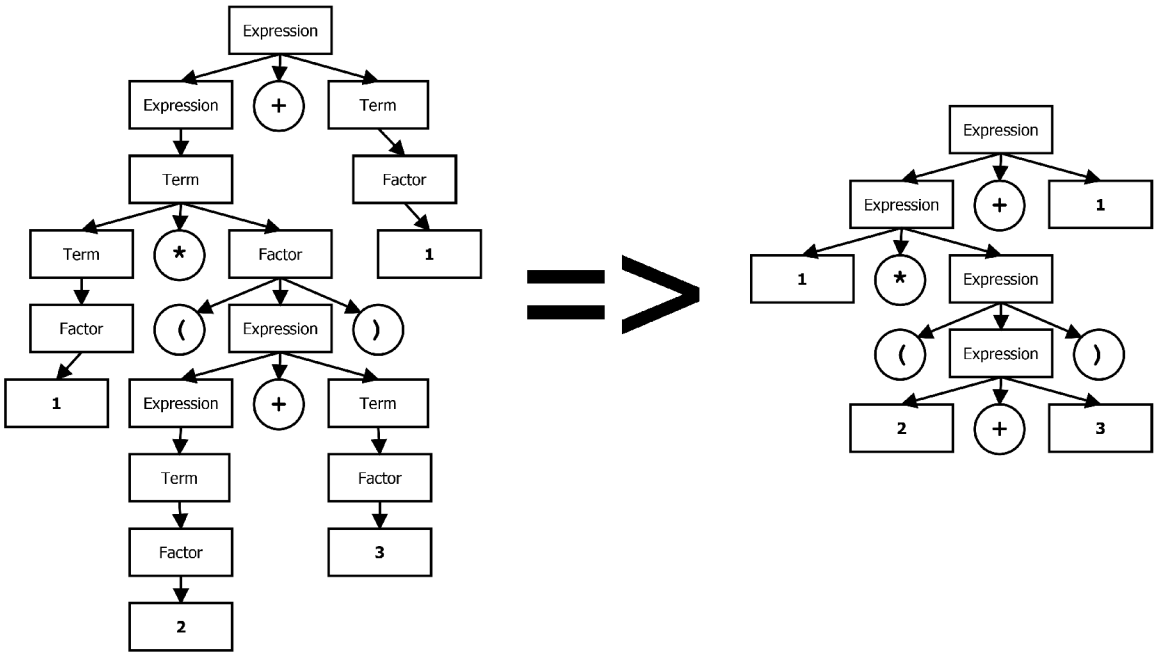
Jedná se o algoritmus syntaktického analyzátoru pracujícího na principu rekurzivního sestupu, následujícího precedenci operátorů a operací. AST produkované tímto algoritmem bývají daleko stručnější, než AST generované srovnatelnými metodami rekurzivního sestupu, jako jsou třeba parsery typu LR.

```
1 Expression => Expression + Term
2 Expression => Term
3 Term => Term * Factor
4 Term => Factor
5 Factor => (Expression)
6 Factor => <number>
```

Výpis 5.1: Ukázková LR gramatika[2]

[1] metacpan.org/pod/PPI::Tokenizer
[2] pygments.org

Uvažujme například gramatiku danou výpisem č. 5.1. Použijeme-li ji pro analýzu výrazu `1 * (2 + 3) + 1`, získáme AST ve tvaru jak je vyobrazeno na obrázku č. 5.1 vlevo. Tento AST je zbytečně složitý s dlouhými derivačními větvemi. Pro srovnání je uveden i minimalizovaný AST odpovídající stejnému výrazu. Prattův algoritmus stoupání precedencí je schopen generovat takovéto AST bez nutnosti následné minimalizace.



Obrázek 5.1: Příklad AST vytvořeného LR parserem a jeho zjednodušená forma

Mějme tedy tabulku operátorů s jejich ohodnocením, dle precedence, a asociativitou, rozdělené na unární a binární (viz tabulka č. 5.1). Fungování takového parseru by se pak dalo vyjádřit pomocí gramatiky ve výpise č. 5.2.

Operace	Precedence	Arita	Asociativita
`||`	0	Binární	Levá
`&&`	1	Binární	Levá
`=`	2	Binární	Levá
`+`, `-`	3	Binární	Levá
`-`	4	Unární	Pravá
`*`, `/`	5	Binární	Levá
`^`	6	Binární	Pravá

Tabulka 5.1: Příklad tabulky precedencí

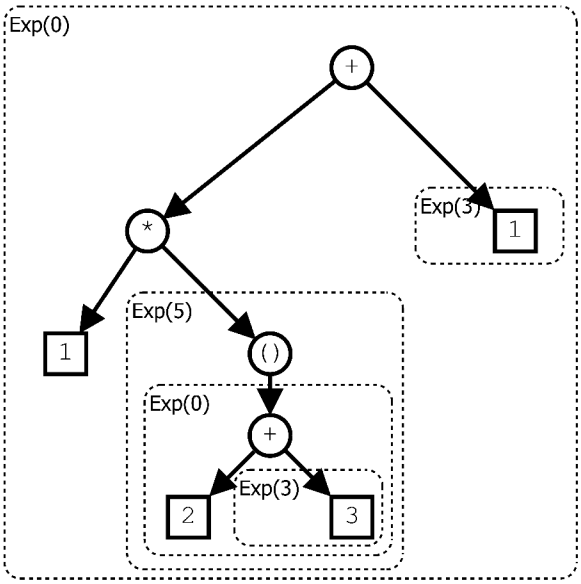
```
E => Exp(0)
Exp(p) => P { B Exp(q) }
P => U Exp(q) | '(' E ')' | value
B => '+' | '-' | '*' | '/' | '*' | '^' | '||' | '&&' | '='
U => '-'
```

Výpis 5.2: Gramatika algoritmu stoupání precedencí[13]

Tato gramatika používá parametrizovaný nonterminál `Exp(p)`. Tento nonterminál, dle pravidla č. 2, je možné expandovat na subvýraz neobsahující žádný binární operátor s precedenční silou menší než `p`. Hodnota tohoto parametru poté záleží na posledním operátoru použitého v rámci expanze pravidla č. 2 a 3, kdy nastavení parametru `q` je dáno precedencí předcházejícího operátoru a jeho asociativitou a to následujícím způsobem[13]:

- Pokud je následující token binární operátor:
  - Pokud je tento operátor levě asociativní, `q` budiž precedenční síla tohoto operátoru + 1
  - Pokud je tento operátor pravě asociativní, `q` budiž precedenční síla tohoto operátoru
- Pokud je následující token operátor unární, `q` budiž precedenční síla tohoto operátoru

Použijeme-li tento algoritmus na již výše zmíněný výraz `1 * (2 + 3) + 1`, dostaneme derivační strom znázorněný na obrázku č. 5.2. Na obrázku jsou vyznačeny domény které byly expandovány z nonterminálu `Exp(p)` společně s použitým parametrem `p`.

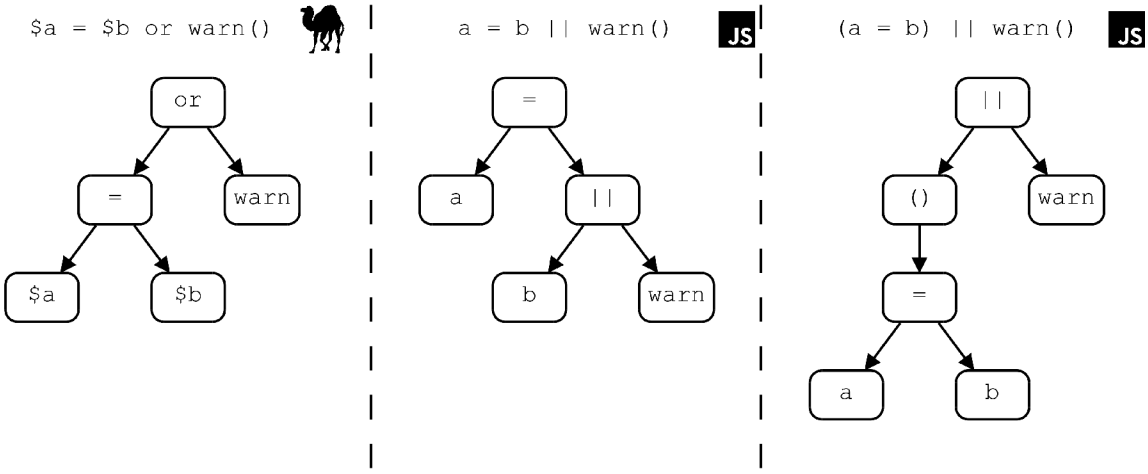


Obrázek 5.2: Schéma derivačního stromu získaného pomocí Prattova algoritmu stoupání precedencí

5.2 Statická analýza AST

S takto sestaveným AST můžeme přistoupit ke statické analýze kontextu. Zde se budeme řídit principy používanými v interpretu jazyka Perl. Výchozím kontextem je tedy kontext nulový. Tento kontext nastavíme kořeni AST a následně provedeme průchod do hloubky, nastavujíc adekvátní kontexty, s několika speciálními případy. Většina uzlů AST pouze předá svůj kontext svým následovníkům. Mezi výjimky však patří například uzly reprezentující přiřazení. Těmto uzlům bude přiřazen kontext jim předcházejících uzlů, avšak dále se tento kontext propagovat nebude. Místo toho dojde k vytvoření nové hodnoty kontextu na základě levé strany přiřazení. Ta bude propagována do pravého podstromu.

Na základě takto zpropagovaného kontextu může začít překlad. Opět rekurzivním průchodem AST do hloubky. To znamená, že překladem každého uzlu jest překlad složený z prvků které pramení z něj samotného, a z překladu jeho podstromu. Překlad uzlu podmínky „if“ tedy bude složený z klíčového slova `if`, překladu podstromu vyjadřující podmínku a překladu podstromu reprezentující podmíněný blok.



Obrázek 5.3: Rozdílný výklad priorit v cílovém a zdrojovém jazyce

V průběhu tohoto překladu musí dojít k několika dalším kontrolám. První z nich je kontrola převrácených precedencí. K převrácení precedencí může dojít v případě, kdy se precedence sémanticky odpovídající dvojice operátorů liší. Jako příklad můžeme uvažovat dvojici operátorů `or`, jazyka Perl, a `||` jazyka JavaScript. V obou případech se jedná o sémanticky stejný operátor – logickou disjunkci. Avšak varianta tohoto operátoru v jazyce JavaScript má vyšší precedenci než jeho obdoba v jazyce Perl. Toto by mohlo vyústit v rozdílné interpretace vstupu. Uvážímeli například výrok `$a = $b or warn()`, Perl jeho sémantický význam chápe jako „přiřaď skalární hodnotu proměnné `b` do proměnné `a`, a pokud tato přiřazená hodnota je vyhodnotitelná jako pravdivá v booleovském kontextu, pokračuj dále. Pokud však ne, spusť subrutinu `warn` a až pak pokračuj.“ Tento výklad je vyjádřen pomocí AST na obrázku č. 5.3, část vlevo. Pokud však tento výrok přeložíme do cílového jazyka pomocí prostého průchodu AST, jak už bylo popsáno, dojde k nedodržení sémantiky. Přestože operátor `=` i operátor `||` mají v cílovém jazyce stejnou sémantiku jako jejich alternativy v jazyce zdrojovém, nesdílí stejné relativní priority. Důsledkem toho bude do proměnné `a` v přeloženém programu přiřazena buď hodnota proměnné `b`, nebo návratová hodnota funkce `warn`, v případě, že hodnota proměnné `b` nebyla shledána pravdivou, viz obrázek 5.3, uprostřed. Tomuto můžeme zabránit ohodnocením každého operátoru v AST

zdrojového jazyka prioritou daného operátoru v cílovém jazyce. Posléze, ať už při speciálním průchodu či při průchodu za účelem překladu, můžeme kontrolovat takto invertované priority jednoduchým srovnáním cílové priority aktuálního uzlu a priority uzlu následnického. Pokud je priorita uzlu následnického vyšší, je vše v pořádku. Pokud ne, je třeba zvýšit jeho cílovou prioritu, například vložením závorek, jak je znázorněno na obrázku 5.3 vpravo.

Další kontrola je kontrola sémantiky a vynucování typů. Příkladem tohoto je třeba dvojice operátorů jazyka Perl `+` a `.`. Operátor `+` má sémantiku numerického sčítání a při použití provede vynucení numerického typu u obou svých operandů. Na druhou stranu operátor `.` má sémantiku konkatenace řetězců a sám provede vynucení typu řetězec u obou operandů. Cílový jazyk však oba tyto operátory skládá do jednoho; operátor `+` v cílovém jazyce sdílí obě tyto sémantiky v závislosti na typu jednoho či obou operandů. Jeho sémantika je numerické sčítaní, pokud oba operandy jsou numerického typu. Pokud však jeden z operandů je typu řetězce, dojde k vynucení typu řetězce i u operandu druhého a následné konkatenaci obou řetězců. Tento fakt by mohl vést na špatné překlady hned dvěma různými způsoby. Uvažme následující výraz:

```
1 print 1 . 2 . "\n";
```

Sémantika tohoto výrazu je konkatenace řetězce `'1'` a `'2'`, výsledek této konkatenace následně konkatenovat s řetězcem `"\n"` a výsledek vypsat do výstupu. Očekávaný výstup je tedy `12`, s odřádkováním na konci řádku.

Přeložíme-li tento kód však bez kontroly vynucení typů následovně:

```
1 console.log(1 + 2 + "\n");
```

Jeho sémantika se změní na numerický součet čísel `1` a `2` a až následné vynucení typu řetězce a konkatenaci s řetězcem `"\n"`. Výstupem tedy bude `3`. Pro dosažení odpovídajícího výsledku je tedy třeba doplnit ruční vynucení typu řetězec u jednoho z čísel `1` nebo `2`, například následovně:

```
1 console.log(1 + (2).toString() + "\n");
```

Opačný případ bude ovšem u výrazu, kde je první konkatenační operátor ve zdrojovém jazyce nahrazen za operátor numerického sčítání a jeden z jeho operandů, například číslo `2`, je nahrazen svou řetězcovou variantou (`"2"`):

```
1 print 1 + "2" . "\n";
```

Překlad do cílového jazyka bude opět vypadat podobně, jako v případě prvním, avšak jeho sémantika bude naprosto odlišná. Kdeže očekávaný výstup je `3`, výstup přeloženého programu bude `12`, jelikož první operátor + se zachová jako konkatenace také, místo očekávaného numerického sčítání.

```
1 console.log(1 + "2" + "\n");
```

Řešení je v tomto případě elegantnější než v případě předchozím. Použijeme unární operátor `+` cílového jazyka. Jeho sémantika je takřka nulová, dojde pouze k vynucení numerického typu u jeho operandu, což je jeho výsledkem. Upravený výraz pro shodnou sémantiku by vypadal následovně:

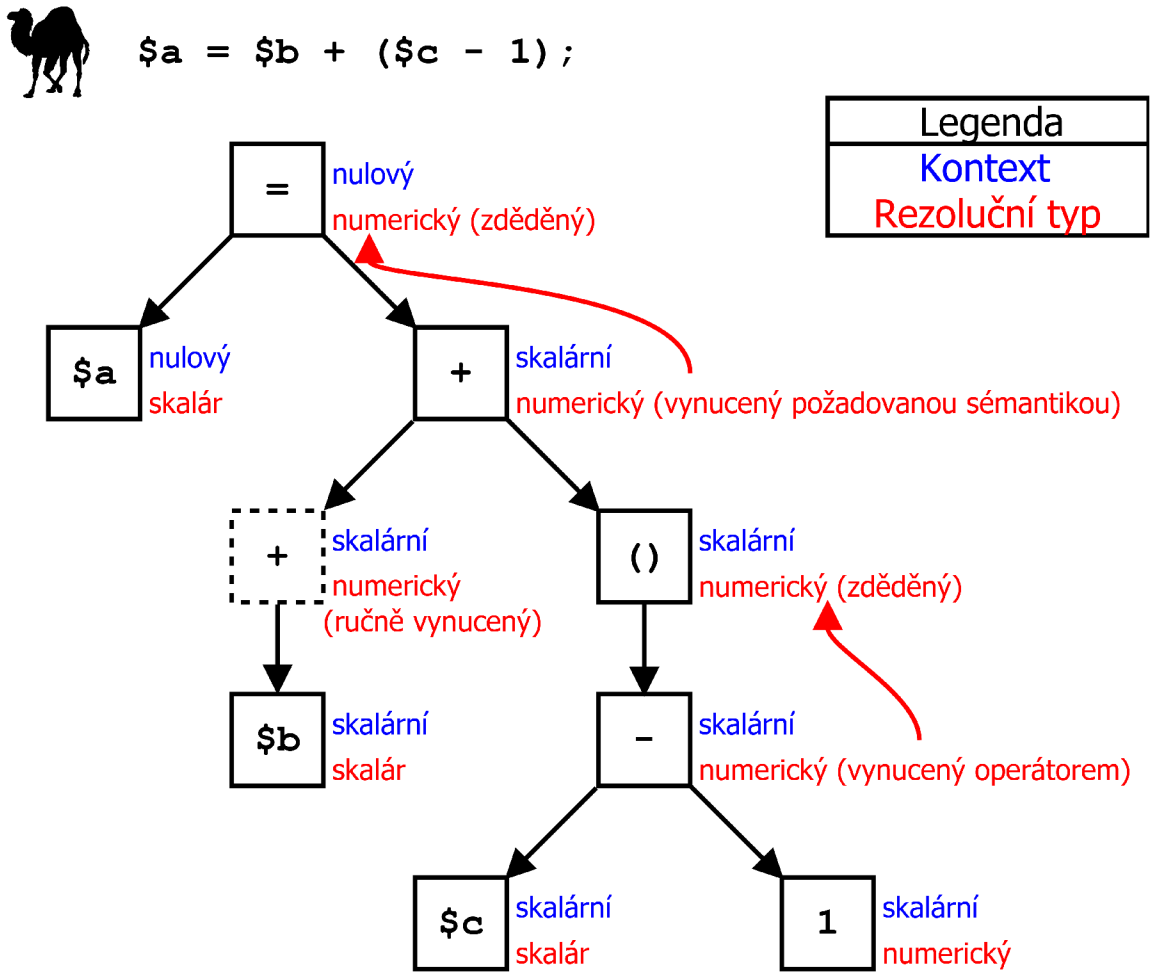
```
1 console.log(1 + +"2" + "\n");
```

Tyto dva případy jsou triviální obtížnosti a statická analýza jejich vlastnosti snadno odhalí, či optimalizace může provést jejich zkrácení již při překladu a tím zcela eliminovat tento problém.

Potíže však nastanou v přídadě, kdy místo literálů budou použity proměnné. Jelikož v případě jak jazyka vstupního, tak jazyka výstupního se jedná o jazyky dynamické, není jednoduše možno rozhodnout typ hodnoty v jenž bude daná proměnná v době běhu programu vyhodnocena. Jelikož zdrojový jazyk příliš nerozlišuje mezi typy skalárních proměnných. V tomto případě můžeme provést alespoň částečnou analýzu a ruční vynucení typů v cílovém jazyce přidáme-li každému výrazu a subvýrazu, tj. i uzlům reprezentujícím proměnné, další atribut – typ rezoluční hodnoty – jedná-li se o skalární hodnotu. Tento typ rezoluční hodnoty může být buď obecný skalár, numerická hodnota, nebo řetězcová hodnota.

Obecný skalár uvažujme tam, kde není jasné, v jaký rezoluční typ vyústí daný subvýraz. Tato hodnota atributu rezolučního typu indikuje naprostou nejistotu jaký z možných typů ekvivalentních skalárům zdrojového jazyka bude použit v jazyce cílovém pro reprezentaci hodnoty daného podvýrazu. Pokud je takováto hodnota použita v podvýrazu, jehož sémantika závisí na typu operandů, je nutno provést vynucení správných typů ručně. Stejně tak je nutno provést ruční vynucení typu v případě kdy je rezoluční typ znám, ale je špatný.

Některé operátory samy provedou vynucení rezolučního typu, takže při průchodu stromem je možné, že přesto že jako originální operandy byly použité hodnoty s neurčitým rezolučním typem, použitím operátorů dojde automaticky k vynucení rezolučního typu a není tak třeba tento typ opakovaně vynucovat, jak je znázorněno na obrázku č. 5.4



Obrázek 5.4: Příklad manuálního vynucení typů

Díky této analýze rezolučních typů bylo přidán operátor unární `+`. Tento operátor vynutí rezoluci skalární proměnné `$b`, která může v cílovém jazyce být reprezentována typem řetězcovým nebo numerickým. Pro úspěšný překlad operátoru `+`, jehož sémantika je numerické sčítání, je potřeba vynutit rezoluci typů v cílovém jazyce na typ numerický. Pokud by k tomu nedošlo, jak je znázorněno ve výpise č. 5.3 na prvním řádku, v případě že by byla proměnná `b` typu řetězec, došlo by ke změně sémantiky operátoru `+` cílového jazyka z numerického součtu na konkatenaci. Tomuto lze předejít „hrubou silou“, jak je znázorněno na druhém řádku, avšak toto vede na zbytečně složitý a nečitelný kód. Není potřeba unárního operátoru `+` před závorkami, které jsou přímým druhým operandem operátoru `+`, jelikož operátor `-`, použitý uvnitř závorek, vynutí sám numerický typ a není ho tedy nutno vynucovat při vyhodnocování subvýrazu závorek, jelikož ty pouze oddělují jednotlivé subvýrazy a nemění rezoluční typy. Optimální překlad je poté na třetím řádku, ten přidává pouze jeden unární operátor nutný k zajištění sémantiky operátoru sčítání.

```
1 a = b + (c - 1);
2 a = +b + +(c - 1);
3 a = +b + (c - 1);
```

Výpis 5.3: Možnosti překladu výrazu z obrázku č. 5.4

5.3 Překlad

Po dokončení těchto statických analýz lze přistoupit k samotnému pokusu o překlad. Zaměříme-li se pouze na imperatvní paradigma obou jazyk, zjistíme, že se jeden druhému velmi podobají. Pro valnou většinu operátorů jazyka vstupního existuje sémantický ekvivalent v jazyce cílovém. Ne vždy se ale jedná o sémantický ekvivalent ve formě operátoru.

Uvážme tedy vestavěný pojmenovaný operátor jazyka Perl `shift`. Jedná se o operátor unární s jedním operandem typu pole. Sémantika tohoto operátoru je hodnota prvního prvku pole a odstranění tohoto prvku z daného pole.

```
1 my $prvni_prvek = shift @pole;
```

V jazyce cílovém neexistuje operátor se stejnou sémantikou. Existuje však metoda *primitivního typu array*, `shift()`, která se chová stejným způsobem, jako operátor `shift` ve zdrojovém jazyce. Výraz výše lze tedy přeložit do cílového jazyka následujícím způsobem:

```
1 var prvni_prvek = pole.shift();
```

Obdobně poté můžeme implementovat další operátory zdrojového jazyka, jako jsou `unshift`, `pop` nebo `push` či `keys`.

Pokud však neexistuje ekvivalent přesný, vždy je možno implementovat danou funkcionalitu pomocí funkce, která může být součástí knihovny funkcí zajišťujících kompatibilitu. Dobrým příkladem je operátor vstupního jazyka `print`. Tento operátor nemá obdoby ani v prostředcích jazyka JavaScript, ani v prostředcích poskytovaných prostředím interpretu (*Node.js*). Je tedy nutno implementovat kompatibilní funkci ručně. Příklad takovéto implementace je ve výpise č. 5.4. Tato implementace je však velmi omezená faktem, že pracuje pouze se standardním výstupem.

Obdobné způsoby jak obejít rozdíly mezi vstupním a výstupním jazykem se dají uplatnit v mnoha dalších případech. Jedním z nich je například využití operátoru `eval`. Ve vstupním jazyce je často používaným idiomem konstrukce znázorněná ve výpise č. 5.5. Tato konstrukce

```
function print(...args){
  for (var arg of args){
    process.stdout.write(arg.toString())
  }
}
```

Výpis 5.4: Ukázková implementace operátoru `print` v JavaScriptu

se chová podobným způsobem jako například `try ...catch` výrok v jazyce cílovém. Přímý překlad by však byl velmi náročný. Blok, který je ve vstupním jazyce operandem operátoru `eval`, odpovídá bloku, který by byl součástí výroku `try` v jazyce JavaScript. Ovšem zdrojový jazyk informace o zachycené výjimce ukládá do vestavěných proměnných, jejichž použití v jazyce cílovém je velmi obtížné (zahrnovalo by totiž vytvoření globálních proměnných), zatímco struktura, která by odpovídala bloku `catch`, chybí. Nelze tedy provést překlad, který by odpovídal idiomům cílového jazyka. Lze však ale provést překlad který sice těmto idiomům odpovídat nebude, zato však bude funkční, viz výpis č. 5.6.

```
eval{
  # Kod ktery muze skoncit s chybou
  die "Chyba";
}
if ($@ eq "Chyba"){ # Kontrola globalni promenne s chybovou hlaskou
  # Obsluha chyby
}
```

Výpis 5.5: Idiom jazyka Perl pro zachytávání výjimek

```
var err; //Globalni promenna

try {
  // Kod produkujici vyjimky
  throw "Chyba"
}
catch (e) {
  err = e //Zachyceni vyjimky a jeji prirazeni do globalni promenne
}
// Obsluha vyjimky pomoci kontroly globalni promenne
if (err == "Chyba"){
  // Obsluha
}
```

Výpis 5.6: Implementace odpovídající použití operátoru `eval`.

Podobně se pak lze zachovat v případě subrutin. Subrutiny v Perlu nemají signatury, tzn. nemají pevně dané počty, pořadí, ani jména argumentů. Je tedy na programátorovi, aby subrutiny invokoval se správným počtem argumentů, ve správném pořadí a aby zajistil, že subrutina zkontroluje počet a typ argumentů, které dostala, a v případě že jsou argumenty nebo jejich počet nepřijatelné, zachová se za běhu adekvátně.

Toto chování se velmi těžko analyzuje ze statického hlediska. Teoreticky je možno použít některých optimalizačních technik pro zjištění proměnných, do nichž budou extrahovány argumenty z výchozího pole argumentů, není však možné přeložit subrutinu do cílového ja-

zyka s pevně danými argumenty, jelikož subrutiny mohou stále pracovat s polem argumentů jako s jedním celkem. Pro takovýto překlad by bylo potřeba velmi detailní analýzy AST.

Co však lze velmi jednoduše, je emulovat chování subrutin Perlu v samotném cílovém jazyce, jak je znázorněno ve výpise č. 5.7. Tento způsob používá operátor cílového jazyka `...` pro „zabalení“ argumentů předaných funkci do pole pojmenovaného `_`, obdobně jak je tomu v Perlu. K tomuto poli je pak možno se chovat stejnými způsoby, jako ve zdrojovém jazyce a není tedy třeba složitých analýz a úprav AST. Tato konstrukce však, byť zachovává funkcionalitu, není idiomem cílového jazyka. To může mít za následek horší čitelnost a udržovatelnost generovaného kódu.

```
1 funciton subroutine(..._){
2   var arg1 = _.shift()
3   // Telo subrutiny
4 }
```

Výpis 5.7: Implementace funkce JavaScriptu obdobná subrutinám Perlu

5.4 Omezení překladu

Pokud porovnáme složitější paradigmata vstupního a výstupního jazyka, jako jsou objektově orientované či funkcionální programování, zjistíme, že zde se odlišnosti mezi zdrojovým jazykem a jazykem cílovým značně kupí. Uvažme jako příklad objektově orientované programování. Vstupní jazyk spoléhá při definici tříd objektů na speciální moduly – balíčky – emulující chování tříd v jiných objektově orientovaných jazycích, včetně dědičnosti. Výstupní jazyk naopak používá striktně prototypový přístup k vytváření a definci objektů a jejich vnitřní struktury, přestože existují syntaktické prostředky jak toto chování ovládat pomocí konstrukcí podobných třídám. Toto je jednom jedna z mála překážek stojících v cestě překladu objektově orientovaných programů. Byť je tedy použití objektů v imperativně orientovaných programech možné (rozumějte volání metod již existujících objektů a přístup k jejich atributům), není jednoduše možno překládat definice tříd a vytváření instancí ze zdrojového jazyka do jazyka cílového, bez nutnosti extenzivní a komprehenzivní analýzy AST zdrojového jazyka. A ani potom nelze zaručit úspěšný překlad. Tato práce tedy tento problém dále zrpacovávat nebude jakožto složitý nad její rámec, přestože teoreticky možná řešitelný.

Kapitola 6

Implementace

V této kapitole bude popsána implementace navrženého překladače. Jedná se o implementaci jako *proof of concept*, poukazující na nedostatky či omezení vyplývající z návrhu.

6.1 Přední část

Účelem přední části implementovaného transpilátoru je číst, analyzovat a kontrolovat vstupní kód v jazyce Perl a z něj následně skládat AST. Pro rychlejší dosažení tohoto cíle bylo použito již existujících nástrojů pro analýzu kódu v jazyce Perl, konkrétně knihovny dostupné pro jazyk Perl – PPI[1]. Tato knihovna umožnuje analýzu vstupního proudu znaků, jeho dělení na jednotlivé tokeny a základní struktury jazyka Perl. Jejím výstupem je stromová struktura, kterou dokumentace této knihovny nazývá *PDOM* (z anglického *Perl Document Object Model*). Jedná se o strukturu podobnou AST, nikoliv však tolik kompletní či komprehenzivní. PDOM rozlišuje makrostruktury jazyka Perl, jako jsou jednotlivé výroky a složené výroky. Rozlišuje také jednotlivé tokeny. Co již však v PDOM chybí je vnitřní struktura jednotlivých výrazů. PDOM nereflektuje totiž precedenci či použití operátorů, a proto je potřeba dodatečné syntaktické analýzy za účelem získání co nejkompletnějšího AST.

Jelikož *PPI* jest knihovnou jazyka Perl, je první fáce překladu prováděna pomocí skriptu pro jazyk Perl. Tento skript, `PPI_XML_dump.pl`, nejprve použije knihovnu *PPI* pro vytvoření PDOM ze vstupního souboru, a následně všechny sémanticky podstatné prvky takto vytvořeného PDOM zapíše do souboru ve formátu XML. Příklad obsahu takového souboru naleznete v příloze, ve výpise č. A.1.

Tento soubor je následně využíván hlavním programem psaným v jazyce Python. Tento program provádí dodatečnou analýzu za účelem získání AST, následně analyzuje samotný AST a provádí překlad.

Pro dodatečnou analýzu je dle návrhu použita varianta algoritmu stoupání precedencí. Tato konkrétní implementace byla značně inspirována webovým článkem *Simple Top-Down Parsing in Python*[1].

Sestavení stromu začíná extrakcí struktury PDOM rekurzivním čtením. Narazí-li se však na výraz, u nějž je třeba dokončit analýzu, aby bylo možno vytvořit AST, přichází na řadu již zmiňovaný algoritmus stoupání precedencí. Jádrem této konkrétní implementace tohoto algoritmu je funkce `parse_expression`, viz výpis č. 6.1.

Argumentem této funkce je precedenční síla operátoru nadřazenému aktuálně analyzovaného podvýrazu, pokud žádný není, výchozí hodnota této síly je 0. Na základě této síly je

[1] metacpan.org/pod/PPI

```
def parse_expression(right_binding_power: int = 0) -> Expression:
    tokenizer: Tokenizer = Globals.TOKENIZER.get()
    token = tokenizer.token
    tokenizer.next()
    left: Expression = token.null_denotation()
    while right_binding_power < tokenizer.token.left_binding_power:
        token = tokenizer.token
        tokenizer.next()
        left = token.left_denotation(left)
    return left
```

Výpis 6.1: Implementace hlavní funkce algroritmu stoupání precedencí

vyhodnocována precedence operátorů. Do podvýrazu budou zahrnuty pouze ty operátory, jejichž síla je vyšší než síla vyžadovaná pro překonání precedenční, nebo chcete-li vazebé, síly aktuálního nadřazeného operátoru.

Pro pochopení fungování této funkce však ještě potřebujeme znát strukturu objektů tokenů. Tato struktura je popsána ve výpise č. 6.2.

```
class Token:
  left_binding_power = 0;  # Precedencni sila

  def null_denotation(self):
    raise SyntaxError

  def left_dentoation(self, left):
    raise SyntaxError
```

Výpis 6.2: Základní struktura tokenů

Každý typ tokenu dědí od této třídy (v implementaci je tato třída nazvána `Parseable`, zde však bude pro konzistenci zvána `Token`). Každý token tedy má tři základní vlastnosti:

- atribut `left_binding_power` – značí precedenci daného tokenu při jeho nalezení na místě operátoru v rámci rozhodování v hlavní funkci analyzátoru
- metoda `null_denotation` – řeší roli tokenu, v případě že je použit na začátku subvýrazu
- metoda `left_denotation` – řeší roli tokenu, v případě že se nachází uvnitř subvýrazu, tzn. je již k dispozici levá strana

Za povšimnutí stojí neexistence výchozí implemetace metod `null_denotation` a `left_denotation` ve třídě `Token`. Jedná se o formu kontroly chyb. Nastane-li situace, kdy je volána neimplementovaná metoda, znamená to, že daný token byl nalezen na neočekávaném místě. Vysvětleme to na příkladu. Výpis č. 6.3 představuje zjednodušenou implementaci třídy tokenu literálu či proměnné. Takovýto token by mělo být možno použít pouze jako součást podvýrazu, nikoliv jako operátor. Nemá tudíž implementovanou metodu `left_denotation`, jelikož neočekává že by měl být použit v situaci, kdy je nadřazen jinému subvýrazu. Metodu `null_denotation` však již implementuje, jelikož je očekáváno, že tento token bude použit na začátku podvýrazu, respektive bude tvořit atomární jednotku podvýrazu – term. Hodnota atributu `left_binding_power` je taktéž ponechána na výchozí hodnotě, jelikož se

nepočítá s využitím tohoto tokenu jako operátoru. Pokus o to by totiž vedl na syntaktickou chybu.

```
class TokenLiteral(Token):
  def null_dentoation(self):
    return ASTNodeLiteral(value=self.value)
```

Výpis 6.3: Příklad implementace tokenu literálů

Jak je také patrné z výpisu č. 6.3, metody `null_denotation` a `left_denotation` mají za úkol vrátit význam daného tokenu v rámci AST, tj. podstrom, který je podřízený danému uzlu, který vyplývá z konkrétního tokenu. Jelikož literál je tokenem s konrétní hodnotou, návratovou hodnotou jeho metody `null_denotation` je listový uzel AST.

```
class TokenBinaryOperator(Token):
  left_binding_power = 10
  def left_dentoation(self, left):
    right = parse_expression(self.left_binding_power)
    return ASTBinaryOperator(left, right)  # ASTBinaryOperator je implementace tridy pro
    konkretni binarni operator, zde pouze pro prehlednost
```

Výpis 6.4: Příklad implementace tokenu binárního operátoru

Pro srovnání uvažme implementaci třídy pro binární operátory, viz výpis č. 6.4. Můžeme si všimnout zaprvé již nastacené precedenční síly, dle precedence daného operátoru, za druhé, implementované metody `left_denotation`. V rámci této metody obdrží daný token svůj levý podvýraz. Jedná se však o binární operátor, a proto dojde k rekurzivnímu volání funkce `parse_expression`. Toto volání bude provedeno s předáním své precedenční síly, efektivně omezující přijímaný podvýraz na takový, ve kterém se vyskytují pouze termy a operátory s vyšší precedencí.

```
class TokenUnaryPrefixOperator(Token):
  left_binding_power = 20
  def null_denotation(self):
    right = parse_expression(self.left_binding_power)
    return ASTNodeUnaryPrefixOperator(right)  # Ditto ASTNodeBinaryOperator


class TokenUnaryPostfixOperator(Token):
  left_binding_power = 20
  def left_denotation(self, left):
    return ASTNodePostfixOperator(left)  # Ditto
```

Výpis 6.5: Příklad implementace tokenů prefixového a postfixového operátoru

Obdobně lze implementovat například prefixové a postfixové unární operátory, jak je demonstrováno ve výpise č 6.5. Token postfixového operátoru stejně jako token operátoru binárního implementuje metodu `left_denotation`, jelikož očekává levou stranu. Avšak v jejím rámci už neprovádí rekurzivní volání parsovací funkce, jelikož pravou stranu nepotřebuje. Proto rovnou vrací sobě odpovídající uzel AST. Naopak prefixový unární operátor implementuje metodu `null_denotation`, protože levou stranu neočekává. Narozdíl však od tokenů literálů, v rámci této metody provede rekurzivní volání parsovací funkce za účelem získání své pravé strany a až pak vrátí sobě odpovídající uzel AST.

O instancování správných tříd tokenů na základě vstupu ze souboru s PDOM ve formátu XML se stará „tokenizér“. Ten extrahuje již předanalyzované tokeny z PDOM a předává je parsovací funkci ve správném formátu.

6.2 Statická analýza

Každý uzel AST má vždy jeden atribut společný pro všechny subtypy uzlů AST, ať už se jedná o operátory či literály, a tím je kontext, ve kterém se daný uzel nachází, a ve kterém se daný případný subvýraz bude vyhodnocovat. Implementačně je kontext realizován pomocí tzv. vlastnosti (z anglického *property*). Jedná se o virtuální atribut třídy v jazyce Python. Tento atribut je dán dvěma metodami. Metodou pro získání jeho hodnoty (*getter*) a metodou pro nastavení jeho hodnoty (*setter*). Použití těchto metod umožňuje pohodlnou implementaci propagace kontextu abstraktním syntaktickým stromem. Metoda pro získání hodnoty kontextu aktuálního uzlu očekávatelně pouze vrátí vnitřní hodnotu kontextu daného uzlu. Metoda pro nastavení hondoty však zajišťuje více, než pouhé nastavení samotné vnitřní hodnoty aktuálního uzlu, nýbrž propaguje kontext uzlům sobě podřízeným. Tato propagace je z pravidla čistá, tzn. kontext který dostane daný uzel od uzlu nadřazeného bude propagován uzlům podřízeným, avšak existují i speciální případy, například do výrazů podmínek složenýc výroků bude vždy propagován skalární kontext a do výrazů na pravých stranách přiřazení bude propagován kontext závisející na straně levé daného přiřazení.

Každý uzel AST při svém vytvoření (tj. při instancování dané třídy) začíná s *neznámým* kontextem. Jedná se o speciální hondotu kontextu, která nemá žádnou sémantiku a neměla by se při vyhodnocování sémantiky nikdy objevit. Pokud k tomu dojde, značí to s největší pravděpodobností chybu v propagaci kontextu. Analýza kontextu tedy probíhá přiřazením nulového kontextu kořeni AST a následnou rekurzivní propagací.

Uveďme tedy praktický příklad takovéhoto analyzovaného stromu. V příloze je přiložen jednoduchý program pro iterativní výpočet prvních 40 prvků Fibonacciho posloupnosti, viz výpips B.1. AST tohoto programu je znázorněn ve výpise č. 6.6. Jedná se o přímý výstup implementovaného překladače. Jednotlivé uzly jsou vyznačeny prvně jménen třídy objektu daného uzlu, která koresponduje zpravidla s významem daného uzlu, druhak kontextem, který daný uzel získal při rezoluci kontextu uzavřeným v ostrých závorkách, a nejposledněji volitelně hodnotou daného uzlu, tj. hodnotou literálu nebo jménem proměnné.

6.3 Zadní část

Po sestavení a analýze AST přichází na řadu samotný překlad. Ten je taktéž řešen rekurzivně pomocí metody základní třídy uzlu AST – `translate`. Jednotlivé třídy uzlů mohou tuto metodu volně nahrazovat vlastními implementacemi svého vlastního překladu. Přičemž při překladu jim je k dispozici celý jejich podstrom a kontext jim přiřazený. Příkladem uzlu, který nechá svůj překlad ovlivňovat kontextem je uzel typu `Symbol`. Jedná se o uzel reprezentující proměnnou, a jako takový nese také vnitřní typ, stejně jako proměnná v Perlu, tj. skalár, pole nebo asociativní pole. Pokud se nachází Symbol s typem pole ve skalárním kontextu, bude jeho překladem invokace jeho atributu `length` v cílovém jazyce. Pokud se však nachází v kontextu pole, bude jeho překladem invokace celé jeho hodnoty, tzn. pouze jeho jméno. Jiné uzly zakládají svůj překlad na překladu svých podstromů a překladu vlastním. Například uzly cyklů typu *while* zakládají svůj překlad na klíčovém slovu `while`, na překladu podstromu podmínky a na překladu těla cyklu.

Překlad tedy již výše zmíněného ukázkového programu B.1 je k nahlédnutí v příloze, ve výpise č. B.2. První nápadnou vlastností přeloženého programu je implementace funkce `print`, která se znaží reflektovat sémantiku operátoru `print` jazyka zdrojového. Tato funkce je však značně zjednodušená.

Druhou velmi nápadnou vlastností daného programu je struktura použitá na řádku č. 5, kde dochází k rozbalení pole `fib` za účelem jeho prodloužení a následného uložení do stejné proměnné. Tato konstrukce, byť ekvivalentní použití operátoru `push` v jazyce zdrojovém, nebo stejnojmenné metody v jazyce cílovém, je v Perlu překvapivě častá. Využívá se totiž zejména za účelem skládání polí a jednotlivých prvků v pole větší:

```
1 my @bigger = (@smaller, $filler, @another);
```

JavaScript však takovýchto metod používá zřídka.

Další zajímavý důsledek překladu jiného programu je patrný ve výpise č. B.4. Zde je vidět porušení dalšího idiomu jazyka JavaScript, a to jest vágní signatura funkce `factorial`. Toto nejenže zhoršuje čitelnost a udržovatelnost kódu, ale navíc interferuje se snahami interpretu cílového jazyka o optimalizaci analýzy, překladu a spouštění.

Toto jsou tedy pouze vybrané problémy pramenící z již už tak velmi omezené překládací schopnosti implementovaného interpretu. Je tedy na povážení, kolik dalších problému by přineslo jeho rozšíření.

```
Program <Void>
+-Statement <Void>
| --OperatorAssignment <Void>
|   +-OperatorMY <Scalar>
|   |--Symbol <Scalar> '$last1'
|   --Number <Scalar> '0'
+-Statement <Void>
| --OperatorAssignment <Void>
|   +-OperatorMY <Scalar>
|   | --Symbol <Scalar> '$last2'
|   --Number <Scalar> '1'
+-Statement <Void>
| --OperatorAssignment <Void>
|   +-OperatorMY <List>
|   | --Symbol <List> '@fib'
|   --VarListStructure ParensStructure <List>
|     +-Symbol <List> '$last1'
|     --Symbol <List> '$last2'
+-WhileStatement <Void>
| +-Expression <Scalar>
| | --OperatorLessThan <Scalar>
| |   +-Symbol <Scalar> '@fib'
| |   --Number <Scalar> '40'
| --StatementList <Void>
|   +-Statement <Void>
|   | --OperatorAssignment <Void>
|   |   +-Symbol <List> '@fib'
|   |   --ListStructure ParensStructure <List>
|   |     +-Symbol <List> '@fib'
|   |     --OperatorPlus_bin <List>
|   |       +-Symbol <List> '$last1'
|   |       --Symbol <List> '$last2'
|   +-Statement <Void>
|   | --OperatorAssignment <Void>
|   |   +-OperatorMY <Scalar>
|   |   | --Symbol <Scalar> '$temp'
|   |   --OperatorPlus_bin <Scalar>
|   |     +-Symbol <Scalar> '$last1'
|   |     --Symbol <Scalar> '$last2'
|   +-Statement <Void>
|   | --OperatorAssignment <Void>
|   |   +-Symbol <Scalar> '$last1'
|   |   --Symbol <Scalar> '$last2'
|   --Statement <Void>
|     --OperatorAssignment <Void>
|       +-Symbol <Scalar> '$last2'
|       --Symbol <Scalar> '$temp'
+-WhileStatement <Void>
| +-Expression <Scalar>
| | --Symbol <Scalar> '@fib'
| --StatementList <Void>
|   --Statement <Void>
|     --OperatorPRINT <Void>
|       +-OperatorSHIFT <Void>
|       | --Symbol <List> '@fib'
|       --Quote <Void> ' '
--Statement <Void>
  --OperatorPRINT <Void>
    --Quote <Void> '\n'
```

Výpis 6.6: AST programu B.1

Kapitola 7

Závěr

Implementovaný transpilátor je schopný analyzovat a sestavit AST omezené podmnožiny jazyka Perl. Nad tímto AST následně provádí statickou analýzu kontextu a posléze přistupuje k překladu. Neimplementuje sice všechny metody analýzy popsané v předchozí kapitole a podporuje překlad pouze velmi omezené podmnožiny jazyka Perl, ovšem i tak je zřejmé, že přes svou funkčnost, která lze teoreticky zaručit, programy překládané navrženými způsoby nedodržují idiomatickou strukturu cílového jazyka.

Přeložitelná podmnožina jazyka Perl je omezena na většinu binárních a unárních operátorů, které nepracují s regulárními výrazy, na definici a deklaraci lokálních proměnných (pouze po jedné), na smyčky typu *while*, na podmínky typu *if... elsif... else* a subrutiny. Důvodem pro takto přísná omezení je exponenciálně vzrůstající komplexita interakcí jednotlivých přidávaných prvků jazyka Perl a to jak při sestavování AST, tak při statické analýze a překladu. Přestože by bylo teoreticky možné implementaci rozšířit na valnou vetšinu imperativních prostředků jazyka Perl, takovýto počin by vyžadoval velmi velké množství úsilí, ať už při vytváření nebo testování vytvářeného transpilátoru či vytváření knihovny cílového jazyka pro zajištění kompatibility operátorů, jež v jazyce cílovém schází.

Přes veškerou tuto snahu by výsledkem byl sice funkční kód cílového jazyka, avšak tento kód by s největší pravděpodobností porušoval valnou většinu idiomů cílového jazyka a jako takový by byl velmi těžko čitelný, udržovatelný či upravitelný. Praktická užitečnost přeloženého kódu by tedy bez velkých zásahů programátora, nebo bez značného rozšíření analýzy a zavedení značných programatických úprav vnitřního AST, byla taktéž značně omezena, nehledě na fakt, že techniky používané pro optimalizaci běhu programu cílového jazyka značně závisí na použití správných idiomatických struktur cílového jazyka. Tudíž takovýmto neidiomatickým překladem může dojít ke ztrátě výhod poskytovaných cílovým jazykem, jež byly důvod pro překlad samotný.

Přestože oba jazyky, jak cílový, tak vstupní, podporují velmi podobné přístupy k programování i stejná programovací paradigmata, nejsou tyto přístupy totožné, či dostatečně podobné na to, aby bylo možno mezi nimi volně překládat bez porozumění významu kódu jako celku, nebo alespoň významu idiomatických struktur. Tuto skutečnost ještě více umocňuje fakt, že Perl je jazykem velmi shovívavým s velmi voulnou syntax a velkou škálou použitelných struktur a jazykových obratů.

Lepších výsledků by tedy bylo možno dosáhnout podrobnou analýzou jazykových prostředků, struktur a obratů používaných v překládané programové základně a omezení implementovaného transpilátoru pouze na používané struktury. Takovéto omezení však velmi zúží použitelnost tímto způsobem implementovaného transpilátoru, výměnou za mírné zjednodušení implementace. Zůžení použitelnosti těchto rozměrů by s největší pravděpodobností

vedlo na jednoúčelnost výsledného transpilátoru a jeho nepoužitelnost pro překlady jiných programových základen. Zde je poté na pováženou, zda je zdrojová programová základna dostatečně konzistentní v použití těchto jazykových prostředků, idiomů a obratů, aby před pokusem o překlad nemuselo docházet k masivním úpravám, nebo aby omezení na používané idiomy, prostředky a obraty přineslo dostatečné zjednodušení komplexity navrhovaného transpilátoru. Zvážena by v takových případech měla být i cena pokusu o implementaci takového transpilátoru, ať už se jedná o cenu vynaloženou v nákladech či úsilí programátora.

Narozdíl od konkrétní implementace, teoretické základy návrhu transpilátoru, by, dle názoru autora, mělo být možné využít, nebo lespoň použít jako inspiraci při vytváření transpilátoru jiného páru vstupního a výstupního jazyka, ať už se bude jednat o dva existující jazyky, či dvojici jazyka existujícího a jeho rozšíření.

Dalšími možnostmi pro implementaci systému s podobným cílem by mohlo být třeba využití umělé inteligence a jazykových modelů, které jsou v současné době na vzestupu. Jejich schopnosti zachovat a řídit se dle daného kontextu by se mohla projevit jako velmi užitečná při této úloze.

V neposlední řadě se také naskytuje možnost poloautomatizovaného překladu, či člověkem asistovaného překladu. Takovýto překlad by mohl být uskutečněn například jednoduchým přepisem zdrojového kódu tak, aby byla použita klíčová slova, struktury a operátory cílového jazyka s tím, že kontrolu a rezoluci rozdílů sémantik provede programátor, který je s danou problematikou seznámen lépe, než kdy jakýkoliv automatizovaný systém by mohl v dnešní době být. Takový přístup přináší výhody jak ze strojového překladu, tak z manuální reimplementace. Takovýto poloautomatický systém může generovat i nevalidní kód, s tím že validitu a validaci zajistí programátor. Tento přístup může pomoci s neduhy které trápí ruční reimplementace, jako jsou repetetivnost takového snažení či překlepy ve výrazech nebo názvech proměnných či přehlédnutí důležitých vlastností zdrojového kódu. Přidaným bonusem je poté samotná implementace takového nástroje a její jednoduchost oproti plně automatizovanému řešení. Takový poloautomatizovaný nástroj by nemusel provádět všechny analyzační metody popsané v této práci a místo toho delegovat tyto úlohy na programátora. S největší pravděpodobností tímto způsobem vznikne nekompletní a čistě jednoúčelový nástroj, avšak zvážíme-li množství ušetřeného úsilí při vykonávání primárního úkolu, jímž je překlad nebo chcete-li reiplementace programové základny, které nám tento, byť jednoúčelný, nástroj poskytne společně s jednoduchostí jeho implementace, předčí zvýšení efektivity překladu tímto způsobem efektivitu kteréhokoliv jednoúčelného plně automatického překladače.

Literatura

[1] *Simple Top-Down Parsing in Python* [online]. effbot [cit. 2023-4-27]. Dostupné z: https://effbot.org/simple-top-down-parsing-in-python/.

[2] Aho, A. V., Lam, M. S., Sethi, R. a Ullman, J. D. *Compilers: principles, techniques and tools*. 2. vyd. Addison Wesley, 2007. ISBN 0-321-48681-1.

[3] Bissyandé, T. F., Thung, F., Lo, D., Jiang, L. a Réveillère, L. Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects. In: *2013 IEEE 37th Annual Computer Software and Applications Conference*. 2013, s. 303–312. DOI: 10.1109/COMPSAC.2013.55.

[4] Christiansen, T., Wall, L., Orwant, J. et al. *Programming Perl: Unmatched power for text processing and scripting*. "O'Reilly Media, Inc.", 2012. ISBN 978-0-596-00492-7.

[5] Ecma International. *ECMAScript® 2015 Language Specification* [online]. 6. vyd. Geneva: Ecma International, červen 2015 [cit. 2023-4-20]. Dostupné z: https://262.ecma-international.org/6.0/.

[6] Glass, R. L. An elementary discussion of compiler/interpreter writing. *ACM Computing Surveys (CSUR)*. ACM New York, NY, USA. 1969, sv. 1, č. 1, s. 55–77.

[7] Grune, D., Van Reeuwijk, K., Bal, H. E., Jacobs, C. J. a Langendoen, K. *Modern compiler design*. 2. vyd. Springer Science & Business Media, 2012. ISBN 978-1-4614-4699-6.

[8] Klint, P. Interpretation techniques. *Software: Practice and Experience*. Wiley Online Library. 1981, sv. 11, č. 9, s. 963–973.

[9] Kulkarni, R., Chavan, A. a Hardikar, A. Transpiler and it's Advantages. *International Journal of Computer Science and Information Technologies*. Citeseer. 2015, sv. 6, č. 2, s. 1629–1631.

[10] Liang, C. Programming language concepts and Perl. *Journal of Computing Sciences in Colleges*. 2004, sv. 19, č. 5, s. 193–204.

[11] McIlroy, R. Firing up the Ignition interpreter. *V8 JavaScript Engine* [online]. 23. srpna 2016 [cit. 2023-4-20]. Dostupné z: https://v8.dev/blog/ignition-interpreter.

[12] MDN Contributors. *JavaScript* [online]. Mozilla, 2023-4-5. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[13] Norvell, T. *Parsing Expressions by Recursive Descent* [online]. 1999 [cit. 2023-4-27]. Dostupné z: https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm#climbing.

[14] Perl 5 Porters. *Perl 5.36.0 Documentation* [online]. 5.36.0. [cit. 2023-4-21]. Dostupné z: https://perldoc.perl.org.

[15] Pratt, V. R. Top down operator precedence. In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1973, s. 41–51.

[16] Verwaest, T. Lazy parsing. *Blazingly fast parsing* [online]. 15. dubna 2019 [cit. 2023-4-20]. Dostupné z: https://v8.dev/blog/scanner.

[17] Verwaest, T. Optimising the scanner. *Blazingly fast parsing* [online]. 25. března 2019 [cit. 2023-4-20]. Dostupné z: https://v8.dev/blog/scanner.

[18] Wall, L. et al. *The Perl programming language*. Prentice Hall Software Series, 1994.

Příloha A

PDOM XML

```
<?xml version="1.0" encoding="UTF-8"?>
<program file=".\perl_program.pl">
  <Document ppi-type="PPI::Document">
    <Statement ppi-type="PPI::Statement::Variable">
      <Token ppi-type="PPI::Token::Word" Token-type="Word" line="1" col="1">my</Token>
      <Token ppi-type="PPI::Token::Symbol" Token-type="Symbol" line="1" col="4">@arr</Token
    >
      <Token ppi-type="PPI::Token::Operator" Token-type="Operator" line="1" col="9">=</
    Token>
      <Structure ppi-type="PPI::Structure::List" Struct-type="List">
        <Statement ppi-type="PPI::Statement::Expression">
          <Token ppi-type="PPI::Token::Number" Token-type="Number" line="1" col="12">1</
    Token>
          <Token ppi-type="PPI::Token::Operator" Token-type="Operator" line="1" col="13">..
    </Token>
          <Token ppi-type="PPI::Token::Number" Token-type="Number" line="1" col="15">3</
    Token>
        </Statement>
      </Structure>
      <Token ppi-type="PPI::Token::Structure" Token-type="Structure" line="1" col="17">;</
    Token>
    </Statement>
    <Statement ppi-type="PPI::Statement">
      <Token ppi-type="PPI::Token::Magic" Token-type="Symbol" line="3" col="1">$"</Token>
      <Token ppi-type="PPI::Token::Operator" Token-type="Operator" line="3" col="4">=</
    Token>
      <Token ppi-type="PPI::Token::Quote::Double" Token-type="Quote" line="3" col="6">" "</
    Token>
      <Token ppi-type="PPI::Token::Structure" Token-type="Structure" line="3" col="9">;</
    Token>
    </Statement>
    <Statement ppi-type="PPI::Statement">
      <Token ppi-type="PPI::Token::Word" Token-type="Word" line="5" col="1">print</Token>
      <Token ppi-type="PPI::Token::Symbol" Token-type="Symbol" line="5" col="7">@arr</Token
    >
      <Token ppi-type="PPI::Token::Structure" Token-type="Structure" line="5" col="11">;</
    Token>
    </Statement>
    <Statement ppi-type="PPI::Statement">
      <Token ppi-type="PPI::Token::Word" Token-type="Word" line="7" col="1">print</Token>
      <Token ppi-type="PPI::Token::Word" Token-type="Word" line="7" col="7">shift</Token>
      <Token ppi-type="PPI::Token::Symbol" Token-type="Symbol" line="7" col="13">@arr</
    Token>
      <Token ppi-type="PPI::Token::Operator" Token-type="Operator" line="7" col="18">&lt;&
    lt;</Token>
      <Token ppi-type="PPI::Token::Number" Token-type="Number" line="7" col="21">1</Token>
      <Token ppi-type="PPI::Token::Structure" Token-type="Structure" line="7" col="22">;</
    Token>
    </Statement>
  </Document>
</program>
```

Výpis A.1: Příklad obsahu XML souboru s PDOM

Příloha B

Jednoduché skripty v jazyce Perl a jejich překlady

```
#!/usr/bin/perl
use strict;
use warnings FATAL => 'all';

my $last1 = 0;
my $last2 = 1;
my @fib = ($last1, $last2);
while (@fib < 40){
    @fib = (@fib, $last1 + $last2);
    my $temp = $last1 + $last2;
    $last1 = $last2;
    $last2 = $temp;
}
while (@fib){
    print shift @fib, " ";
}
print "\n"
```

Výpis B.1: Program pro iterativní výpočet prvních 40 Fibonacciho čísel

```
var last1 = 0;
var last2 = 1;
var fib = [last1, last2];
while (fib.length < 40) {
  fib = [...fib, last1 + last2];
  var temp = last1 + last2;
  last1 = last2;
  last2 = temp;
}
while (fib.length) {
  print(fib.shift(), " ");
}
print("\n");



// Compatibility function
function print(...args){
  for (var arg of args){
    process.stdout.write(arg.toString())
  }
}
```

Výpis B.2: Překlad programu B.1

```
use strict;
use warnings FATAL => 'all';

sub factorial{
    my $factor = shift @_;
    my $ret = 1;
    while ($factor > 0) {
        $ret = $ret * $factor--;
    }
    return $ret;
}

print factorial(7);
```

Výpis B.3: Program pro iterativní výpočet faktoriálu pomocí funkce

```
function factorial (..._){
  var factor = _.shift();
  var ret = 1;
  while (factor > 0) {
    ret = ret * factor--;
  }
  return ret;
}

print(factorial(7));



// Compatibility functions
function print(...args){
  for (var arg of args){
    process.stdout.write(arg.toString())
  }
}
```

Výpis B.4: Překlad programu B.3