SŠPU
STŘEDNÍ ŠKOLA PRŮMYSLOVÁ
A UMĚLECKÁ, OPAVA

ZÁVĚREČNÁ STUDIJNÍ PRÁCE

dokumentace

Programovací jazyk Ruda

Autor: Daniel Antoš

Obor: 18-20-M/01 INFORMAČNÍ TECHNOLOGIE
se zaměřením na počítačové sítě a programování

Třída: IT4

Školní rok: 2023/24

Prohlášení

Prohlašuji, že jsem závěrečnou práci vypracoval samostatně a uvedl veškeré použité informační zdroje.

Souhlasím, aby tato studijní práce byla použita k výukovým a prezentačním účelům na Střední průmyslové a umělecké škole v Opavě, Praskova 399/8.

V Opavě 1. 1. 2024

. . . . . .

Podpis autora

Abstrakt

Ruda je programovací jazyk zaměřený na rychlé a snadné prototypování aplikací, umožňující vývojářům rychle testovat nápady bez zbytečné složitosti. Kompiluje se do systémově nezávislého binárního formátu pro rychlé spuštění pomocí Ruda VM. Jednoduchá syntaxe a vestavěné nástroje usnadňují vytváření prototypů. Důraz je kladen na fázi prototypování, s možností přechodu k jinému jazyku či platformě pro finální implementaci. Ruda poskytuje efektivní prostředky pro zkušební vývoj a validaci nápadů před případným přesunem do robustnějšího softwarového prostředí.

Klíčová slova

Jazyk, kompilace, binární formát

Obsah

Úvod

Ruda je programovací jazyk, který si dává za úkol zjednodušit prvotní fázi vývoje, což je v současnosti mnohdy rezervováno pouze pro omezený počet jazyků. Tyto jazyky ale často vyžadují nástroje třetích stran pro složitější úkoly. Toto se stává problematické zejména pro méně zkušené vývojáře, kteří tak mohou ztratit spustu času výzkumem, nutným k použití knihovny, kterou už nikdy nebudou potřebovat.

Také stojí za zmínku, že vše, co Ruda obsahuje bylo naprogramované mnou, s výjimkou knihovny SFML pro použití oken, knihovny git2 pro práci s verzovacím systémem git a několika menších knihoven pro utility jako například Toml serializace, hashování a parsování terminálových parametrů.

K tomu, abych vyřešil tyto problémy jsem pro Rudu zadal několik cílů:

- Vestavěný správce projektu - jeden správce by měl vystačit pro celý projekt.
- Garbage collector - oproti ostatním metodám úklidu paměti nevyžaduje téměř žádnou pozornost vývojáře.
- Grafické programování - modul pro kreslení na okno zabudovaný přímo do standardní knihovny.

V následujících kapitolách podrobně rozeberu proces kompilace zdrojových souborů, včetně jeho parsování. Nastíním strukturu Ruda bytecode a popíšu interpretaci v rámci virtuálního prostředí. Další témata zahrnují vestavěné nástroje, řešení chyb a praktické příklady použití.

1 Programovací jazyk Ruda

1.1 Pozadí vzniku

1.2 Porovnání alternativ

2 Překladač

Zdaleka nejrozsáhlejší část celého projektu je právě překladač. Každý jazyk musí projít nějakou formou překladu zdrojového kódu do formy, která je lépe pochopitelná pro počítač. Nejčastější metody jsou kompilace a interpretace. Ruda používá oba tyto způsoby tak, že zdrojový kód na počítači vývojáře projde kompilací a výsledný soubor obsahuje bytecode, který si potom může spustit klient pomocí interpretu Ruda VM.

Proces překladu se nazívá „compiler pipeline," při čemž projde kód několika navazujícími transformacemi, kde každá přibližuje kód více k finálnímu produktu.

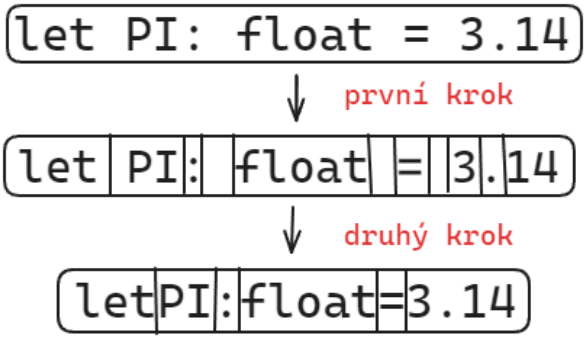
2.1 Analýza

Při analýze má překladač za úkol pochopit strukturu a fungování zdrojového kódu. A připravuje informace nutné, pro vytvoření výsledného bytecode.

2.1.1 Lexikální

Lexer, jinak řečeno tokenizer, tvoří první část analýzy zdrojového souboru. Hledá nejmenší významové prvky kódu, tzv. tokeny. Ty můžou být například slova, speciální znaky či „bílé znaky".

I přestože zpracování tokenů není starost lexeru, moje implementace využívá pro tokenizaci 2 kroky. První krok vytvoří „atomické prvky." Neprovádí přítom žádnou logiku a pouze zapisuje, co vidí. Druhý krok poté očistí bílé znaky a vytvoří složené tokeny, jako například řetězce a čísla.



2.1.2 Syntaktická

Při syntaktické analýze dochází k parsování vstupních tokenů, hovoříme tak o parseru. Ten postupně prochází tokeny a vytváří z nich strukturu, určenou ke zpracování logickou částí.

Pro Rudu jsem vytvořil vlastní univerzální parser. Používá programovatelný syntax čtený ze souboru, chybová hlášení generuje automaticky, a k tomu nabízí funkce pro debugování pravidel a jednoduchou validaci. Jeho syntax je jednoduchý a přehledný, programátor by měl být schopný zorientovat se už po pár minutách. Klíčové slovo let by v něm mohlo vypadat následovně:

Kód 2.1: Ukázka klíčového slova let

```
1 KWLet ident type expr
2         "let" harderr="true"
3         "'text" set="ident"
4         : ?
5                 type ? set="type"
6         = ?
7                 expr ? set="expr"
8         ; ?;
```

První řádek je hlavička slova začínající názvem „KWLet“ a pokračuje jeho parametry. Druhý porovnává, jestli je typ momentálního tokenu text s výrazem „let.“ Pokud ne, vrací error. Pokud ano, pokračuje dál a zapne harderr, ten říká, že pokud odteď dojde k chybě, tak není chybové pouze toto slovo, ale i jeho rodič, který dané slovo porovnával. Dále na řádku 3 porovnává jakýkoliv token reprezentující text. Pokud dojde ke schodě, nastaví parametr ident na porovnávaný token. Pokud nikoliv, vyhodí těžkou chybu (protože je zapnutý harderr). Otazník na řádku 4 znamená, že token není nutný. Pokud bude přítomen, vykoná se kód s vyšším odsazením (řádek č. 5). Středník na konci řádku 8 pouze oznamuje konec definice slova. Vytvořené slovo se později použije stejně jako na řádcích 5 a 7.

Za zmínku stojí také metoda připojení zdrojových souborů. Kompilátor nemůže jen tak zpracovat všechny soubory s příponou .rd, co vidí. To by bylo náročné na zdroje počítače. Tento problém řeší tak, že nejprve získá vstupní soubor (mometálně pouze main.rd). Ten zpracuje a zapíše si použití klíčového slova import, které má cestu k použitému zdroji. Zatím si je pouze zapíše, protože kdyby je ihned zpracoval, tak může dojít k zamrznutí, při použití cyklických importů. Poté je zpracuje a označí jako zpracované.

2.1.3 Sémantická

Poslední fáze analýzy má za úkol dát smysl struktuře získané po parsování. Protože parser je dynamický, tak musí nejprve vše vytáhnout do ucelené podoby a provést případné validace. Kontroluje například zda se neopakují názvy, ale také vytváří typy, parsuje matematické výrazy, apod.

2.2 GENERACE BYTECODE

Generace je nejdůležitější část každého překladače. Jde o proces, kdy zpracuje všechna získaná data a vytvoří užitečný produkt v podobě programu, či knihovny.

Má implementace obsahuje generaci nejen bytecode, ale i informací pro debugování. To je užitečné v případě, že běžící Ruda program narazí na problém. Ruda VM díky toho zvládne podat důležité informace o původu problému.

2.2.1 Generace

Překladač před generací prošel třemi fázemi analýzi, aby měl znalost celého kódu. Samotná generace je ale stále velmi náročný proces. V kódu je obrovské množství způsobů, jak přistupovat k datům a jejich manipulaci. Transformace jazyk -> bytecode se potom stává velmi složitá, zmatečná a pokud provedení není perfektní, vytváří prostor pro nečekané chyby.

Překladač začíná tím, že všem funkcím rozdá unikátní ID, které bude velmi důležité později. Vytvoří potřebné objekty společně s Ruda VM (pouze pro kompatibilitu; nikdy ho nespustí). Až bude vše připravené, začne s generací funkcí. Nezáleží na jejich pořadí, nakonec je zpracuje všechny.

Pro překlad funkce používá rekurzivní prohledávání uzlů, které reprezentují výrazy. Pro každý vygeneruje patřičný bytecode. Například pro klíčové slovo let:

Kód 2.2: Překlad klíčového slova let

```
1 // vyraz
2 let a = new 60
3
4 // bytecode
5 ReadConst(const_int_60, GENERAL_REG1)
6 AllocateStatic(1)
7 WritePtr(GENERAL_REG1)
8 Write(a_stack_pos, POINTER_REG)
```

Výraz vytvoří proměnnou a, do které zapíše ukazatel na hodnotu 60, která je uložená na haldě. Bytecode nejprve přečte hodnotu 60 z paměti a uloží ji do registru GENERAL-REG0. Potom na haldě vytvoří místo pro jednu hodnotu a ukazatel se automaticky uloží do registru POINTER-REG, který potom využije instrukce WritePtr, k tomu, aby na něj zapsala hodnotu z GENERAL-REG1, která je stále číslo 60. Už jenom potřebuje zapsat ukazatel, který stále přebývá v POINTER-REG do pozice na stacku vyhrazené pro proměnnou a.

Tím tenhle příklad končí, ale je nutné vědět, že takhle bude bytecode vypadat až po optimalizaci. Generátor kódu je sám o sobě hloupí a nestará se o žádné stavy. Vždy zohledňuje pouze nejhorší možný případ a vytváří přitom poměrně nekvalitní a hlavně pomalý kód.

2.2.2 Optimalizace

3 VIRTUÁLNÍ STROJ

V minulé kapitole jsme prošli, jak ze zdrojového kódu vznikne program. Zde se zaměříme na jeho spuštění pomocí Ruda VM, který je zodpovědný za provedení Ruda programů.

3.1 VLASTNOSTI

3.1.1 Instrukce

Instrukce jsou jedna ze částí Ruda bytecode. Jde o jednoduché pokyny, popisující co má provést Ruda VM. Jednotlivé instrukce jsou navrhnuté, pro co nejširší možné využití v rámci uzavřeného runtimu.

Právě proto lze Rudu použít tam, kde je bezpečnost priorita, pokud se povolí pouze moduly standardní knihovny pracující nezávisle na systému. Tato vlastnost by v budoucnu mohla Ruda běžet například na serverech, nebo jako skriptovací jazyk v jiném programu.

Pro referenci jednotlivých instrukcí doporučuji prohlédnout přímo zdrojový kód runtimu, který lze nalézt na https://github.com/it-2001/Ruda/blob/main/vm/runtime/src/lib.rs. Zde můžete vyhledat "enum Instructions"(okolo řádku 2500), kde jsou všechny instrukce popsané.

3.1.2 Běh programu

Samotné jádro Rudy je pouze knihovna, nelze tudíž přímo spustit. K tomu pomáhá program Ruda VM, který zpracuje bytecode a připraví podle něj příslušné prostředí pro běh. Kromě toho podává zprávy o stavu, parsuje vstupní argumenty a připravuje potřebné knihovny.

Poté, co Ruda VM připraví vše nutné, tak teprve spustí Ruda program. Instrukce běží sériově, dokud nenarazí na problém, nebo na konec programu označený instrukcí End.

3.1.3 Model paměti

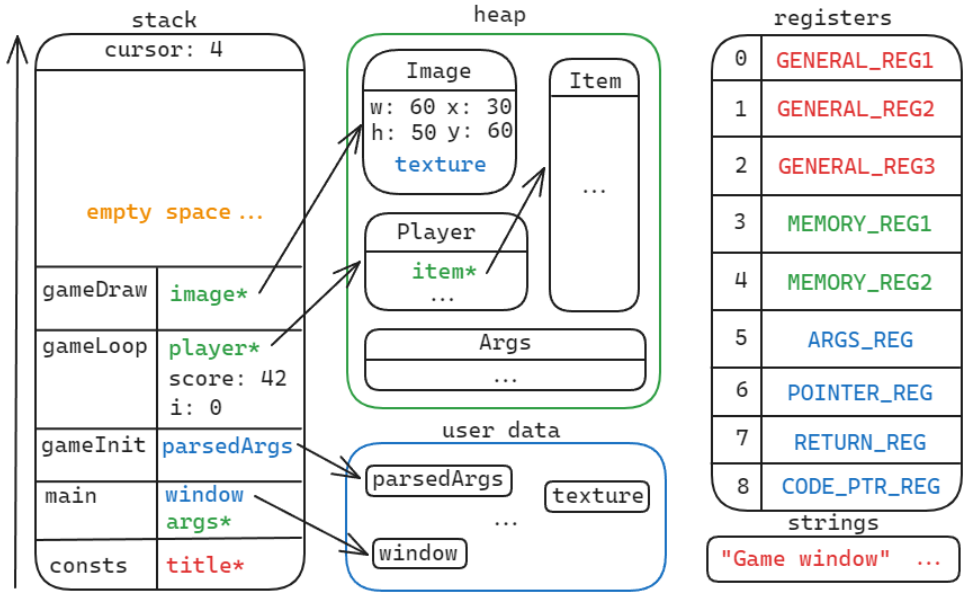
Pamět funguje na základě velmi využívané kombinace stacku, haldy a registrů. Pro většinu dat je tento způsob dostačující. Nedostatky se začnou ukazovat pokud chceme použít víc, než jen vlastní struktury. K pochopení tohoto problému je ale nutné vědět, jak Ruda ukládá data.

Ruda je zároveň silně typovaný a dynamický jazyk. To znamená, že se typy kontrolují při kompilaci a zároveň za běhu programu. K tomu, aby typy mohly být otestovány dynamicky je nutné, aby každá hodnota obsahovala hlavičku udávající její typ. V případě ukazatelů nestačí pouze standardních 4 až 8 bytů z důvodů, které popíšu později a protože všechny typy musí mít schodnou velikost, tak každá hodnota zabírá přesně 24 bytů paměti.

To by samo o sobě bylo v pořádku. Problém ale nastává při ukládání řetězců. Jeden řetězec s obsahem "Hello, world!"by měl zabrat 13 bytů. V Rudě vyroste ten stejný řetězec na celých 312 bytů. Tento nárust může být problematický zejména při zpracovávání velkých textů, jako jsou soubory, které mohou mnohonásobně nabýt na náročnosti pro paměť. Řešení bylo vytvořit novou haldu určenou pro řetězce. To umožňuje používat String typ v Rustu, který není jen rychlejší, ale přináší i užitečné vlastnosti jako kompatibilita s UTF-8 a celou škálu transformačních metod.

Zbývalo ještě vyřešit, jak povolit nativním knihovnám ukládat vlastní data do Rudy. Tyto knihovny by mohly vytvořit objekt na haldě a vrátit ukazatel. Taková operace je ale náročná, protože vývojář knihovny musí zjistit, jak bude Ruda k takovému objektu přistupovat. Proto vznikla struktura podobná haldě, určená pro ukládání tzv. „user data“ - data neznámé velikosti, které splňují UserData interface. To dává Rudě schopnost vlastnit data, která by nešla uložit tradičními způsoby, jako například otevřené okno.

Pro okamžitý přístup k jednotlivým hodnotám má Ruda celkem 9 registrů.



3.2 GARBAGE COLLECTOR

Všechny výše zmíněné struktury se musí čistit od nepoužívaných dat. Ruda proto nabízí vlastní garbage collector, fungující na principu dvou kroků.

3.2.1 Označení

První krok se nazívá označení. Jeho funkce je projít známou paměť a vyhledat všechna data, ke kterým se dostane pomocí existujících ukazatelů. Za známá data se považuje stack, registry a speciální struktura pro uložení argumentů funkcí. Pokud při označování nalezne ukazatel na haldu, řetězec, nebo user data, tak si zaznačí, že data můžou být stále použity. V případě, že zaznačí data na haldě, hledá nové ukazatele na dané pozici.

3.2.2 Úklid

Při úklidu najde všechna data, ke kterým není přístup na základě informací z první fáze a příslušné data uvolní pro další použití. Prostor zůstává přiřazený Ruda programu i po uvolnění, protože je velmi pravděpodobné, že pokud byl prostor použit jednou, tak bude znovu. Doporučuji proto periodicky spouštět úklid, aby bylo maximální využití paměti. Ruda nabízí modul pro manipulaci s garbage collectorem, takže si uživatel může vybrat vhodný čas.

3.3 ROZŠÍŘENÍ POMOCÍ NATIVNÍCH KNIHOVEN

Možnost pro nativní knihovny byla nevyhnutelná. Ruda nedovoluje žádný input ani output, takže program napsaný v čisté Rudě by byl zbytečný. Standardní knihovna proto používá veřejný interface pro Ruda nativní knihovny.

Tento interface může použít každý pro vlastní účely a rozšířit tak vlastnosti Rudy.

Zatím pouze pro jazyk Rust.

3.4 SCHOPNOST SKRIPTOVÁNÍ

Jedna z výhod toho, že Ruda runtime je pouze knihovna je, že lze použít jako skriptovací jazyk v jiném programu. Momentálně pro podobné využití nemá dostatečnou podporu, ale v budoucnu to bude standardní součást jazyka.

4 Nástroje pro práci s jazykem

4.1 Dokumentace

4.2 Package manager

4.3 Standardní knihovna

4.4 Grafický inspektor Ruda projektu

4.5 Syntax highlighting

5 Instalace

5.1 Podporované platformy

5.2 Stažení

5.2.1 Kompilace ze zdrojových kódů

5.2.2 Stažení předkompilované verze

5.3 Proměnné prostředí

5.4 Možné problémy

6 Programování v Rudě

6.1 Syntaxe

6.2 Příklady

7 Zhodnocení

7.1 Splněné cíle

7.2 Poznatky

7.3 Možnosti dalšího vývoje

LITERATURA

[1] DOKULIL Jakub. *Šablona pro psaní SOČ v programu LaTeX* [Online]. Brno, 2020 [cit. 2020-08-24]. Dostupné z: https://github.com/Kubiczek36/SOC_sablona

[2] OETIKER, Tobias, Hubert PARTL, Irene HYNA, Elisabeth SCHEGL, Michal KOČER a Pavel SÝKORA. *Ne příliš stručný úvod do systému LaTeX2e* [online]. 1998 [cit. 2020-08-24]. Dostupné z: https://www.jaroska.cz/elearning/informatika/typografie/lshort2e-cz.pdf

[3] *Wikibooks: LaTeX* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2020-08-24]. Dostupné z: https://en.wikibooks.org/wiki/LaTeX

[4] *TeX - LaTeX Stack Exchange* [online]. Stack Exchange, 2020 [cit. 2020-09-01]. Dostupné z: https://tex.stackexchange.com

[5] *Střední škola průmyslová a umělecká Opava* [online]. [cit. 2023-11-11]. Dostupné z: https://www.sspu-opava.cz

[6] *Citace PRO* [online]. Citace.com, 2020 [cit. 2020-08-31]. Dostupné z: https://www.citacepro.com

[7] BORN, Max a Emil WOLF. *Principles of optics: electromagnetic theory of propagation, interference and diffraction of light*. 7th (expanded) edition. Reprinted wirth corrections 2002. 15th printing 2019. Cambridge: Cambridge University Press, 2019. ISBN 978-0-521-64222-4.

Seznam obrázků

Seznam tabulek