

# 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

<b>Úvod</b>	<b>3</b>
<b>1 Programovací jazyk Ruda</b>	<b>5</b>
1.1 Pozadí vzniku . . . . .	5
1.2 Porovnání alternativ . . . . .	5
<b>2 Překladač</b>	<b>7</b>
2.1 Analýza . . . . .	7
2.2 Generace bytecode . . . . .	9
<b>3 Virtuální stroj</b>	<b>11</b>
3.1 Popis . . . . .	11
3.2 Garbage collector . . . . .	11
3.3 Rozšíření pomocí dynamických knihoven . . . . .	11
3.4 Nevýhody využití Rustu . . . . .	11
3.5 Schopnost skriptování . . . . .	11
<b>4 Nástroje pro práci s jazykem</b>	<b>13</b>
4.1 Dokumentace . . . . .	13
4.2 Package manager . . . . .	13
4.3 Standardní knihovna . . . . .	13
4.4 Grafický inspektor Ruda projektu . . . . .	13
4.5 Syntax highlighting . . . . .	13
<b>5 Instalace</b>	<b>15</b>
5.1 Podporované platformy . . . . .	15
5.2 Stažení . . . . .	15
5.3 Proměnné prostředí . . . . .	15
5.4 Možné problémy . . . . .	15
<b>6 Programování v Rudě</b>	<b>17</b>
6.1 Syntaxe . . . . .	17
6.2 Příklady . . . . .	17
<b>7 Zhodnocení</b>	<b>19</b>
7.1 Splněné cíle . . . . .	19
7.2 Poznátky . . . . .	19
7.3 Možnosti dalšího vývoje . . . . .	19



# 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 nazývá „compiler pipeline,“ při čemž projde kód několika navazujícími transformacemi, kde každá přibližuje kód více k finálnímu produktu.

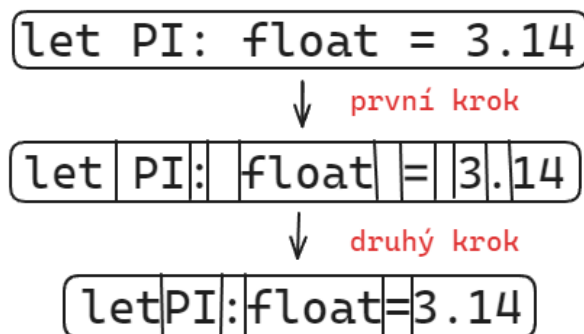
### 2.1 ANALÝZA

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 mohou 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 přitom 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 odted' dojde k chybě, tak není chybové pouze toto slovo, ale i jeho rodič, který dané slovo porovnával. Dále na řádce 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 řádce 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 (momentálně 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 analýzy, 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 poměrně hloupý a nestará se ani o budoucnost, ani o minulost. Vždy zohledňuje pouze nejhorší možný případ a vytváří přitom poměrně nekvalitní a hlavně pomalý kód.

### **2.2.2 Optimalizace**

## **3 VIRTUÁLNÍ STROJ**

### **3.1 POPIS**

#### **3.1.1 Instrukcí**

#### **3.1.2 Běh programu**

#### **3.1.3 Model paměti**

### **3.2 GARBAGE COLLECTOR**

### **3.3 ROZŠÍŘENÍ POMOCÍ DYNAMICKÝCH KNIHOVEN**

### **3.4 NEVÝHODY VYUŽITÍ RUSTU**

### **3.5 SCHOPNOST SKRIPTOVÁNÍ**



## **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 L<sup>A</sup>T<sub>E</sub>X* [Online]. Brno, 2020 [cit. 2020-08-24]. Dostupné z: [https://github.com/Kubiczek36/SOC\\_sablona](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 L<sup>A</sup>T<sub>E</sub>X2<sub>ε</sub>* [online]. 1998 [cit. 2020-08-24]. Dostupné z: <https://www.jaroska.cz/elearning/informatika/typografie/lshort2e-cz.pdf>
- [3] *Wikibooks: L<sup>A</sup>T<sub>E</sub>X* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2020-08-24]. Dostupné z: <https://en.wikibooks.org/wiki/LaTeX>
- [4] *TeX - L<sup>A</sup>T<sub>E</sub>X 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 with corrections 2002. 15th printing 2019. Cambridge: Cambridge University Press, 2019. ISBN 978-0-521-64222-4.

## Seznam obrázků



# Seznam tabulek