Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Návrh a implementace zásobníkového jazyka s JIT kompilací v jazyce Rust

2026

Albert Klinkovský

Vedoucí práce:
Mgr. Jan Laštovička, Ph.D.

Studijní program: Informatika,
Specializace: Programování a vývoj software

Bibliografické údaje

Autor:	Albert Klinkovský
Název práce:	Návrh a implementace zásobníkového jazyka s JIT kompilací v jazyce Rust
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2026
Studijní program:	Informatika, Specializace: Programování a vývoj software
Vedoucí práce:	Mgr. Jan Laštovička, Ph.D.
Počet stran:	43
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Albert Klinkovský
Title:	Design and implementation of a stack language with JIT compilation in Rust
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2026
Study program:	Computer Science, Specialization: Programming and Software Development
Supervisor:	Mgr. Jan Laštovička, Ph.D.
Page count:	43
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Práce se zabývá návrhem a implementací kompilátoru zásobníkového programovacího jazyka inspirovaného jazykem Forth. Kompilátor je implementován v jazyce Rust a podporuje více cílových platforem (Rust, C). Součástí práce je interaktivní prostředí REPL využívající JIT kompilaci, optimalizační průchody nad mezikódem a standardní knihovna jazyka.

Synopsis

This thesis deals with the design and implementation of a compiler for a stack-based programming language inspired by Forth. The compiler is implemented in Rust and supports multiple target platforms (Rust, C). The work includes an interactive REPL environment using JIT compilation, optimization passes over intermediate representation, and a standard library for the language.

Klíčová slova: kompilátor; Forth; Rust; zásobníkový jazyk; JIT kompilace; optimalizace

Keywords: compiler; Forth; Rust; stack-based language; JIT compilation; optimization

Děkuji vedoucímu práce za odborné vedení a cenné rady.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

Seznam tabulek

Seznam zdrojových kódů

1 Úvod

Zásobníkové programovací jazyky představují specifickou kategorii jazyků, které se od běžných jazyků odlišují fundamentálním přístupem k předávání dat mezi operacemi. Místo pojmenovaných proměnných a explicitního předávání argumentů využívají implicitní datový zásobník, na který operace ukládají své výsledky a ze kterého odebírají své operandy. Tento přístup vede k minimalistické syntaxi a vysoké kompozičnosti kódu.

Nejvýznamnějším představitelem zásobníkových jazyků je Forth, navržený Charlesem Moorem v šedesátých letech 20. století. Forth si získal oblibu zejména v oblasti vestavěných systémů a řízení hardware, kde jeho jednoduchost a efektivita představují významné výhody. Přestože dnes Forth nepatří mezi nejrozšířenější programovací jazyky, jeho koncepty ovlivnily návrh mnoha moderních technologií, včetně virtuálních strojů, jako jsou JVM a WebAssembly.

Tato práce se zabývá návrhem a implementací kompilátoru pro zásobníkový jazyk nazvaný Roth, který je inspirován jazykem Forth, ale přináší některé moderní prvky. Kompilátor je implementován v jazyce Rust, což umožňuje využít jeho silný typový systém a záruky bezpečnosti paměti bez nutnosti použití garbage collectoru.

1.1 Motivace

Motivace pro tuto práci vychází z několika oblastí:

1.1.0.1 Vzdělávací hodnota zásobníkových jazyků. Zásobníkové jazyky poskytují jedinečný pohled na principy výpočtu a kompilace. Jejich jednoduchost umožňuje soustředit se na podstatu problému bez syntaktického balastu běžných jazyků. Studium a implementace takového jazyka nabízí hluboký vhled do fungování kompilátorů, optimalizací a generování kódu.

1.1.0.2 Praktické využití kompilátoru. Kompilátor Roth není pouze akademickým cvičením. Generuje efektivní kód pro reálné platformy (Rust, C) a obsahuje interaktivní prostředí REPL s JIT kompilací, které umožňuje okamžitou zpětnou vazbu při vývoji. Tato kombinace činí z Roth nástroj použitelný jak pro výuku, tak pro experimentování s nízkoúrovňovým programováním.

1.1.0.3 Volba jazyka Rust pro implementaci. Rust představuje moderní systémový programovací jazyk, který kombinuje výkon jazyků jako C nebo C++ s bezpečností paměti garantovanou již v době kompilace. Pro implementaci kompilátoru nabízí Rust několik výhod: algebraické datové typy ideální pro reprezentaci AST a mezikódu, pattern matching pro elegantní zpracování syntaktických konstrukcí, a ownership systém zabraňující běžným chybám při správě paměti.

1.2 Cíle práce

Hlavním cílem práce je navrhnout a implementovat plnohodnotný kompilátor zásobníkového jazyka. Tento cíl lze rozdělit do následujících dílčích cílů:

1. **Návrh jazyka Roth** — Definovat syntaxi a sémantiku zásobníkového jazyka inspirovaného jazykem Forth, který bude dostatečně expresivní pro praktické použití.

2. **Implementace kompilátoru** — Vytvořit kompilátor v jazyce Rust, který provádí lexikální, syntaktickou a sémantickou analýzu zdrojového kódu a generuje mezikód (IR).

3. **Optimalizace mezikódu** — Implementovat sadu optimalizačních průchodů včetně constant foldingu, eliminace mrtvého kódu, peephole optimalizací a inliningu.

4. **Generování cílového kódu** — Vytvořit backendy pro generování kódu v jazycích Rust a C, což umožní přenositelnost na různé platformy.

5. **REPL s JIT kompilací** — Implementovat interaktivní prostředí, které využívá just-in-time (JIT) kompilaci pro okamžité vyhodnocování příkazů.

6. **Standardní knihovna** — Vytvořit základní knihovnu obsahující běžně používané operace pro práci se zásobníkem, aritmetiku, vstup a výstup.

1.3 Struktura dokumentu

Dokument je strukturován následovně:

Kapitola 2 poskytuje teoretický základ nutný pro pochopení práce. Představuje jazyk Forth, jeho historii a principy zásobníkové architektury. Dále se věnuje obecné teorii kompilátorů, fázím překladu a optimalizacím. Závěr kapitoly popisuje relevantní vlastnosti jazyka Rust.

Kapitola 3 se zabývá analýzou požadavků na kompilátor a návrhem jeho architektury. Definuje instrukční sadu mezikódu a navrhuje optimalizační průchody.

Kapitola 4 detailně popisuje implementaci jednotlivých částí kompilátoru: lexer, parser, sémantický analyzátor, generátor mezikódu, optimalizátor, backendy pro generování kódu, REPL prostředí a standardní knihovnu.

Kapitola 5 prezentuje metodiku testování kompilátoru a výsledky testů. Obsahuje také porovnání s existujícími implementacemi jazyka Forth.

Závěr shrnuje dosažené výsledky, hodnotí splnění cílů a navrhuje možná rozšíření.

Přílohy obsahují elektronickou verzi zdrojových kódů, uživatelskou dokumentaci a formální gramatiku jazyka.

2 Teoretický základ

2.1 Jazyk Forth

Jazyk Forth představuje jedinečnou kategorii programovacích jazyků, která se fundamentálně odlišuje od většiny běžně používaných jazyků. Jeho minimalistický design, zásobníková architektura a interaktivní povaha z něj činí ideální základ pro studium principů kompilace a návrhu programovacích jazyků.

2.1.1 Historie a filozofie

Jazyk Forth byl vytvořen Charlesem H. Moorem na přelomu šedesátých a sedmdesátých let 20. století. Moore původně vyvíjel tento jazyk pro řízení radioteleskopů na observatoři Kitt Peak v Arizoně, kde potřeboval jazyk, který by umožnil interaktivní vývoj a ladění v prostředí s omezenými výpočetními prostředky [**moore1974**].

Název Forth vznikl jako zkratka pro *fourth generation language* (jazyk čtvrté generace), přičemž kvůli omezením souborového systému IBM 1130, který povoloval pouze pětiznaková jména souborů, byl název zkrácen na FORTH. Moore považoval svůj jazyk za výrazný krok kupředu oproti tehdejším jazykům třetí generace, jako byly FORTRAN a COBOL.

Filozofie jazyka Forth je založena na několika klíčových principech, které Leo Brodie shrnul ve své vlivné knize *Thinking Forth* [**brodie2004**]:

2.1.1.1 Jednoduchost a minimalismus. Forth upřednostňuje jednoduchou implementaci před složitou syntaxí. Základní jádro jazyka může být implementováno v řádově stovkách řádků kódu, což umožňuje jeho nasazení i na velmi omezených platformách. Tato jednoduchost není náhodná — Moore věřil, že složitost je nepřítelem spolehlivosti a údržby.

2.1.1.2 Kompozice malých částí. Programy ve Forthu jsou budovány z malých, opakovaně použitelných jednotek nazývaných *slova* (words). Každé slovo by mělo řešit jeden konkrétní problém a jeho délka by neměla přesáhnout zhruba jeden řádek. Tento přístup předznamenal moderní principy jako *Unix philosophy* a mikroslužby.

2.1.1.3 Interaktivita a okamžitá zpětná vazba. Forth byl od počátku navržen jako interaktivní jazyk. Programátor může okamžitě testovat jednotlivá slova a pozorovat jejich efekt na zásobník. Tato vlastnost činí Forth ideálním nástrojem pro explorativní programování a ladění v reálném čase.

2.1.1.4 Programátor jako architekt. Na rozdíl od jazyků, které definují pevnou strukturu programů, Forth poskytuje programátorovi nástroje pro vytvoření vlastního jazyka specifického pro danou doménu. Každý Forth program je vlastně rozšířením jazyka samotného.

Historicky našel FORTH široké uplatnění v oblastech, kde jsou kritické efektivita a přímý přístup k hardware:

- **Embedded systémy** — díky malým nárokům na paměť a možnosti přímé manipulace s registry.
- **Kosmický průmysl** — NASA použila FORTH v řadě vesmírných misí včetně sondy Philae.
- **Inicializace systémů** — standard IEEE 1275 (Open Firmware) používá FORTH pro bootovací firmware [**open-firmware**].
- **Radioastronomie** — původní doména jazyka, kde se používá dodnes.

2.1.2 Zásobníková architektura

Fundamentálním rysem jazyka FORTH je jeho zásobníková architektura. Na rozdíl od většiny programovacích jazyků, které používají pojmenované proměnné pro předávání dat mezi operacemi, FORTH využívá implicitní datový zásobník [**koopman1989**].

2.1.2.1 Datový zásobník. Datový zásobník (data stack, také parameter stack) je primární strukturou pro předávání hodnot mezi slovy. Operace odebírají své operandy z vrcholu zásobníku a ukládají výsledky zpět na zásobník. Tento přístup eliminuje potřebu explicitního pojmenování mezivýsledků.

Uvažujme výpočet aritmetického výrazu `(3 + 4) * 5`. V jazyce FORTH by tento výpočet vypadal následovně:

```
3 4 + 5 *
```

Průběh výpočtu demonstruje tabulka 1.

Tabulka 1: Průběh výpočtu výrazu na zásobníku

Operace	Zásobník	Popis
`3`	`3`	Uložení konstanty 3
`4`	`3 4`	Uložení konstanty 4
`+`	`7`	Součet dvou vrchních hodnot
`5`	`7 5`	Uložení konstanty 5
`*`	`35`	Součin dvou vrchních hodnot

2.1.2.2 Návratový zásobník. Kromě datového zásobníku používá FORTH také návratový zásobník (return stack), který primárně slouží pro uložení návratových adres při volání slov. Programátor má však přístup i k tomuto zásobníku prostřednictvím slov `>R` (přesun z datového na návratový zásobník), `R>` (opačný směr) a `R@` (kopie vrcholu návratového zásobníku). Tato vlastnost umožňuje dočasné „odložení“ hodnot z datového zásobníku.

2.1.2.3 Reverzní polská notace. Zásobníková architektura přirozeně vede k použití postfixové neboli reverzní polské notace (Reverse Polish Notation, RPN). V této notaci se operátor zapisuje za své operandy, nikoliv mezi ně. RPN má několik výhod:

- Nevyžaduje závorky pro určení priority operací.
- Odpovídá přirozenému pořadí vyhodnocování na zásobníku.
- Zjednodušuje implementaci parseru a interpretru.
- Vede ke kompaktnějšímu zápisu složitých výrazů.

2.1.2.4 Stack effects. Konvence ve FORTHu vyžaduje dokumentovat efekt každého slova na zásobník pomocí notace zvané *stack effect comment*. Tato notace má tvar `( before - after )`, kde `before` popisuje hodnoty očekávané na vstupu a `after` hodnoty zanechané na výstupu. Například:

- `+ ( n1 n2 - n3 )` — sečte dvě čísla
- `DUP ( n - n n )` — zduplikuje vrchol zásobníku
- `SWAP ( n1 n2 - n2 n1 )` — prohodí dva vrchní prvky
- `DROP ( n - )` — odstraní vrchol zásobníku

Tyto anotace jsou klíčové pro správné použití slov, neboť při nesprávném počtu hodnot na zásobníku dochází k chybám za běhu.

2.1.3 Syntaxe a sémantika

Syntaxe jazyka FORTH je extrémně minimalistická. Zdrojový kód se skládá ze *slov* oddělených bílými znaky. Neexistují žádné speciální oddělovače příkazů, závorky pro volání funkcí ani klíčová slova v tradičním smyslu [**brodie1981**].

2.1.3.1 Slova. Základní jednotkou programu ve FORTHu je *slovo* (word). Slovo je sekvence znaků oddělená bílými znaky, která reprezentuje buď vestavěnou operaci (primitive), uživatelem definovanou funkci, nebo číselnou konstantu.

```
( Definice noveho slova KVADRAT )
: KVADRAT ( n -- n^2 )  DUP * ;

( Pouziti )
5 KVADRAT .   ( Vypise: 25 )
```

Definice nového slova začíná dvojtečkou (`:`), následuje název slova, tělo definice a ukončení středníkem (`;`). Komentáře se uvádějí v kulatých závorkách.

2.1.3.2 Čísla. Číselné konstanty jsou rozpoznány lexikálním analyzátorem a při zpracování jsou automaticky uloženy na zásobník. FORTH tradičně podporuje celočíselnou aritmetiku, přičemž moderní implementace často přidávají podporu pro čísla s plovoucí řádovou čárkou.

2.1.3.3 Řídicí struktury. FORTH poskytuje základní řídicí struktury pro větvení a cykly:

```
( Podminka IF-ELSE-THEN )
: SIGNUM ( n -- -1|0|1 )
  DUP 0 < IF DROP -1 ELSE
  DUP 0 > IF DROP 1 ELSE DROP 0 THEN THEN ;

( Pocitany cyklus DO-LOOP )
: VYPIS-5 ( -- )  5 0 DO I . LOOP ;

( Cyklus s podminkou BEGIN-WHILE-REPEAT )
: COUNTDOWN ( n -- )
  BEGIN DUP 0 > WHILE DUP . 1 - REPEAT DROP ;
```

Důležité je si povšimnout, že `IF` ve FORTHu je postfixové — podmínka je vyhodnocena před klíčovým slovem `IF`, které pak odebere hodnotu ze zásobníku a rozhodne o dalším průběhu.

2.1.3.4 Proměnné a paměť. Ačkoliv FORTH primárně pracuje se zásobníkem, poskytuje také mechanismy pro práci s pojmenovanou pamětí:

```
VARIABLE POCITADLO      ( Deklarace promenne )
0 POCITADLO !           ( Inicializace na 0 )
POCITADLO @             ( Cteni hodnoty )
POCITADLO @ 1+ POCITADLO !  ( Inkrementace )
```

Operátor `!` (store) ukládá hodnotu na adresu, operátor `@` (fetch) čte hodnotu z adresy. Název proměnné vrací její adresu, nikoliv hodnotu.

2.1.4 Existující implementace

Za více než padesát let existence jazyka FORTH vzniklo mnoho jeho implementací, od minimalistických embedded verzí po plnohodnotné vývojové systémy.

2.1.4.1 Gforth. Gforth (GNU Forth) je referenční implementací standardu Forth 2012 [**gforth**]. Je vyvíjen pod záštitou projektu GNU a představuje nejpoužívanější open-source implementaci FORTHu. Gforth je napsán v jazycích C a FORTH a je dostupný pro většinu operačních systémů. Mezi jeho hlavní přednosti patří:

- Plná kompatibilita se standardem Forth 2012.

- Rozsáhlá dokumentace a aktivní komunita.
- Efektivní interpretace pomocí techniky *threaded code* [**ertl2002**].
- Podpora pro čísla s plovoucí řádovou čárkou a dynamickou alokaci paměti.

2.1.4.2 Historické a specializované implementace. Mezi další významné implementace patří:

- **fig-Forth** — historická implementace od Forth Interest Group, která pomohla rozšířit jazyk na osmibitové počítače.
- **SwiftForth** — komerční implementace od společnosti Forth, Inc., optimalizovaná pro výkon.
- **Mecrisp** — moderní implementace zaměřená na mikrokontroléry ARM Cortex-M.
- **Factor** — moderní zásobníkový jazyk inspirovaný FORTHem, rozšířený o objektově orientované programování, garbage collection a bohatou standardní knihovnu.

2.1.4.3 Standard Forth 2012. Jazyk FORTH byl standardizován americkým institutem ANSI v roce 1994 (ANSI X3.215-1994) a později rozšířen komunitním standardem Forth 2012 [**forth-standard**]. Standard definuje základní slovníkové sady (word sets):

Core — základní slova nezbytná pro každou implementaci

Core Extensions — užitečná rozšíření základní sady

Block — práce s blokovým souborovým systémem

Double-Number — podpora pro dvojnásobnou přesnost

Exception — mechanismus pro zpracování výjimek

Facility — interakce s uživatelem a systémem

File — práce se soubory

Floating-Point — čísla s plovoucí řádovou čárkou

Locals — lokální proměnné

Memory-Allocation — dynamická alokace paměti

Programming-Tools — ladící nástroje

Search-Order — správa slovníků

String — práce s řetězci

2.1.4.4 Vztah jazyka Roth k existujícím implementacím. Jazyk Roth, jehož implementace je předmětem této práce, se inspiruje základními koncepty jazyka Forth, ale záměrně se nedrží striktně standardu. Cílem je vytvořit moderní zásobníkový jazyk, který zachovává filozofii a eleganci FORTHu, ale využívá moderní techniky kompilace a optimalizace. Kompilátor Roth generuje nativní kód prostřednictvím backendů pro jazyky Rust a C, což jej odlišuje od interpretovaných implementací jako je Gforth.

2.2 Kompilátory a interpretery

Překladač (kompilátor) je program, který transformuje zdrojový kód napsaný v jednom programovacím jazyce do jiného jazyka, typicky do strojového kódu nebo mezikódu [**aho2006**]. Tato sekce poskytuje teoretický základ pro pochopení architektury kompilátoru Roth.

2.2.1 Fáze kompilace

Proces kompilace lze rozdělit do několika oddělených fází, z nichž každá provádí specifickou transformaci nad vstupními daty. Moderní kompilátory typicky implementují následující fáze:

2.2.1.1 Lexikální analýza (lexer). První fáze kompilace převádí vstupní text (proud znaků) na proud tokenů. Token je základní lexikální jednotka jazyka, například klíčové slovo, identifikátor, číslo nebo operátor. Lexer ignoruje bílé znaky a komentáře (případně je předává jako speciální tokeny).

Pro jazyk Roth rozlišujeme následující typy tokenů:

- `Number` — celočíselná konstanta
- `Word` — identifikátor (slovo)
- `StartDefinition` — znak `:`
- `EndDefinition` — znak `;`
- `StringLiteral` — řetězcová konstanta
- `Comment` — komentář v závorkách

2.2.1.2 Syntaktická analýza (parser). Parser převádí proud tokenů na abstraktní syntaktický strom (AST). AST reprezentuje hierarchickou strukturu programu a zachycuje vztahy mezi jednotlivými konstrukcemi jazyka. Na rozdíl od konkrétního syntaktického stromu AST abstrahuje od nepodstatných detailů syntaxe.

Zásobníkové jazyky mají obecně velmi jednoduchou syntaxi, což zjednodušuje implementaci parseru. AST jazyka Roth obsahuje následující typy uzlů:

- `Program` — kořenový uzel obsahující seznam příkazů
- `Definition` — definice slova s názvem a tělem
- `Number` — číselná konstanta
- `Word` — volání slova
- `StringLiteral` — řetězcová konstanta
- `VariableDeclaration` — deklarace proměnné

2.2.1.3 Sémantická analýza. Sémantický analyzátor ověřuje, zda je program sémanticky správný. Kontroluje například, zda jsou všechna použitá slova definována, zda nedochází k redefinici vestavěných slov a zda jsou typy kompatibilní (v jazycích s typovým systémem).

V kompilátoru Roth sémantická analýza udržuje tabulku definovaných slov a proměnných. Při analýze každého slova ověřuje, zda bylo dříve definováno buď jako vestavěné slovo, uživatelská definice, nebo proměnná.

2.2.1.4 Generování mezikódu. Po sémantické analýze následuje transformace AST do mezikódu (intermediate representation, IR). Mezikód je nízkoúrovňová reprezentace programu, která je nezávislá na cílové platformě, ale dostatečně konkrétní pro efektivní optimalizaci a generování cílového kódu.

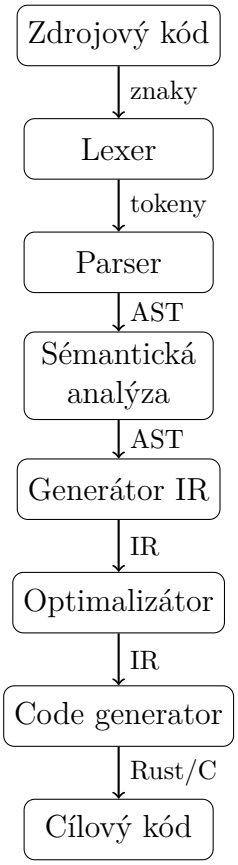
2.2.1.5 Optimalizace. Optimalizační fáze provádí transformace mezikódu s cílem zlepšit výkon nebo zmenšit velikost výsledného programu. Optimalizace mohou být lokální (v rámci jednoho bloku instrukcí) nebo globální (přes celý program).

2.2.1.6 Generování cílového kódu. Poslední fáze transformuje optimalizovaný mezikód do cílového jazyka. V případě tradičních kompilátorů je cílovým jazykem strojový kód nebo assembler. Kompilátor Roth generuje kód v jazycích Rust nebo C, které jsou následně kompilovány standardními kompilátory těchto jazyků.

Obrázek 1 schematicky znázorňuje průchod dat jednotlivými fázemi kompilátoru.

2.2.2 Mezikód

Mezikód (intermediate representation, IR) je klíčovou součástí moderních kompilátorů. Odděluje frontend (analýza zdrojového kódu) od backendu (generování cílového kódu), což umožňuje nezávislý vývoj obou částí a podporu více cílových platforem.



Obrázek 1: Fáze kompilátoru Roth

2.2.2.1 Typy mezikódu.

Existuje několik základních typů mezikódu [**aho2006**]:

Tříadresový kód — instrukce mají tvar `x = y op z`, kde každá instrukce má nejvýše tři operandy. Tento typ je blízký instrukcím RISC procesorů.

SSA forma (Static Single Assignment) — varianta tříadresového kódu, kde každá proměnná je přiřazena právě jednou. SSA zjednodušuje mnoho optimalizací, zejména analýzu datového toku.

Zásobníkový kód — instrukce pracují s implicitním zásobníkem, podobně jako bajtkód JVM nebo WebAssembly. Tento typ je přirozený pro zásobníkové jazyky.

Graf toku řízení (Control Flow Graph, CFG) — program je reprezentován jako orientovaný graf, kde uzly jsou základní bloky a hrany reprezentují možné přechody mezi nimi.

2.2.2.2 Mezikód kompilátoru Roth.

Kompilátor Roth používá zásobníkový mezikód, který přirozeně odpovídá sémantice zdrojového jazyka. Každá instrukce IR popisuje operaci nad datovým zásobníkem a je anotována svým

stack effect — počtem hodnot, které odebírá ze zásobníku, a počtem hodnot, které na zásobník ukládá.

IR programu v kompilátoru Roth se skládá z následujících komponent:

- **IRProgram** — reprezentuje celý program, obsahuje hlavní funkci a slovník uživatelských funkcí.
- **IRFunction** — reprezentuje jednu funkci (slovo), obsahuje seznam instrukcí a informaci o stack effect.
- **IRInstruction** — jednotlivá instrukce mezikódu.

Tabulka 2 uvádí přehled kategorií instrukcí mezikódu.

Tabulka 2: Kategorie instrukcí mezikódu

Kategorie	**Příklady**	**Popis**
Zásobníkové	Push, Pop, Dup, Swap	Manipulace se zásobníkem
Aritmetické	Add, Sub, Mul, Div	Aritmetické operace
Porovnávací	Equal, Less, Greater	Relační operace
Logické	And, Or, Not	Booleovské operace
Řídicí tok	Jump, JumpIf, Call	Větvení a volání
Paměťové	Load, Store	Práce s proměnnými
I/O	Print, ReadChar	Vstup a výstup

2.2.3 Optimalizace

Optimalizace jsou transformace programu, které zachovávají jeho sémantiku, ale zlepšují některou z metrik kvality — typicky rychlost běhu nebo velikost kódu. Kompilátor Roth implementuje několik standardních optimalizačních technik.

2.2.3.1 Constant folding. Vyhodnocení konstantních výrazů v době kompilace. Například sekvence instrukcí `Push(5) Push(3) Add` může být nahrazena jedinou instrukcí `LoadConst(8)`. Tato optimalizace je zvláště účinná v kombinaci s inliningem.

2.2.3.2 Dead code elimination. Odstranění kódu, který nemá vliv na výsledek programu. Typickým příkladem je hodnota uložená na zásobník, která je ihned odstraněna bez použití: `Push(42) Drop` lze zcela eliminovat.

2.2.3.3 Peephole optimalizace. Lokální optimalizace, které hledají a nahrazují krátké vzory instrukcí efektivnějšími ekvivalenty. Příklady:

- `Dup Add` → násobení dvěma

- `Swap Swap` → odstranění obou instrukcí
- `Push(a) Push(b) Swap` → `Push(b) Push(a)`

2.2.3.4 Strength reduction. Náhrada drahých operací levnějšími ekvivalenty. Například násobení dvěma lze nahradit sečtením hodnoty se sebou samou (`Dup Add`), což může být na některých platformách rychlejší než obecné násobení.

2.2.3.5 Function inlining. Nahrazení volání funkce jejím tělem. Inlining eliminuje režii volání funkce a umožňuje další optimalizace v kontextu volajícího kódu. Kompilátor Roth provádí inlining pro malé funkce bez řídicích struktur.

2.2.3.6 Iterativní optimalizace. Optimalizační průchody se typicky spouštějí opakovaně, dokud se program stabilizuje (žádný průchod neprovede změnu). Jedna optimalizace totiž může vytvořit příležitosti pro jinou. Například inlining může vytvořit nové konstantní výrazy, které lze následně zoptimalizovat pomocí constant foldingu.

Kompilátor Roth implementuje optimalizační pipeline, která spouští průchody v definovaném pořadí:

1. Function inlining — nejdříve vložit těla funkcí
2. Constant folding — vyhodnotit konstantní výrazy
3. Peephole optimalizace — aplikovat lokální vzory
4. Strength reduction — nahradit drahé operace
5. Dead code elimination — odstranit mrtvý kód

Pipeline se opakuje až do dosažení fixního bodu nebo maximálního počtu iterací (standardně 10).

2.3 Programovací jazyk Rust

Rust je moderní systémový programovací jazyk vyvinutý společností Mozilla a poprvé vydaný v roce 2015. Kombinuje výkon jazyků jako C a C++ s bezpečností paměti garantovanou v době kompilace, bez nutnosti garbage collectoru. Pro implementaci kompilátoru nabízí Rust několik vlastností, které činí vývoj efektivnějším a bezpečnějším.

2.3.1 Algebraické datové typy

Rust poskytuje plnou podporu pro algebraické datové typy (ADT) prostřednictvím konstrukcí `struct` (součinové typy) a `enum` (součtové typy). Tyto konstrukce jsou ideální pro reprezentaci datových struktur kompilátoru.

2.3.1.1 Reprezentace AST.

Abstraktní syntaktický strom lze v Rustu elegantně vyjádřit pomocí `enum`, kde každá varianta reprezentuje jeden typ uzlu:

```
pub enum AstNode {
    Number(i32, Position),
    Word(String, Position),
    StringLiteral(String, Position),
    Definition { name: String, body: Vec<AstNode>, position: Position },
    Program(Vec<AstNode>),
}
```

Tato reprezentace má několik výhod oproti tradičním objektově orientovaným hierarchiím:

- Kompilátor garantuje, že všechny varianty jsou ošetřeny při pattern matchingu.
- Přidání nové varianty vyžaduje úpravu všech míst, kde se AST zpracovává — kompilátor na chybějící případy upozorní.
- Data asociovaná s variantou jsou přímo součástí typu, bez nutnosti dodatečných alokací.

2.3.1.2 Reprezentace mezikódu.

Obdobně je implementován mezikód, kde každá instrukce je variantou typu `IRInstruction`:

```
pub enum IRInstruction {
    Push(IRValue), Pop, Dup, Drop, Add, Sub, Mul, Div,
    Jump(IRLabel), JumpIf(IRLabel), Call(String), Return,
    // ... dalsi instrukce
}
```

2.3.2 Pattern matching

Pattern matching je v Rustu implementován prostřednictvím konstrukce `match`, která umožňuje elegantní dekompozici algebraických datových typů. Na rozdíl od konstrukce `switch` v jazycích jako C nebo Java je pattern matching v Rustu:

- **Exhaustivní** — kompilátor ověřuje, že jsou pokryty všechny možné případy.
- **Destrukturující** — umožňuje extrahovat hodnoty z vnořených struktur.
- **S guardy** — podmínky lze kombinovat s vzory.

Příklad použití pattern matchingu v optimalizátoru:

```
fn try_fold_binary_op(&self, op: &IRInstruction,
                      a: i32, b: i32) -> Option<IRInstruction> {
    match op {
        IRInstruction::Add => Some(IRInstruction::LoadConst(a + b)),
```

```
            IRInstruction::Sub => Some(IRInstruction::LoadConst(a - b)),
            IRInstruction::Mul => Some(IRInstruction::LoadConst(a * b)),
            IRInstruction::Div if b != 0 =>
                Some(IRInstruction::LoadConst(a / b)),
            _ => None,
        }
    }
```

Výraz `if b != 0` je guard, který přidává dodatečnou podmínku k vzoru. Varianta `_` zachytává všechny neošetřené případy.

2.3.3 Ownership a bezpečnost paměti

Rust garantuje bezpečnost paměti bez garbage collectoru pomocí systému ownership. Každá hodnota v Rustu má právě jednoho vlastníka a je automaticky uvolněna, když vlastník opustí svůj scope.

2.3.3.1 Pravidla ownership.

1. Každá hodnota má právě jednoho vlastníka.
2. Když vlastník opustí scope, hodnota je uvolněna.
3. Vlastnictví lze přenést (move) nebo vypůjčit (borrow).

2.3.3.2 Výpůjčky (borrowing). Rust rozlišuje dva typy výpůjček:

- **Sdílená výpůjčka** (`&T`) — umožňuje čtení, ale ne modifikaci. Může existovat libovolné množství sdílených výpůjček současně.
- **Exkluzivní výpůjčka** (`&mut T`) — umožňuje modifikaci. V daném okamžiku může existovat pouze jedna exkluzivní výpůjčka a žádná sdílená.

Tato pravidla eliminují celé třídy chyb běžných v C a C++: data races, use-after-free, double-free a dangling pointery. Pro kompilátor to znamená, že transformace nad AST a IR jsou automaticky bezpečné — kompilátor Rustu ověří, že žádná část kódu nemodifikuje data, která jsou současně čtena jinde.

2.3.4 Traity a polymorfismus

Traity v Rustu jsou obdobou interfaces v Javě nebo type classes v Haskellu. Definují sadu metod, které musí typ implementovat.

V kompilátoru Roth jsou traity použity pro definici rozhraní optimalizačních průchodů:

```
pub trait IROptimizationPass {
    fn name(&self) -> &str;
    fn optimize_program(&mut self, prog: &mut IRProgram) -> bool;
    fn optimize_function(&mut self, func: &mut IRFunction) -> bool;
}
```

Každý optimalizační průchod implementuje tento trait, což umožňuje jejich jednotné zpracování v optimalizační pipeline:

```
pub struct IROptimizer {
    passes: Vec<Box<dyn IROptimizationPass>>,
    max_iterations: usize,
}
```

Konstrukce `Box<dyn IROptimizationPass>` reprezentuje dynamicky typovaný ukazatel na objekt implementující daný trait (trait object), což umožňuje heterogenní kolekce různých optimalizačních průchodů.

2.3.5 Procedurální makra

Rust podporuje procedurální makra, která umožňují generování kódu v době kompilace. Na rozdíl od textových maker v C pracují procedurální makra s tokenizovaným vstupem a produkují validní Rust kód.

V kompilátoru Roth jsou procedurální makra použita pro automatické odvození stack effects instrukcí mezikódu. Místo manuální implementace metody `stack_effect()` pro každou instrukci:

```
#[derive(StackEffect)]
pub enum IRInstruction {
    #[stack_effect(consumes = 0, produces = 1)]
    Push(IRValue),
    #[stack_effect(consumes = 1, produces = 0)]
    Drop,
    #[stack_effect(consumes = 2, produces = 1)]
    Add,
    // ...
}
```

Makro `#[derive(StackEffect)]` automaticky vygeneruje implementaci metody, která vrací správný stack effect pro každou variantu instrukce. Toto řešení:

- Eliminuje duplicitu kódu a možnost nekonzistence.
- Přesouvá metadata blíže k definici dat.
- Umožňuje validaci v době kompilace.

2.3.6 Správa závislostí a ekosystém

Rust disponuje integrovaným systémem pro správu závislostí a sestavování projektů nazvaným Cargo. Projekt je definován souborem `Cargo.toml`, který specifikuje:

- Metadata projektu (název, verze, autoři)

- Externí závislosti a jejich verze
- Konfiguraci kompilace (profily, features)

Pro kompilátor Roth Cargo zajišťuje:

- Reprodukovatelné sestavení díky souboru `Cargo.lock`.
- Automatické stažení a kompilaci závislostí.
- Integraci s testovacím frameworkem (`cargo test`).
- Generování dokumentace (`cargo doc`).

Ekosystém Rustu (crates.io) poskytuje tisíce knihoven, z nichž kompilátor Roth využívá zejména knihovny pro dynamické načítání knihoven (pro JIT kompilaci v REPL).

3 Analýza a návrh

3.1 Požadavky na kompilátor

Před samotnou implementací je nutné definovat požadavky na kompilátor Roth. Tyto požadavky lze rozdělit na funkční (co má systém dělat) a nefunkční (jaké vlastnosti má systém mít).

3.1.1 Funkční požadavky

3.1.1.1 Kompilace zdrojového kódu. Kompilátor musí být schopen přeložit zdrojový kód v jazyce Roth do cílového jazyka. Vstupem je soubor s příponou `.rt` obsahující program v jazyce Roth, výstupem je soubor v jazyce Rust nebo C, který lze následně zkompilovat standardními nástroji.

3.1.1.2 Podpora více cílových platforem. Systém musí podporovat generování kódu pro více cílových jazyků:

- **Rust** — primární cílový jazyk, využívající typovou bezpečnost a optimalizace kompilátoru `rustc`,
- **C** — sekundární cílový jazyk pro maximální portabilitu.

3.1.1.3 Interaktivní prostředí REPL. Kompilátor musí poskytovat interaktivní prostředí typu Read-Eval-Print Loop, které umožňuje:

- okamžité vyhodnocování výrazů,
- definování nových slov s perzistencí mezi příkazy,
- zobrazování stavu zásobníku po každém příkazu.

3.1.1.4 Optimalizace kódu.

Kompilátor musí implementovat optimalizační průchody nad mezikódem, které zlepšují výkon generovaného programu bez změny jeho sémantiky.

3.1.1.5 Sémantická kontrola.

Před generováním kódu musí kompilátor provádět sémantickou analýzu, která kontroluje:

- definovanost všech použitých slov,
- nepřípustnost redefinice vestavěných slov,
- správnost syntaktických konstrukcí (párování `IF`/`THEN`, `DO`/`LOOP` atd.).

3.1.1.6 Podpora pro inkluze souborů.

Kompilátor musí podporovat direktivu `INCLUDE` pro vkládání obsahu externích souborů, včetně detekce cyklických závislostí.

3.1.2 Nefunkční požadavky

3.1.2.1 Rozšiřitelnost.

Architektura kompilátoru musí umožňovat snadné přidávání:

- nových optimalizačních průchodů,
- nových cílových platforem (backendů),
- nových vestavěných slov.

3.1.2.2 Modularita.

Jednotlivé fáze kompilace musí být odděleny do samostatných modulů s jasně definovanými rozhraními. To umožňuje nezávislé testování a vývoj jednotlivých částí.

3.1.2.3 Diagnostika chyb.

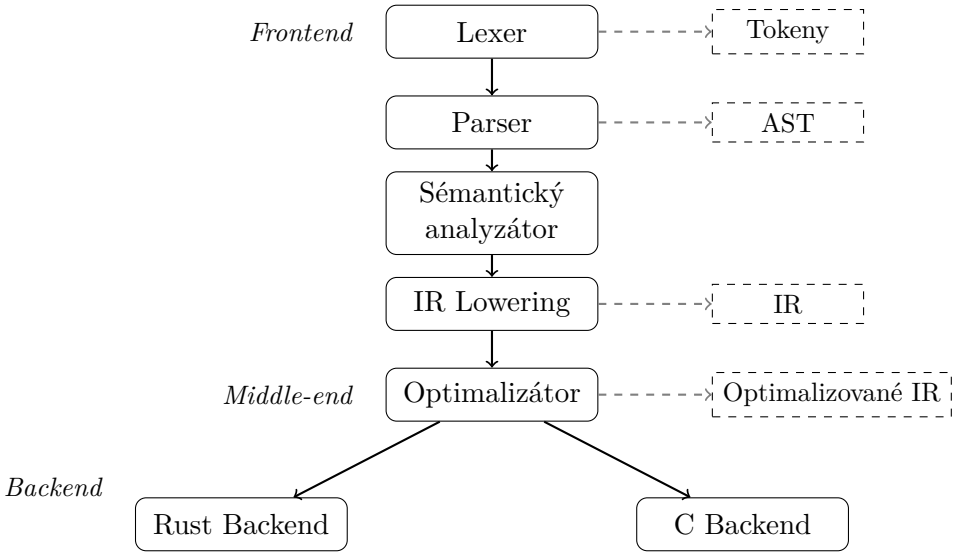
Chybové zprávy musí obsahovat informace o pozici chyby ve zdrojovém kódu (řádek, sloupec) a srozumitelný popis problému.

3.1.2.4 Výkonnost.

Generovaný kód by měl být srovnatelný s výkonem existujících implementací jazyka Forth. Samotná kompilace by měla být dostatečně rychlá pro interaktivní použití v REPL.

3.2 Architektura systému

Kompilátor Roth je navržen jako víceprůchodový kompilátor s explicitní reprezentací mezikódu. Architektura systému je znázorněna na obrázku 2.



Obrázek 2: Architektura kompilátoru Roth

3.2.1 Frontend

Frontend kompilátoru zajišťuje analýzu zdrojového kódu a jeho transformaci do abstraktní reprezentace. Skládá se z následujících komponent:

3.2.1.1 Lexer. Lexikální analyzátor (`lexer.rs`) převádí vstupní text na proud tokenů. Rozpoznává čísla, slova, řetězce, komentáře a speciální znaky pro definice slov (`:` a `;`).

3.2.1.2 Parser. Syntaktický analyzátor (`parser.rs`) konstruuje abstraktní syntaktický strom (AST) z proudu tokenů. AST reprezentuje hierarchickou strukturu programu pomocí následujících typů uzlů:

`Program` — kořenový uzel obsahující seznam příkazů,

`Definition` — definice uživatelského slova,

`Number` — číselná konstanta,

`Word` — volání slova,

`StringLiteral` — řetězcová konstanta,

`VariableDeclaration` — deklarace proměnné.

3.2.1.3 Sémantický analyzátor. Modul `analyzer.rs` provádí sémantickou kontrolu AST. Udržuje tabulku definovaných slov (vestavěných i uživatelských) a proměnných. Kontroluje, zda jsou všechna použitá slova definována a zda nedochází k redefinici vestavěných slov.

3.2.2 Middle-end

Middle-end kompilátoru pracuje s mezikódem (IR) a provádí platformově nezávislé transformace.

3.2.2.1 IR Lowering. Modul `ir_lowering.rs` transformuje AST do mezikódu. Tato transformace zahrnuje:

- překlad slov na odpovídající IR instrukce,
- generování návěstí pro řídicí struktury,
- alokaci proměnných.

3.2.2.2 Optimalizátor. Modul `ir_optimizer.rs` implementuje sadu optimalizačních průchodů, které transformují IR za účelem zlepšení výkonu. Optimalizátor používá iterativní přístup — průchody se opakují, dokud nedojde k dosažení fixního bodu.

3.2.3 Backend

Backend kompilátoru generuje cílový kód z optimalizovaného IR. Každý backend implementuje společné rozhraní definované traitem `CodeGenerator`:

Zdrojový kód 1: Rozhraní generátoru kódu

```
pub trait CodeGenerator {
    fn generate(&mut self, ast: &AstNode) -> String;
    fn get_file_extension(&self) -> &str;
    fn get_compile_command(&self, filename: &str) -> String;
}
```

3.2.3.1 Rust Backend. Generuje idiomatický kód v jazyce Rust využívající vektor jako zásobník. Podporuje všechny IR instrukce včetně řídicích struktur a volání funkcí.

3.2.3.2 C Backend. Generuje kód v jazyce C s manuální správou zásobníku pomocí pole. Tento backend je vhodný pro prostředí bez podpory Rustu.

3.3 Návrh mezikódu

Mezikód (IR) kompilátoru Roth je navržen jako zásobníkový kód, který přirozeně odpovídá sémantice zdrojového jazyka. Každá instrukce má explicitně definovaný *stack effect* — počet hodnot, které odebírá ze zásobníku, a počet hodnot, které na zásobník ukládá.

3.3.1 Struktura IR

IR programu se skládá ze tří hlavních komponent:

3.3.1.1 IRProgram. Reprezentuje celý program. Obsahuje hlavní funkci (`main`) a slovník uživatelských funkcí.

Zdrojový kód 2: Struktura IR programu

```
pub struct IRProgram {
    pub functions: HashMap<String, IRFunction>,
    pub main: IRFunction,
}
```

3.3.1.2 IRFunction. Reprezentuje jednu funkci (slovo). Obsahuje název, seznam instrukcí a informaci o stack effect celé funkce.

Zdrojový kód 3: Struktura IR funkce

```
pub struct IRFunction {
    pub name: String,
    pub instructions: Vec<IRInstruction>,
    pub stack_effect: StackEffect,
}
```

3.3.1.3 StackEffect. Popisuje vliv instrukce nebo funkce na zásobník.

Zdrojový kód 4: Stack effect

```
pub struct StackEffect {
    pub consumes: usize,   // Pocet hodnot odebranych ze zasobniku
    pub produces: usize,   // Pocet hodnot ulozenych na zasobnik
}
```

3.3.2 Instrukční sada

Instrukční sada IR je navržena tak, aby pokrývala všechny konstrukce jazyka Roth a zároveň umožňovala efektivní optimalizaci. Tabulka 3 uvádí kompletní seznam instrukcí.

3.3.3 Stack Effect anotace

Pro automatické odvození stack effect jednotlivých instrukcí využívá kompilátor Roth procedurální makro `#[derive(StackEffect)]`. Toto makro generuje implementaci metody `stack_effect()` na základě anotací `#[stack_effect(consumes = N, produces = M)]`:

Zdrojový kód 5: Příklad anotace stack effect

```
#[derive(Debug, Clone, PartialEq, StackEffect)]
pub enum IRInstruction {
    #[stack_effect(consumes = 0, produces = 1)]
    Push(IRValue),
    #[stack_effect(consumes = 2, produces = 1)]
    Add,
    // ...
}
```

Tato informace je využívána optimalizátorem pro analýzu datového toku a detekci mrtvého kódu.

3.3.4 Hodnoty v IR

Mezikód pracuje s několika typy hodnot reprezentovaných výčtem `IRValue`:

Zdrojový kód 6: Typy hodnot v IR

```
pub enum IRValue {
    Constant(i32),      // Ciselna konstanta
    StackTop,           // Vrchol zasobniku
    StackPos(usize),    // Pozice na zasobniku (0 = vrchol)
    Variable(String),   // Pojmenovana promenna
    Temporary(usize),   // Docasna hodnota s ID
}
```

3.4 Návrh optimalizačních průchodů

Optimalizátor kompilátoru Roth je navržen jako rozšiřitelná sada průchodů, které transformují IR. Každý průchod implementuje společný trait `IROptimizationPass`.

3.4.1 Rozhraní optimalizačního průchodu

Zdrojový kód 7: Trait pro optimalizační průchody

```
pub trait IROptimizationPass {
    fn name(&self) -> &str;
    fn optimize_program(&mut self, program: &mut IRProgram) -> bool;
    fn optimize_function(&mut self, function: &mut IRFunction) -> boo
}
```

Metoda `optimize_program` vrací `true`, pokud průchod provedl nějakou změnu. Tato informace je využívána pro iterativní optimalizaci.

3.4.2 Implementované průchody

Kompilátor Roth implementuje pět optimalizačních průchodů:

3.4.2.1 Function Inlining.

Nahrazuje volání malých funkcí jejich tělem. Funkce je považována za vhodnou k inliningu, pokud:

- má méně než 20 instrukcí,
- neobsahuje rekurzivní volání,
- neobsahuje řídicí struktury (skoky, návěstí).

Inlining eliminuje režii volání funkce a vytváří příležitosti pro další optimalizace (např. constant folding).

3.4.2.2 Constant Folding.

Vyhodnocuje konstantní výrazy v době kompilace. Průchod hledá vzory typu:

`Push(a) Push(b) BinaryOp` → `LoadConst(result)`

Podporuje všechny aritmetické, porovnávací a logické operace. Dělení nulou je zachováno pro runtime detekci chyby.

3.4.2.3 Peephole Optimization.

Aplikuje lokální transformace na krátké sekvence instrukcí:

- `Push(a) Dup Add` → `LoadConst(a*2)`
- `Push(a) Push(b) Swap` → `Push(b) Push(a)`
- `Swap Swap` → odstranění obou instrukcí
- `Dup Drop` → odstranění obou instrukcí

3.4.2.4 Strength Reduction.

Nahrazuje drahé operace levnějšími ekvivalenty:

- `Push(0) Add` → odstranění (identita)
- `Push(1) Mul` → odstranění (identita)
- `Push(0) Mul` → `Drop LoadConst(0)`
- `Push(2) Mul` → `Dup Add`

3.4.2.5 Dead Code Elimination.

Odstraňuje instrukce, které nemají vliv na výsledek programu:

- instrukce `Nop`,
- sekvence `Push Drop` (hodnota není použita),
- sekvence `Dup Drop` (efektivně no-op).

3.4.3 Optimalizační pipeline

Průchody jsou organizovány do pipeline, která je spouštěna iterativně:

Zdrojový kód 8: Konfigurace optimalizační pipeline

```
pub struct IROptimizer {
    passes: Vec<Box<dyn IROptimizationPass>>,
    max_iterations: usize,  // Vychozi: 10
}
```

Pořadí průchodů je navrženo tak, aby maximalizovalo efektivitu optimalizací:

1. **Function Inlining** — nejdříve vložit těla funkcí, aby byly dostupné pro další optimalizace
2. **Constant Folding** — vyhodnotit konstantní výrazy (včetně těch vzniklých inliningem)
3. **Peephole Optimization** — aplikovat lokální vzory
4. **Strength Reduction** — nahradit drahé operace
5. **Dead Code Elimination** — odstranit mrtvý kód (včetně Nop instrukcí z peephole optimalizací)

Pipeline se opakuje, dokud žádný průchod neprovede změnu, nebo dokud není dosažen maximální počet iterací (standardně 10). Toto iterativní vyhodnocování umožňuje, aby optimalizace spolupracovaly — například inlining může odhalit nové konstantní výrazy pro constant folding.

3.4.4 Příklad optimalizace

Uvažujme následující program v jazyce Roth:

```
: DOUBLE DUP + ;
5 DOUBLE .
```

Po IR lowering:

```
function DOUBLE:
  Dup
  Add
  Return

function main:
  Push(5)
  Call(DOUBLE)
  Print
```

Po inliningu:

```
function main:
  Push(5)
  Dup
  Add
  Print
```

Po peephole optimalizaci (`Push Dup Add` → `LoadConst`):

```
function main:
  LoadConst(10)
  Print
```

Výsledný kód obsahuje pouze dvě instrukce místo původních čtyř, přičemž veškeré výpočty byly provedeny v době kompilace.

4 Implementace

4.1 Lexikální analýza

Lexikální analyzátor převádí vstupní text na sekvenci tokenů, které jsou následně zpracovány parserem. V kompilátoru
Roth je lexer implementován v modulu `src/lexer.rs` jako struktura `Lexer`, která si uchovává: (1) celý vstup jako řetězec, (2) aktuální pozici ve vstupu a (3) informace o řádku a sloupci pro diagnostiku chyb.

Každý token nese kromě svého typu také původní text (pole `raw`) a pozici `Position` (řádek, sloupec a offset ve vstupu). Tato volba zjednodušuje generování chybových hlášení ve všech následujících fázích.

Typy tokenů jsou definovány výčtem `TokenType` v `src/types.rs`. Lexer rozpoznává celočíselné literály (`i32`), identifikátory (slova), oddělovače definic (`:` a `;`), komentáře v závorkách `(...)` a řetězcové literály.

Specifickým prvkem inspirovaným FORTH je podpora syntaxe `S"..."`, která umožňuje zapisovat řetězec bez explicitního uzavírání do tokenů jednotlivých slov. Lexer tento vzor detekuje ještě před obecným rozlišením tokenu.

Metoda `read_token` načte nejdelší podřetězec oddělený bílými znaky a pokusí se jej převést na číslo. Pokud převod selže, token je interpretován jako slovo a normalizován na velká písmena (`to_uppercase`). To sjednocuje zpracování slov nezávisle na velikosti písmen ve zdrojovém kódu.

Komentáře v závorkách jsou tokenizovány jako `TokenType::Comment`. Parser i další fáze je následně mohou ignorovat nebo využít (např. pro zachování informativních poznámek v IR ve formě `Comment` instrukcí).

4.2 Syntaktická analýza

Parser (`src/parser.rs`) je implementován jako jednoduchý rekurzivně-sestupný analyzátor nad vektorem tokenů. Vzhledem k tomu, že zásobníkový jazyk má lineární zápis (program je sekvence slov a literálů), je výsledný abstraktní syntaktický strom (AST) záměrně mělký.

Zdrojový kód 9: Rozpoznání tokenu v lexeru

```
fn next_token(&mut self, start_pos: Position) -> Result<Token, ParseE
    let ch = self.current_char();
    if (ch == 'S' || ch == 's') && self.peek_char() == Some('"') {
        return self.read_s_quote_literal(start_pos);
    }
    match ch {
        '(' => self.read_comment(start_pos),
        '"' => self.read_string_literal(start_pos),
        ':' | ';' => self.read_definition_token(start_pos),
        _ => self.read_token(start_pos),
    }
}
```

Kořenem AST je uzel `Program`, který obsahuje seznam příkazů. Jedinou hierarchickou konstrukcí jsou definice slov uzavřené mezi `:` a `;`, které jsou reprezentovány uzlem `Definition` s vlastním tělem.

Základní uzly AST jsou definovány v `src/types.rs`.

Zdrojový kód 10: Základní uzly AST

```
pub enum AstNode {
    Number(i32, Position),
    Word(String, Position),
    StringLiteral(String, Position),
    Definition { name: String, body: Vec<AstNode>, position: Position
    VariableDeclaration { name: String, position: Position },
    Program(Vec<AstNode>),
}
```

Parser v celém programu ignoruje tokeny typu `Comment`. Tím se snižuje komplexita syntaktické analýzy bez ztráty informace o pozicích chyb, protože pozice je nesena i ostatními tokeny.

Definice slova je parsována následovně: po `:` musí následovat jméno slova, poté parser postupně načítá uzly těla až do `;`. Vnořené definice jsou zakázány a jsou vyhodnoceny jako syntaktická chyba.

Parser zároveň implementuje speciální pravidlo pro konstrukci `VARIABLE`. Pokud se v toku tokenů objeví slovo `VARIABLE`, parser očekává následující identifikátor, který uloží do uzlu `VariableDeclaration`. Vlastní práce s proměnnými je řešena až v nižších fázích (sémantika a IR).

4.3 Sémantická analýza

Sémantická analýza (`src/analyzer.rs`) ověřuje, že program používá pouze existující slova a že uživatelské definice neporušují pravidla jazyka. V kompilátoru

Roth je analýza založena na udržování tří tabulek: `builtin_words`, `defined_words` a `defined_variables`.

Při inicializaci analyzátoru je tabulka vestavěných slov naplněna seznamem primitivních operací (aritmetika, práce se zásobníkem, I/O, řízení toku, práce s pamětí). Vestavěná slova jsou považována za součást definice jazyka a nelze je redefinovat.

Analýza prochází AST rekurzivně:

- při uzlu `Definition` ověří, že jméno není vestavěné, následně přidá slovo mezi definovaná ještě *před* analýzou těla; tím je umožněna rekurze pomocí `RECURSE`,
- při uzlu `Word` ověří, že se jedná o vestavěné slovo, uživatelsky definované slovo, nebo proměnnou,
- při `VariableDeclaration` registruje proměnnou v tabulce.

V režimu REPL je tato fáze rozšířena o kontext předchozích vstupů: před voláním `analyze` jsou do analyzátoru doplněny již definované slova a proměnné, aby bylo možné na ně navazovat (viz sekce 4.7).

4.4 Generování mezikódu

Mezikód (IR) je generován v modulu `src/ir_lowering.rs` transformací AST na strukturu `IRProgram`. Implementace používá pomocnou strukturu `IRBuilder` (`src/ir.rs`), která umožňuje postupně emitovat instrukce do právě budované funkce.

Lowering je realizován ve třech krocích, které odpovídají přirozené struktuře Forth-like programu:

1. nejprve se sesbírají všechny definice slov do mapy `word_definitions`,
2. následně se pro každé slovo vytvoří samostatná IR funkce,
3. nakonec je přeložena hlavní část programu (kód mimo definice) do funkce `main`.

Číselné literály jsou překládány na instrukci `Push(Constant)`. Řetězcové literály jsou v současné implementaci reprezentovány jako sekvence znaků uložených na zásobník (ASCII kódy) následovaná délkou řetězce; to odpovídá očekávanému stack efektu slova `TYPE`.

4.4.0.1 Proměnné. Při zpracování uzlu `VariableDeclaration` je proměnné přidělena celočíselná adresa z rostoucího čítače. Při použití proměnné jako slova (tj. token `Word` odpovídá jménu proměnné) lowering vloží na zásobník její adresu. Operace `@` a `!` jsou mapovány na instrukce `Load` a `Store`.

4.4.0.2 Řídicí struktury. Základní struktury `IF/ELSE/THEN` a `DO/LOOP` jsou překládány na instrukce se skoky a návěstími (`Jump`, `JumpIfNot`, `Label`) pomocí generování unikátních návěstí v `IRBuilder`. Pro správné párování konstrukcí lowering udržuje pomocné zásobníky (`loop_stack` a `conditional_stack`).

4.4.0.3 Výpočet stack effect. Po vygenerování IR je nad funkcemi spuštěna analýza stack efektu (`StackEffectAnalyzer`), která z anotací instrukcí (procedurální makro `#[derive(StackEffect)]` v `roth-derive`) vypočítá čistou změnu hloubky zásobníku pro každou funkci. Tato informace je později využita optimalizátorem i pro ladicí výpisy.

4.5 Optimalizace

4.5.1 Constant folding

Průchod constant folding (`ConstantFoldingPass` v `src/ir_optimizer.rs`) vyhledává lokální vzory typu `Push(a) Push(b) op` a nahrazuje je jednou instrukcí `LoadConst(result)`. Implementace podporuje aritmetické, relační i logické operace. Pro booleovské výsledky je použita konvence FORTH: `-1` znamená pravdu a `0` nepravdu.

4.5.2 Dead code elimination

Eliminace mrtvého kódu (`DeadCodeEliminationPass`) odstraňuje instrukce, které nemají vliv na výsledek výpočtu. V aktuální implementaci se jedná zejména o `Nop` a krátké sekvence jako `Push Drop` nebo `Dup Drop`, které vznikají jako vedlejší produkt jiných optimalizací.

4.5.3 Peephole optimalizace

Peephole optimalizace (`PeepholeOptimizationPass`) pracuje nad malými okny instrukcí (typicky délky 2–3). Cílem je nahradit časté sekvence efektivnějším ekvivalentem, případně vytvořit příležitost pro následné odstranění instrukcí (např. nahrazení dvojice instrukcí `Dup Drop` za `Nop`, který odstraní DCE).

4.5.4 Strength reduction

Strength reduction (`StrengthReductionPass`) implementuje standardní algebraické identity a zjednodušení, například násobení dvěma na `Dup Add` nebo eliminaci neutrálních prvků (`+ 0`, `* 1`).

4.5.5 Inlining

Inlining (`FunctionInliningPass`) nahrazuje instrukci `Call` tělem volané funkce, pokud je funkce dostatečně malá (výchozí limit 20 instrukcí) a neobsahuje

skoky ani rekurzivní volání. Průchod je implementován jako globální optimalizace nad celým `IRProgram`, protože potřebuje přístup k mapě všech funkcí.

Optimalizační pipeline (`IROptimizer`) spouští průchody iterativně, dokud nedojde ke stabilizaci, nebo dokud není překročen maximální počet iterací (výchozí hodnota 10). Iterativní spouštění je důležité například proto, že inlining může vytvořit nové konstantní výrazy vhodné pro constant folding.

4.6 Generování cílového kódu

4.6.1 Backend pro Rust

Generování cílového kódu pro Rust je implementováno dvěma cestami. První cesta (`rust-ir`) používá přímočarý generátor `IRRustGenerator` (`src/ir_codegen.rs`), který převádí IR do samostatného spustitelného programu v Rust. Vygenerovaný kód obsahuje strukturu `OptimizedForth` s datovým zásobníkem `Vec<i32>` a metodami, které implementují jednotlivá uživatelská slova jako vnořené funkce.

Pro instrukce řídicího toku generátor používá jednoduchý stavový automat nad čítačem instrukcí (`__pc`), což umožňuje reprezentovat skoky `Jump`/`JumpIf`/`JumpIfNot` i bez explicitních `goto`.

Druhá cesta je novější modulární framework (`src/codegen/*`), který odděluje překlad instrukcí (`IRTranslator`) od emitování kódu (`CodeEmitter`) a od specifik cílového jazyka (`TargetLanguage`). V této práci je tento framework využit především pro experimenty a rozšiřitelnost.

4.6.2 Backend pro C

Backend pro jazyk C (`c-ir`) je implementován generátorem `IRCGenerator` v `src/ir_codegen.rs`. Vygenerovaný kód implementuje zásobník jako pole pevné velikosti `STACK_SIZE` a používá pomocné funkce `push`/`pop`. Tato varianta je vhodná pro jednodušší portaci a snadné spuštění na systémech bez toolchainu pro Rust.

Stejně jako u Rust backendu je možné použít i modulární variantu backendu (`c-modular`), která je registrována v `codegen/registry.rs`.

4.7 REPL s JIT kompilací

REPL je implementován v `src/repl/` a realizuje tzv. *JIT-like* přístup: každý vstup uživatele je přeložen do Rust kódu, zkompilován do sdílené knihovny (`cdylib`) a načten za běhu programu.

Vstupní řetězec prochází stejnými fázemi jako kompilace souboru: lexing, parsing, sémantická analýza, IR lowering a optimalizace. Následně se použije specializovaný generátor `ReplCodegen` (`src/repl/codegen.rs`), který vytvoří kód exportující funkci `__repl_entry`. Ta přijímá `RuntimeContext` z knihovny `roth-runtime` a vykoná instrukce nad perzistentním zásobníkem a pamětí.

Kompilace a dynamické načítání jsou zapouzdřeny v `LibraryLoader` (`src/repl/loade` Loader zapisuje vygenerovaný Rust do dočasného adresáře, spustí `rustc -crate-type=cdy` a výslednou knihovnu načte pomocí knihovny `libloading`. REPL si uchovává všechny načtené knihovny v paměti, aby nedošlo k uvolnění symbolů.

Perzistentní stav mezi příkazy je rozdělen na:

- **runtime kontext** (`RuntimeContext`) — zásobník, paměť proměnných a registry slov,
- **kompilační kontext** (`CompilerContext`) — slovník již definovaných slov v IR a množinu deklarovaných proměnných. Tyto informace umožňují sémantické ověření a IR lowering s ohledem na historii REPL.

4.8 Standardní knihovna

Standardní knihovna je uložena ve složce `std/` a je psaná přímo v jazyce Roth. Hlavní soubor `std/std.rt` postupně načítá jednotlivé moduly pomocí direktivy `INCLUDE`.

Knihovna je rozdělena do tematických souborů:

- `core.rt` — základní konstanty a utility,
- `stack.rt` — rozšířené zásobníkové operace,
- `math.rt` — matematické funkce,
- `io.rt` — formátovaný výstup a I/O pomocné funkce,
- `control.rt` — rozšířené řídicí konstrukce,
- `compiler.rt` — dokumentace slov implementovaných na úrovni kompilátoru.

V offline kompilaci je direktiva `INCLUDE` zpracována ještě před lexikální analýzou (funkce `preprocess_includes` v `src/main.rs`). Preprocesor rekurzivně rozbalí všechny inkluze, detekuje cykly a vloží do výsledného zdroje značky ve formě komentářů, aby bylo možné zpětně dohledat původ vloženého kódu.

5 Testování a vyhodnocení

5.1 Testovací metodika

Testování kompilátoru bylo rozděleno do dvou úrovní:

- **jednotkové testy** ověřují jednotlivé moduly (lexer, parser, sémantická analýza, IR, optimalizace a codegen) na malých izolovaných případech,

- **integrační testy** ověřují celý překladový řetězec přes veřejné CLI rozhraní a kontrolují, že je vygenerovaný kód vytvořen a obsahuje očekávané konstrukce.

Jednotkové testy jsou implementovány ve dvou formách: (1) testy přímo v modulech v `src/` pomocí `#[cfg(test)]` a (2) testy v adresáři `tests/`, kde jsou jednotlivé části kompilátoru testovány přes veřejné API knihovny `roth`.

Integrační testy (`tests/integration_tests.rs`) spouštějí binární program přes `cargo run` a ověřují vznik výstupu v adresáři `.build/`. Tím se simuluje běžné použití kompilátoru z příkazové řádky.

Všechny testy lze spustit příkazem:

Zdrojový kód 11: Spuštění testů

```
cargo test
```

5.2 Výsledky testování

Testovací sada pokrývá základní syntaktické konstrukce jazyka, práci s datovým zásobníkem, překlad uživatelských definic, vybrané řídicí struktury a optimalizační průchody nad IR.

V adresáři `tests/` jsou testy rozděleny do tematických souborů: `lexer_tests.rs`, `parser_tests.rs`, `analyzer_tests.rs`, `ir_tests.rs`, `codegen_tests.rs` a `integration_tests.rs`. Tyto testy společně ověřují správnost výstupů jednotlivých fází i to, že generování kódu odpovídá očekávané sémantice.

Při vývoji byly odhaleny a opraveny zejména chyby související s hranami tokenizace (pozice tokenů přes víceřádkový vstup, escapování řetězců) a se správným zacházením s kontextem REPL (registrace slov a proměnných mezi vstupy).

5.3 Porovnání s existujícími implementacemi

Cílem práce není nahradit vysoce optimalizované implementace jako GForth, ale ověřit, že navržená architektura (IR + optimalizační průchody + více backendů) umožňuje generovat efektivní cílový kód a zároveň zůstává rozšiřitelná.

Ve srovnání s tradičním Forth je
Roth koncipován více jako kompilátorový projekt: jazyková sada je záměrně omezena na konstrukce potřebné pro demonstraci frontendu, middle-endu i backendů. Výhodou je možnost experimentovat s optimalizacemi na explicitním mezikódu a s alternativními cílovými jazyky.

Tabulka 3: Instrukční sada mezikódu Roth

Instrukce	Popis	Consumes	Produces
Zásobníkové operace			
`Push(v)`	Vloží hodnotu na zásobník	0	1
`Pop`	Odebere hodnotu ze zásobníku	1	0
`Dup`	Duplikuje vrchol zásobníku	1	2
`Drop`	Zahodí vrchol zásobníku	1	0
`Swap`	Prohodí dva vrchní prvky	2	2
`Over`	Zkopíruje druhý prvek na vrchol	2	3
`Rot`	Rotuje tři vrchní prvky	3	3
Aritmetické operace			
`Add`	Sečte dva prvky	2	1
`Sub`	Odečte prvky	2	1
`Mul`	Vynásobí prvky	2	1
`Div`	Celočíselně vydělí	2	1
`Mod`	Zbytek po dělení	2	1
`Neg`	Negace čísla	1	1
Porovnávací operace			
`Equal`	Test rovnosti	2	1
`NotEqual`	Test nerovnosti	2	1
`Less`	Menší než	2	1
`Greater`	Větší než	2	1
`LessEqual`	Menší nebo rovno	2	1
`GreaterEqual`	Větší nebo rovno	2	1
Logické operace			
`And`	Logický součin	2	1
`Or`	Logický součet	2	1
`Not`	Logická negace	1	1
Řídicí tok			
`Jump(l)`	Nepodmíněný skok	0	0
`JumpIf(l)`	Skok pokud true	1	0
`JumpIfNot(l)`	Skok pokud false	1	0
`Call(f)`	Volání funkce	*	*
`Return`	Návrat z funkce	0	0
`Label(l)`	Návěstí	0	0
Smyčky			
`DoLoop(s,e)`	Začátek DO smyčky	2	0
`Loop(l)`	Konec LOOP (inkrement, test)	0	0
`PushLoopIndex`	Vloží index smyčky (I)	0	1
Vstup/výstup			
`Print`	Vypíše číslo	1	0
`PrintChar`	Vypíše znak	1	0
`PrintStack`	Vypíše celý zásobník	0	0
`ReadChar`	Načte znak	0	1
Paměťové operace			
`Load(a)`	Načte z paměti	0	1
`Store(a)`	Uloží do paměti	1	0

Závěr

Tato práce představila návrh a implementaci kompilátoru zásobníkového jazyka inspirovaného jazykem Forth. Výsledkem je nástroj
Roth implementovaný v jazyce Rust, který zahrnuje kompletní překladový řetězec od lexikální a syntaktické analýzy přes sémantickou kontrolu až po generování mezikódu, optimalizaci a produkci cílového kódu.

Byl navržen a realizován zásobníkový mezikód, nad kterým běží sada optimalizačních průchodů (constant folding, inlining, peephole optimalizace, strength reduction a eliminace mrtvého kódu). Implementace podporuje generování kódu do jazyků Rust a C a obsahuje interaktivní REPL prostředí využívající JIT-like kompilaci přes dynamicky načítané knihovny.

Možná budoucí rozšíření zahrnují zlepšení backendů (kompletní pokrytí všech IR instrukcí, lepší práce se skoky a smyčkami), rozšíření standardní knihovny a důkladnější měření výkonu na sadě referenčních programů.

Conclusions

This thesis presented the design and implementation of a stack-based language inspired by Forth. The resulting tool, Roth, is implemented in Rust and provides a full compilation pipeline: lexing, parsing, semantic validation, IR lowering, IR-level optimizations, and target code generation.

A dedicated stack-based intermediate representation was designed and used as the foundation for optimization passes such as constant folding, inlining, peephole optimization, strength reduction, and dead code elimination. The implementation supports generating Rust and C code and includes an interactive REPL based on a JIT-like workflow using dynamically compiled and loaded shared libraries.

Future work includes improving backend completeness, extending the standard library, and providing more extensive performance evaluation on a benchmark suite.

A Obsah elektronických dat

thesis/
: Zdrojové soubory práce v LaTeXu a výsledné PDF (`kidiplom.pdf`).

src/
: Zdrojové kódy kompilátoru Roth v jazyce Rust (frontend, IR, optimalizace, codegen).

src/repl/
: Implementace REPL režimu s JIT-like kompilací.

roth-runtime/
: Knihovna poskytující běhový kontext (zásobník, paměť) pro REPL.

std/
: Standardní knihovna jazyka ve formě Roth zdrojových souborů.

tests/
: Jednotkové a integrační testy kompilátoru.

test_source/
: Ukázkové vstupy pro integrační testy (včetně očekávaných výstupů).

README.md
: Stručný popis projektu.

Cargo.toml
: Konfigurační soubor projektu pro Rust.

.build/
: Výstupní adresář generovaného kódu (vytváří se při běhu kompilátoru).

B Uživatelská dokumentace

Tato příloha popisuje použití kompilátoru Roth z příkazové řádky a základní práci v režimu REPL.

B.1 Požadavky

Pro sestavení projektu je potřeba nainstalovaný Rust toolchain. Pro spuštění některých režimů jsou dále potřeba:

- `rustc` v `PATH` (používá se při `-run` a v režimu REPL),
- `gcc` (používá se při `-run` s backendem pro C).

B.2 Sestavení a testy

Projekt lze sestavit a otestovat standardními příkazy:

Zdrojový kód 12: Sestavení a testy

```
cargo build
cargo test
```

B.3 Kompilace souboru

Kompilace souboru probíhá spuštěním binárky `roth` se jménem vstupního souboru. Výstupní kód je uložen do adresáře `.build/`.

Zdrojový kód 13: Kompilace souboru

```
cargo run -- path/to/program.fs
```

Volba backendu:

Zdrojový kód 14: Volba backendu

```
cargo run -- path/to/program.fs --backend rust-ir
cargo run -- path/to/program.fs --backend c-ir
```

Volitelně lze nastavit jméno výstupního souboru (stále bude umístěn do `.build/`):

Zdrojový kód 15: Vlastní název výstupu

```
cargo run -- path/to/program.fs -o out.rs
```

B.3.0.1 Režim `-run`. Při použití `-run` kompilátor kromě vygenerování kódu automaticky spustí překlad cílového kódu a výsledný program vykoná. Pro Rust backend se používá `rustc -O`, pro C backend `gcc -O2`.

Zdrojový kód 16: Kompilace a spuštění

```
cargo run -- path/to/program.fs --backend rust-ir --run
cargo run -- path/to/program.fs --backend c-ir --run
```

B.4 Ladění výpisů

Volba `-debug` řídí množství ladicích informací:

- `0` — bez výpisů,
- `1` — vypíše vygenerovaný cílový kód,
- `2` — vypíše tokeny, AST, IR a statistiky optimalizací,
- `3` — navíc vypíše barevně zvýrazněný cílový kód (pokud je dostupný highlighter).

Volba `-no-color` vypne barevné zvýraznění.

B.5 REPL režim

REPL se spustí bez argumentu vstupního souboru, nebo pomocí `-i`/`-interactive`:

Zdrojový kód 17: Spuštění REPL

```
cargo run
cargo run -- -i
```

V REPL lze psát běžné Roth příkazy (např. `2 3 + .`) a definovat slova pomocí `: NAME ... ;`. REPL navíc obsahuje příkazy začínající dvojtečkou bez mezery (např. `:help`).

Zdrojový kód 18: Základní REPL příkazy

```
:help       (napoveda)
:stack      (vypis zasobniku)
:words      (seznam uzivatelskych slov)
:vars       (seznam promennych)
:clear      (vypradneni zasobniku)
:reset      (reset stavu)
:debug 2    (nastaveni debug urovne)
:quit       (ukonceni)
```

Konstrukce `INCLUDE` je podporována pouze při kompilaci ze souboru (je implementována jako preprocesor v `src/main.rs`); v režimu REPL se nevyhodnocuje.

C Gramatika jazyka

Tato sekce popisuje implementovanou syntaxi jazyka Roth ve zjednodušené EBNF. Jazyk je tokenizován na základě bílých znaků a několika speciálních znaků (`:`, `;`, `(`, `)`, `"`).

C.1 Lexikální pravidla

Whitespace mezery, tabulátory a konce řádků oddělují tokeny.

Comment `(` *libovolné znaky bez* `)` . Komentáře jsou ignorovány parserem.

Number dekadický literál `i32`. Pokud převod selže (např. přetečení), token je brán jako slovo.

Word posloupnost znaků do prvního bílého znaku, `(`, `)`, `"`. Slova jsou case-insensitive (lexer je normalizuje na velká písmena).

String buď `"..."` (s podporou escape sekvencí
`n`,
`t`,

```
r,
",
```

), nebo `S"..."` (bez escape sekvencí; volitelná mezera po `S"`).

C.2 Syntaktická pravidla (EBNF)

Zdrojový kód 19: EBNF gramatika

```
program         ::= { statement } ;

statement       ::= number
                  | string
                  | word
                  | definition
                  | variable_decl
                  | comment ;

definition      ::= ':' word { stmt_in_def } ';' ;
stmt_in_def     ::= number | string | word | variable_decl | comment ;
                  // vnorene definice nejsou dovoleny

variable_decl   ::= 'VARIABLE' word ;

number          ::= [ '-' ] digit { digit } ;

string          ::= '"' { char | escape } '"'
                  | 'S"' [ ' ' ] { char } '"' ;

comment         ::= '(' { char } ')' ;

word            ::= token ;
```

Při kompilaci ze souboru je před lexikální analýzou proveden preprocesing direktivy `INCLUDE`, která rekurzivně vkládá obsah souborů do zdrojového textu. Název souboru může být uveden v uvozovkách nebo bez nich a cesta je vyhodnocena relativně k aktuálnímu souboru (a případně k pracovnímu adresáři).