VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

IMPLEMENTACE INTERPRETU JAZYKA PRO MATEMATICKÉ VÝPOČTY

IMPLEMENTATION OF A LANGUAGE INTERPRETER FOR MATHEMATICAL COMPUTATIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MARTIN KOBELKA
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR VEIGEND
SUPERVISOR

BRNO 2019

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

22065

Student: **Kobelka Martin**
Program: Informační technologie
Název: **Implementace interpretu jazyka pro matematické výpočty**
Implementation of a Language Interpreter for Mathematical Calculations
Kategorie: Modelování a simulace
Zadání:

1. Seznamte se s nástroji pro profesionální matematické výpočty (např. MATLAB, MAPLE) a identifikujte jejich slabiny z pohledu uživatelské přívětivosti.
2. Prostudujte výpočetní jazyky, které tyto nástroje používají. Navrhněte možná zlepšení (např. efektivní práci s rozšířenou aritmetikou).
3. Navrhněte nový výpočetní jazyk.
4. Implementujte interpret navrženého jazyka. Dále pro tento interpret vytvořte jednoduché uživatelské rozhraní. Rozhraní by mělo být uživatelsky přívětivé a umožňovat snadnou vizualizaci výsledků a ladění výpočtu.
5. Zaměřte se také na rozšiřitelnost vytvořeného interpretu/uživatelského rozhraní. Implementujte vzorovou knihovnu pro numerický výpočet diferenciálních rovnic pomocí metody vyššího řádu. Knihovna bude podporovat minimálně dvě numerické metody pro řešení diferenciálních rovnic.
6. Srovnejte efektivitu současných nástrojů a vašeho řešení. Zaměřte se také na použitelnost a přívětivost pro uživatele.

Literatura:

- Kunovský, J.: Modern Taylor series method. Habilitation work, VUT Brno, 1994.
- Press, H., et al. "Numerical Recipes in C++. The Art of Computer Programming." (2002).
- M. Kubíček, M. Dubcová, D. Janovská: Numerické metody a algoritmy. VŠCHT Praha, 2005. ISBN 80-7080-558-7
- Další dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Veigend Petr, Ing.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Cílem práce je navrhnout nový programovací jazyk, který umožňuje efektivní zápis matematických výpočtů, implementovat demonstrační interpret zpracovávající vhodně zvolenou podmnožinu tohoto jazyka a navrhnout pro něj grafické uživatelské rozhraní, které zápis výpočtu usnadňuje a umožňuje efektivní a přehlednou vizualizaci výsledků výpočtu a jeho základní ladění. V práci je výsledný jazyk rozebrán a jsou s ním prováděny experimenty za pomocí vytvořeného interpretu. Jsou také popsány rozdíly mezi navrženým řešením a řešením, které nám poskytují konkurenční platformy.

Abstract

The main goal of this bachelor thesis is to design and implement the new programing language, which can be used for mathematical computations, implement the demonstration interpret of this language and design a graphical user interface for it. The user interface makes it easy to write the calculation, enables effective and clear visualization of calculation results and basic debugging of calculation. The properties of the resulting language are described in the thesis with the several experiments with the interpret, which implements a subset of the language. Differences between designed solution and other platforms are also described in the thesis.

Klíčová slova

interpret, programovací jazyk, Java, JavaFX, vývojové prostředí, vizualizace, ANTLR4, diferenciální rovnice, Eulerova metoda, Runge kutta

Keywords

interpret, programing language, Java, JavaFX, development environment, visualization, ANTLR4, differential equation, Euler method, Runge kutta

Citace

KOBELKA, Martin. *Implementace interpretu jazyka pro matematické výpočty.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Petr Veigend

Implementace interpretu jazyka pro matematické výpočty

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Petra Veigenda. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Martin Kobelka
15. května 2019

Poděkování

Chtěl bych poděkovat Ing. Petru Veigandovi, vedoucímu mé bakalářské práce, za vedení, odborné konzultace v oblasti matematiky a připomínky k mé práci, které mě vždy orientovaly správným směrem. Dále bych chtěl poděkovat paní Mgr. Jarmile Kalábkové za gramatickou korekturu práce.

Obsah

Kapitola 1

Úvod

Teorie formálních jazyků přinesla do oblasti informatiky obrovský rozmach v návrhu vyšších programovacích jazyků. Architektura překladače či interpretu každého programovacího jazyka je dnes na této teorii přímo vystavěna. Teorie formálních jazyků přinesla především jednoznačnost syntaktických konstrukcí, oddělení jednotlivých fází překladu a možnost syntézy částí překladače ze snadno čitelného popisu zapsaného ve formě gramatik, regulárních výrazů či jiných formalismů. Díky tomu mohlo vzniknout rozsáhlé spektrum nových programovacích jazyků v mnoha různých oblastech použití.

Pointou této práce bylo studium jisté kategorie programovacích jazyků. Jedná se o jazyky zaměřené na zadávání a řešení matematických výpočtů. Rozumíme jimi především úzce specializované jazyky, které nalezly uplatnění především ve vědě a výuce. Uživateli poskytují velmi vysokou úroveň abstrakce, jsou nejčastěji interpretované, což umožňuje použití vysokoúrovňových konstrukcí typu asociativní pole, netypovost nebo optimalizace typu vyhodnocování kódu až za běhu a mnoho dalších. Obsahují často velké množství podpůrných nástrojů a knihoven pro vizualizaci výsledků, paralelizaci výpočtu a vysokou modularitu pro tvorbu rozšíření do dalších oblastí vědy a techniky. Tyto jazyky přináší studentům a vědcům značnou úsporu času a zlepšení orientace při práci s výpočty, které mohou být zapsány mnohem stručnějším, čitelnějším a elegantnějším způsobem.

Cílem této práce je navrhnout nový programovací jazyk pro zápis a simulaci matematických výpočtů, implementovat interpret, který vhodně zvolenou podmnožinu tohoto jazyka zpracovává, a pro tento interpret navrhnout a implementovat jednoduché demonstrační grafické uživatelské rozhraní, které umožní pohodlné zadávání vstupu, demonstraci výsledků a základní ladění výpočtu.

Při vývoji bylo třeba se zaměřit především na přístupnost pro uživatele a jednoduché, rychlé a efektivní použití. Architektura aplikace je založena na objektově orientovaných paradigmatech a implementace je provedena v programovacím jazyce *Java* za použití knihovny *ANTLR*, která umožňuje syntetizovat některé části interpretu ze snadno čitelného formálního popisu, což velmi usnadňuje a urychluje vývoj interpretu. V rámci interpretu byl také implementován modul umožňující numerické řešení diferenciální rovnice vyššího řádu s lineárními koeficienty. Zdrojový kód je vhodně komentován tak, aby nezávislý čtenář po vhodném úvodu do problematiky byl schopen provádět úpravy libovolného rozsahu. Demonstrační grafické uživatelské rozhraní je vytvořeno za použití moderního a flexibilního frameworku *JavaFX*.

V kapitole 2 je připomenuta teorie formálních jazyků. Dále jsou zde popsány jednotlivé fáze překladu a interpretace tak, jak následují logicky za sebou při jejich vykonávání. Velký důraz je kladen na vybrané partie, které jsou nutné pro pochopení zdrojového kódu

demonstračního interpretu, který je realizačním výstupem této práce, zejména pak *LL gramatiky* 2.1. V první části kapitoly jsou definovány základní pojmy. Na těchto pojmech jsou vystavěny modely a algoritmy, které používá nástroj *ANTLR* pro generování lexikálního a syntaktického analyzátoru.

V kapitole 3 jsou analyzovány nástroje pro řešení matematických výpočtů. Jsou také stručně rozebrány výpočetní jazyky, které nám tato řešení poskytují. Následně je v kapitole 4 diskutován návrh vytvořeného jazyka. Kapitola je pojata jako kompletní referenční příručka k vytvořenému jazyku. V kapitole 5 je z několika hledisek srovnáno navržené řešení s řešením, které poskytují konkurenční platformy.

Kapitola 6 přibližuje použité technologie, na kterých byla demonstrační aplikace zahrnující interpret a grafické uživatelské rozhraní vystavěna. Ukazuje výhody použití těchto technologií a srovnává je s technologiemi konkurenčními, které nebyly použity. Dále popisuje implementaci demonstrační aplikace. Je podrobně rozebrána její výstavba a kroky, které byly při vývoji zvoleny. Některé fáze překladu musely být oproti jejich formálním zápisům mírně poupraveny. Důvodem byla především vyšší efektivita a rychlost implementace. Popis těchto modifikací je také součástí této kapitoly.

V poslední kapitole 7 jsou shrnuty dosažené výsledky této práce a ukázány možnosti budoucího vývoje.

Kapitola 2

Teoretický úvod do formálních jazyků

Implementace *překladače* či *interpretu* programovacího jazyka je postavena na hlubokých teoretických základech vycházejících z diskrétní matematiky[1] a teorie formálních jazyků. Pochopení těchto disciplín je nezbytné pro plné porozumění architektuře a zdrojovému kódu libovolného překladače či interpretu.

V následující kapitole jsou definovány vybrané pojmy, které je nutné před popisem jazyka a implementace jeho interpretu připomenout. Veškeré pojmy z této kapitoly vycházejí z knihy [11] a studijní opory k předmětu Formální jazyky a překladače [12]. Doplňující obrázky byly inspirovány prezentacemi z předmětu Formální jazyky a překladače [12].

2.1 Vymezení základních pojmů

V této sekci budou nejprve definovány elementární pojmy formou jejich přesných matematických definic. Definice čtenáře postupně dovedou až k formálním modelům, které jsou základem nástroje *ANTLR*, na kterém je vystavěna fáze lexikální a syntaktické analýzy použité při implementaci demonstračního interpretu.

V následující sekci 2.2 se na tyto pojmy plynule naváže vysvětlením principu fungování překladače. Princip bude vysvětlen rozdělením překladu na fáze, které na sebe logicky navazují. Dále budou definovány klíčové rozdíly mezi překladem a interpretací.

Základní pojmy z teorie formálních jazyků

Definice 1 *Abeceda je konečná, neprázdná množina elementů, které nazýváme symboly.*

Abecedu [11] zpravidla označujeme symbolem Σ.

Definice 2 *Nechť Σ je abeceda. Potom*

1. *ε[2] je prázdný řetězec nad abecedou Σ,*
2. *pokud x je řetězec nad abecedou Σ a $y \in \Sigma$, potom o xy mluvíme také jako o řetězci nad abecedou Σ[3] [12].*

[1] Detailní popis základních pojmů z oblasti diskrétní matematiky je mimo rozsah práce. Zájemce odkazuji na vhodnou literaturu [6].

[2] $\mathcal{E}$ je dnem této rekurzivně zadané definice. Jeho délka je stanovena na hodnotu 0.

[3] Délka řetězce xy je poté stanovena jako délka řetězce $y + 1$.

Definice 3 *Nechť za Σ^* označme množinu všech řetězců nad abecedou Σ. Potom každá množina L taková, že $L \subseteq \Sigma^*$ je jazyk nad abecedou Σ [12].*

Každý formální jazyk lze popsat gramatikou. Gramatika je matematický model, který je definován následovně:

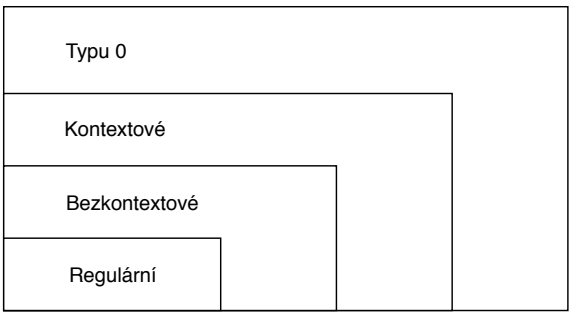
Definice 4 *Gramatika [12] je čtveřice $G = (N, T, P, S)$, kde*

- *N je abeceda neterminálních symbolů (neterminálů),*
- *T je abeceda terminálních symbolů (terminálů), přičemž platí, že $N \cap T = \emptyset$,*
- *P je konečná množina přepisovacích pravidel tvaru $A \to x$, kde $A \in (\Sigma \cup N)^* N (\Sigma \cup N)^*$, $x \in (N \cup T)^*$,*
- *$S \in N$ je počáteční neterminál.*

Gramatiky lze podle vyjadřovacích schopností[4] rozdělit do *tříd formálních gramatik*, do *Chomského hierarchie* zavedené Noamem Chomským v roce 1956 [23]. Ta je tvořena čtyřmi třídami. Každá z těchto tříd je definována pomocí modifikace definice obecné gramatiky, od níž se liší pouze tvarem přepisovacího pravidla z množiny P. Třídy Chomského hierarchie jsou následující:

- gramatiky typu 0 - Frázové nebo též Rekurzivně spočetné gramatiky,
- gramatiky typu 1 - Kontextové gramatiky,
- gramatiky typu 2 - Bezkontextové gramatiky,
- gramatiky typu 3 - Regulární nebo též Levolineární gramatiky.

Pro třídy Chomského hierarchie platí, že každý *regulární jazyk* je také *bezkontextový*, každý *bezkontextový jazyk* je také *kontextový* a každý *kontextový jazyk* také *rekurzivně spočetný*. Uvedené gramatiky jsou tedy vzájemně ve vztahu ostré inkluze. Znázornění této inkluze je demonstrováno na obrázku 2.1 [23].



Obrázek 2.1: Znázornění ostré inkluze v Chomského hierarchii.

Lze dokázat, že existují dokonce i takové formální jazyky, které nelze popsat ani gramatikou typu 0. Tyto jazyky označujeme jako jazyky mimo třídu 0 [30].

[4] Vyjadřovací schopností třídy gramatik rozumíme schopnost popsat určitou skupinu jazyků. V případě, že za pomocí gramatik z jedné třídy lze generovat širší skupinu formálních jazyků než za pomocí třídy druhé, říkáme o ní, že její vyjadřovací schopnost je vyšší než vyjadřovací schopnost třídy druhé.

Regulární jazyky

Třídou *regulárních jazyků* označujeme dle *Chomského hierarchie* třídu jazyků s nejnižší vyjadřovací schopností. Třídu jazyků často v teorii formálních jazyků definujeme pomocí více různých modelů[5]. Regulární jazyky lze definovat pomocí tří formálních modelů, jimiž jsou *konečný automat*, *regulární výraz* a *levolineární gramatika*. Tyto modely lze mezi sebou libovolně převádět.

Definice 5 *Konečný automat [11] je pětice $M = (Q, \Sigma, R, s, F)$, kde*

1. *Q je konečná množina stavů,*
2. *Σ je vstupní abeceda,*
3. *R je konečná množina pravidel tvaru: $pa \rightarrow q$, kde $p, q \in Q, a \in (\Sigma \cup \varepsilon)$,*
4. *$s \in Q$ je počáteční stav,*
5. *$F \subseteq Q$ je množina koncových stavů.*

Uvedená definice popisuje *nedeterministrický konečný automat*. Ten lze algoritmicky převést na *deterministrický konečný automat*[6], jehož implementace je pro nás daleko snazší a výhodnější. Druhým modelem je *Regulární výraz*.

Definice 6 *Nechť Σ je abeceda. Regulární výrazy nad abecedou Σ a jazyky, které značí jsou definovány následovně [12]:*

1. *$\emptyset$ je regulární výraz značící jazyk $L = \emptyset$,*
2. *ε je regulární výraz značící jazyk $L = \{\varepsilon\}$,*
3. *a, kde $a \in \Sigma$ je regulární výraz značící jazyk $L = \{a\}$.*
4. *Nechť r a s jsou regulární výrazy značící po řade jazyky L_r a L_s, potom:*
   - *$(r.s)$ je regulární výraz značící jazyk $L = L_r.L_s$,*
   - *$(r + s)$ je regulární výraz značící jazyk $L = L_r \cup L_s$,*
   - *(r^*) je regulární výraz značící jazyk ${L_r}^*$.*

Regulární výraz lze algoritmicky převést na *konečný automat* i *regulární gramatiku*. Toho se využívá při tvorbě *lexikálního analyzátoru* (sekce 2.3). *Regulární gramatika* vzniká modifikací definice *obecné gramatiky*. Na tvar přepisovacího pravidla klade nejstriktnější požadavky.

[5] Jsou jimi zpravidla automaty a gramatiky. Každý z těchto modelů má pro nás jiný význam. Gramatiku používáme k tomu, abychom vygenerovali věty jazyka, zatímco automat používáme k tomu, abychom rozpoznali, zda věta náleží danému jazyku.

[6] Slovem deterministrický rozumíme jednoznačný z hlediska výběru pravidla pro následující přechod. M je deterministický konečný automat, pokud pro každé pravidlo $(pa \rightarrow q) \in R$ platí, že množina $R - \{pa \rightarrow q\}$ neobsahuje žádné pravidlo s levou stranou pa, a zároveň $q \neq \varepsilon$ [12].

Definice 7 *Nechť G je gramatika a nechť $p = (A \rightarrow x)$ je přepisovací pravidlo. Pokud všechna pravidla mají tvar $(A \rightarrow \alpha) \vee (A \rightarrow \alpha B)$, kde $\alpha \in \Sigma$ a $A, B \in N$, potom tuto gramatiku označujeme jako pravolineární, regulární, nebo případně gramatiku typu 3. k pravolineární gramatice existuje ještě výpočetně ekvivalentní gramatika, kterou nazýváme levolineární*[7] *[11].*

Definice 8 *Nechť L je jazyk. Jazyk L nazýváme regulární, pokud existuje regulární výraz, který jej definuje, konečný automat, který jej přijímá, a regulární gramatika, která jej generuje [12].*

Bezkontextové jazyky

Třída *bezkontextových jazyků* je z hlediska *Chomského hierarchie* nadtřídou třídy *regulárních jazyků* a podtřídou třídy *kontextových jazyků*. Formální modely definující *bezkontextové jazyky* jsou *Zásobníkový automat* a *bezkontextová gramatika*.

Definice 9 *Zásobníkový automat [12] je sedmice $M = (Q, \Sigma, \delta, R, s, S, F)$, kde:*

- *Q je konečná množina stavů,*
- *Σ je vstupní abeceda,*
- *δ je zásobníková abeceda,*
- *R je konečná množina pravidel tvaru $Apa \rightarrow wq$, kde $A \in \delta$, $p, q \in Q$, $a \in (\Sigma \cup \varepsilon)$, $w \in \delta*$,*
- *s je počáteční stav,*
- *$S \in \delta$ je počáteční symbol na zásobníku,*
- *$F \subseteq Q$ je množina koncových stavů.*

Definice *bezkontextové gramatiky* je podobná definici *obecné gramatiky*. Přidává však omezující podmínky na přepisovací pravidlo. Omezující podmínky, která jsou na toto pravidlo kladena jsou méně striktní než u *regulární gramatiky*. Díky tomu jsme schopni přijímat nejen opakující se struktury, ale i struktury, které se do sebe rekurzivně zanořují. Jelikož jde opět o ekvivalentní modely, lze zásobníkový automat a bezkontextovou gramatiku vzájemně převádět [12].

Definice 10 *Nechť G je gramatika a nechť $p = (A \rightarrow x)$ je přepisovací pravidlo z této gramatiky. Pokud pro všechna přepisovací pravidla platí, že $A \in N$ a $x \in (N \cup T)^*$, potom G označujeme jako bezkontextovou gramatiku [11].*

Definice 11 *Nechť L je jazyk. L je bezkontextový (BKJ), pokud existuje bezkontextová gramatika, která jej generuje, a zásobníkový automat, který jej přijímá [12].*

[7] Záměnou druhého tvaru přepisovacího pravidla na $A \rightarrow B\alpha$ dostáváme levolineární gramatiku. Tato gramatika je výpočetně ekvivalentní s pravolineární gramatikou.

Backusova–Naurova forma

Formální způsob zápisu *bezkontextové gramatiky* jako čtveřice je sice možný, nicméně pro praktické použití je značně zdlouhavý a nepohodlný. Proto byly časem pro popis reálných programovacích a značkovacích jazyků vyvinuty zjednodušující zápisy, které umožňují zapsat gramatiku stručněji a přehledněji. Nejznámější jsou *Backusova–Naurova forma* a *Rozšířená Backusova–Naurova forma*. *Backusova-Nauerova forma* je sada odvozovacích pravidel tvaru:

```
<symbol> ::= <výraz se symboly>
```

kde `<symbol>` je neterminál a `<výraz se symboly>` sestává ze sekvence symbolů, nebo sekvencí oddělených svislou čárou `'|'`, která označuje alternativní možnost. `<výraz se symboly>` vpravo představuje možnou náhradu za `<symbol>` vlevo. Jedná se o původní návrh[8] použitý pro návrh jazyka *Algol*. *Rozšířená Backusova–Naurova forma* rozšiřuje *Backusovu–Naurovu formu* na jednoduchý jazyk s komentáři a možností znovupoužití některých částí za pomocí fragmentů [21]. Tento jazyk lze poté použít k návrhu jiných jazyků.

LL gramatiky

Zkonstruovat obecný syntaktický analyzátor *bezkontextové gramatiky* je sice možné [12], nicméně výsledek by byl velmi těžkopádný, složitě implementovatelný, pro účely syntaktické analýzy reálného programovacího jazyka nepraktický a z hlediska časové či prostorové složitosti velmi neefektivní. Proto pro praktické použití vyjadřovací schopnost bezkontextových jazyků úmyslně snižujeme, abychom mohli zkonstruovat jednodušší a z hlediska časové a prostorové složitosti také efektivní analyzátor. Jedním z nejužívanějších modelů, který se v rámci výstavby překladačů používá, je *LL gramatika*:

Definice 12 *Nechť $G = (N, T, P, S)$ je bezkontextová gramatika 10. G je $LL(k)$ gramatika [24], pokud*

1. *pro každý řetězec $w \in \Sigma^*$ délky nejvýše K,*
2. *pro každý neterminální symbol $A \in V$,*
3. *pro každý řetězec symbolů terminálu $w_1 \in \Sigma^*$*

existuje nejvýše jedno přepisovací pravidlo $r \in R$ takové, že u některých řetězců symbolů terminálu $w_2, w_3 \in \Sigma^*$ platí, že

1. *řetězec w_1Aw_3 lze odvodit ze startovacího symbolu S,*
2. *w_2 může být odvozen z A po prvním použití pravidla r,*
3. *první k symboly w a w_2w_3 musí souhlasit.*

Neformálně řečeno nahlížíme na k následujících symbolů proto, abychom odvodili w_1Aw_3 s A jako nejbližším neterminálem. *LL gramatiky* jsou velmi důležitým pojmem z teorie formálních jazyků jazyků. Jejich vyjadřovací schopnost je nižší než u *bezkontextových gramatik*, ale vyšší než u *regulárních gramatik*. Tvoří tedy podtřídu *bezkontextových jazyků* a nadtřídu *regulárních jazyků*. V případě, že jsme ochotni toto omezení akceptovat, dostáváme model, který lze velmi snadno implementovat.

[8] Existuje řada variant a rozšíření Backusovy–Naurovy formy, které vznikly z důvodu dosažení vyšší jednoduchosti nebo stručnosti.

Q gramatiky

Q gramatika rozšiřuje LL gramatiku o ε pravidla. ε pravidlo je definováno následovně:

Definice 13 *Nechť $p \in P$ je pravidlo. Pokud má p tvar $A \rightarrow x$, kde $A \in N$ a $x = \varepsilon$, potom p označujeme jako ε pravidlo.*

LL syntaktický analyzátor

Program rozpoznávající jazyk typu LL se nazývá *LL syntaktický analyzátor* (sekce 2.3). Pracuje shora dolů, což znamená, že se snaží z počátečního pravidla a vstupní věty rozhodnout, je-li vstupní věta větou jazyka. Dále vrací simulaci konstrukce derivačního stromu.

Velmi často se pro rozpoznání programovacího jazyka používají $LL(1)$ gramatiky, tedy takové $LL(k)$ gramatiky, kde $k = 1$[9]. $LL(1)$ analyzátor lze navíc v případech, kdy nám $LL(1)$ gramatika k popisu syntaktické konstrukce nestačí, snadno modifikovat na $LL(2)$, případně i $LL(k)$ analyzátor. Jedná se o časem velmi dobře prověřený model, na kterém jsou díky své vysoké jednoduchosti a dobré časové i prostorové složitosti vystavěny i profesionální překladače.

2.2 Překladač

V následující sekci jsou definovány pojmy *překladač* a *interpret*. Je vysvětleno, jak spolu souvisí a jaké jsou mezi nimi rozdíly.

Překladač

Překladač je program, který na vstupu čte zdrojový kód zapsaný ve zdrojovém jazyce a na výstup poskytuje cílový kód zapsaný v cílovém jazyce. Překlad probíhá zpravidla z jazyka vyšší úrovně abstrakce do jazyka nižší úrovně abstrakce. Můžeme však najít i takové překladače, pro které toto neplatí[10]. Extrémním příkladem jsou *reverzní překladače*[11]. Nejčastějším výstupním jazykem je jazyk symbolických instrukcí neboli *assembler*. Jazyk symbolických instrukcí je přímo spustitelný na cílové architektuře. Výstupem překladu však může být kód v libovolném jazyce, jehož vyjadřovací schopnost je stejná nebo vyšší než vyjadřovací schopnost zdrojového jazyka.

Programovací jazyk, který překládáme do jazyka symbolických instrukcí, nazýváme kompilovaný. Mezi kompilované jazyky můžeme zařadit například jazyky *C*, *C++*, *Pascal* nebo *Go*. Kompilované jazyky používáme zpravidla tam, kde vyžadujeme maximální výkon.

Interpret

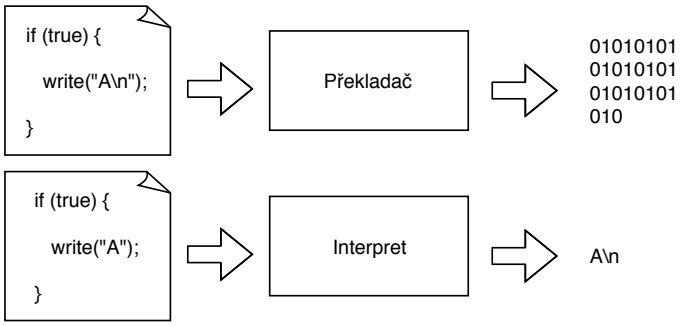
Interpret je program, který čte zdrojový program zapsaný ve zdrojovém jazyce, a v reálném čase jej přímo vykonává. Program není třeba pro jeho vykonání převádět do strojového

[9] Absolutní většinu běžně používaných jazykových konstrukcí lze transformovat na $LL(1)$ gramatiky, jež lze velmi snadno programově rozpoznávat. U ostatních si lze dopomoci modifikací analyzátoru na $LL(2)$, případně $LL(k)$.

[10] Překladač, který překládá mezi dvěma jazyky poskytující stejnou úroveň abstrakce se nazývá *transpiler*.

[11] Reverzní překladač neboli dekompilátor je počítačový program, který na vstupu přijímá výstup z překladu, a snaží se z něj rekonstruovat zdrojový kód původní aplikace.

kódu. Úvodní fáze interpretace jsou shodné s úvodními fázemi překladu (sekce 2.2). Hlavní rozdíly mezi interpretem a překladačem můžeme najít v posledních fázích. V případě interpretace není na výstup poskytnut výstupní kód, ale vnitřní kód je převeden do běhové reprezentace a ta je přímo vykonána. Vstupem interpretu může být i mezikód[12] [16]. Typickými zástupci interpretovaných programovacích jazyků jsou zpravidla jazyky vysoké úrovně abstrakce založené na velmi abstraktních programovacích paradigmatech. Jedná se především o skriptovací jazyky jako *Python* nebo *PHP*, případně funkcionální jazyky typu *Haskel*. Srovnání interpretu a překladače je demonstrováno na obrázku 2.2.



Obrázek 2.2: Srovnání výstupu interpetu a kompilátoru.

2.3 Struktura překladače

Překlad zdrojového kódu ze zdrojového jazyka probíhá v několika fázích, které lze od sebe velmi dobře oddělit. Tyto fáze na sebe logicky navazují. Identifikaci a definici fází překladu nám přinesla teorie formálních jazyků (sekce 2.1).

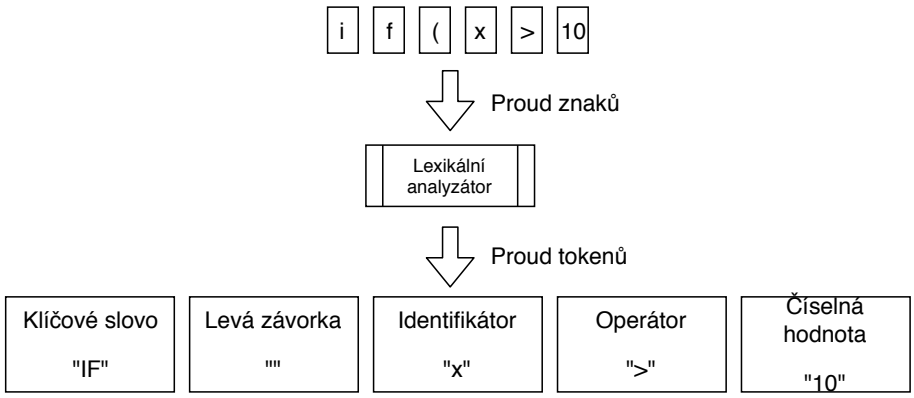
Lexikální analýza

První fází při překladu či interpretaci zdrojového kódu je fáze *lexikální analýzy*, která je vykonávána *lexikálním analyzátorem*. *Lexikální analyzátor* na vstupu přijímá zdrojový kód zapsaný ve zdrojovém jazyce. Tento kód čte po jednotlivých znacích a následně jej zpracovává. Pro zpracování zdrojového kódu na vstupu používá modely *regulárních jazyků*. Zdrojový kód tak rozděluje na lexikálně oddělitelné jednotky, *lexémy* [12]. *Lexémy* jsou poté programově reprezentovány jako *tokeny*. Ukázka převodu vstupu na proud tokenů je na obrázku 2.3.

Definice 14 Token *je struktura složená nejčastěji ze dvou částí [11]. Těmito částmi jsou:*

- *název tokenu,*
- *hodnota atributu.*

[12] Z důvodu rychlosti jsou některé jazyky kompilovány do tzv. *mezikódu*. Jedná se o kompromis mezi kompilací a interpretací. Zdrojový kód je zkompilován do velmi optimalizovaného a stručného kódu, mezikódu. Mezikód je jazyk, který lze velmi jednoduše a rychle interpretovat. Úvodní fáze interpretace mezikódu, tedy načtení a zpracování zdrojového kódu může proběhnout násobně rychleji než u zdrojového jazyka. Jazyky využívající mezikód jsou například *Java*, *C#* nebo *Visual basic*.



Obrázek 2.3: Znázornění činnosti lexikálního analyzátoru.

Tato definice však není téměř nikdy striktně dodržována. *Token* může nést jakoukoliv hodnotu, která pro nás může mít v pozdějších fázích nějaký význam. Například pozici prvního a posledního symbolu lexémy z důvodu ohlášení lexikální chyby, nebo délku lexémy. V případě, že nastane chyba při rozpoznávání lexémy, lexikální analyzátor ohlásí vznik lexikální chyby a překlad je již v této fázi ukončen. Kromě této základní funkce lexikální analyzátor obvykle provádí také vynechávání bílých znaků[13] a komentářů.

Formálními modely lexikálního analyzátoru jsou *regulární výraz*, *konečný automat* a *levolineární gramatika*. Jednotlivé lexémy jsou popsány za pomocí *regulárních výrazů*. *Regulární výrazy* jsou převedeny do jednoho velmi rozsáhlého *deterministrického konečného automatu* (definice 5), který se z počátečního stavu snaží čtením vstupu přejít do koncového stavu a rozpoznat tak nejdelší možnou lexému. Po jejím rozpoznání se vrací zpět do startujícího stavu. V případě neočekávaného vstupního symbolu tento automat také identifikuje vznik lexikální chyby.

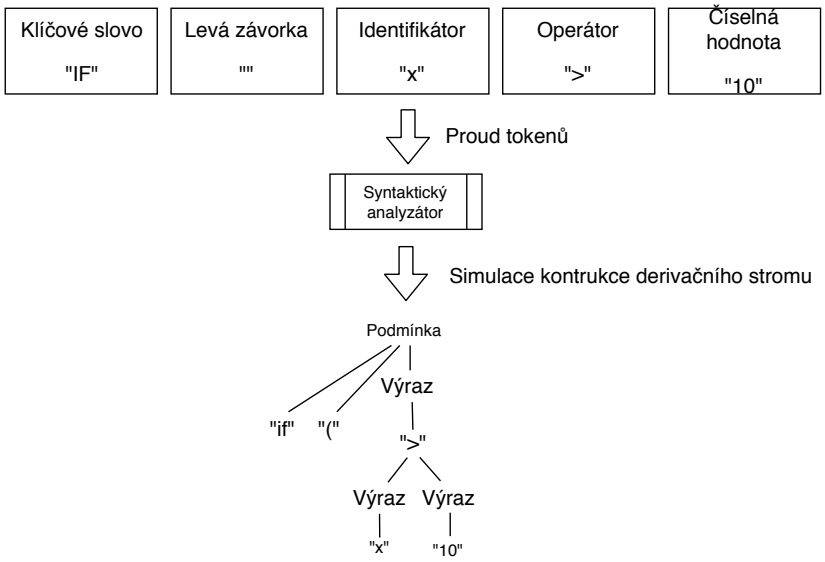
Tvorba lexikálního analyzátoru je vysoce rutinní záležitost. Lze jej velmi jednoduše napsat ručně jako *deterministrický konečný automat*, který typ lexémy rozeznává koncovým stavem [12], nebo jej lze generovat automaticky za pomocí nástrojů k tomu určeným (sekce 6.3). Generování probíhá nejčastěji zápisem jednotlivých lexém pomocí regulárních výrazů. Z těchto regulárních výrazů je poté syntetizátorem generován výsledný konečný automat. Výsledný kód má často podobu černé skříňky, ke které následně přistupujeme za pomocí rozhraní [12].

Syntaktická analýza

Syntaktický analyzátor je jádrem celého překladače. Na vstup přijímá proud tokenů dodaných z *lexikálního analyzátoru* a na výstup poskytuje simulaci konstrukce derivačního stromu[14]. Kontroluje, zda vstupní posloupnost tokenů tvoří syntakticky správně napsaný program. Pokud analýzou není nalezena příslušná simulace konstrukce derivačního stromu, je překlad ukončen se syntaktickou chybou. Znázornění činnosti syntaktického analyzátoru lze vidět na obrázku 2.4. Formálními modely syntaktického analyzátoru jsou bezkontextová gramatika a zásobníkový automat (definice 9).

[13] Bílý znak je znak představující prázdné místo. Typickým zástupcem bílého znaku je mezera nebo tabulátor. Bílé znaky se používají pro formátování kódu.

[14] Simulaci konstrukce derivačního stromu si lze představit jako strom pravidel, která byla použita k odvození věty jazyka. Strom ale nesestavujeme, pouze simulujeme jeho konstrukci [12].



Obrázek 2.4: Znázornění činnosti syntaktického analyzátoru.

Pro syntaktickou analýzu můžeme použít dva přístupy. Prvním přístupem je syntaktická analýza *shora dolů*, kdy analýzu začínáme od startujícího pravidla a snažíme se aplikací dalším možných pravidel dopracovat k posloupnosti tokenů, která tvoří větu jazyka. Druhou možností jak provést syntaktickou analýzu je metoda *zdola nahoru*. U této metody začínáme analýzu od vstupních tokenů a snažíme se nalézt přepisovací pravidla, která lze použít pro získání věty jazyka reprezentovanou vstupní posloupností tokenů [11].

Syntaxí řízený překlad

Syntaxí řízený překlad je nejoblíbenější architektonický model používaný pro návrh překladače. Použití tohoto modelu uvažuje syntaktický analyzátor jako jádro překladu. Syntaktický analyzátor řídí činnost všech ostatních částí překladače a zasílá jim požadavky na akce, které je třeba vykonat. Na požádání si od lexikálního analyzátoru žádá další tokeny a volá přidružené sémantické akce, které mohou přímo generovat kód. Tato architektura je dnes považována za nepsaný standard [27], jež je vždy první možností, kterou při vývoji překladače volíme.

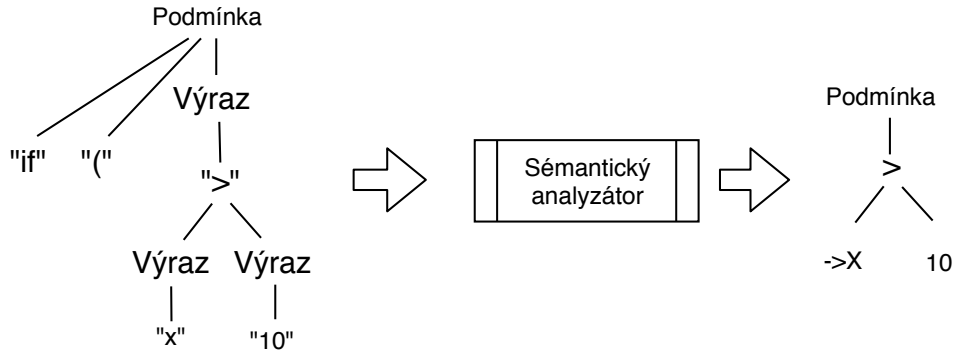
LL syntaktický analyzátor

LL syntaktický analyzátor je syntaktický analyzátor používající metodu *shora-dolů*. Analyzuje vstup *zleva* (**L**eft) doprava a konstruhuje *nejlevější* (**L**eft) *derivaci věty* [12]. *LL syntaktický analyzátor* umí rozpoznávat *pouze* LL gramatiky (definice 2.1). *LL syntaktický analyzátor* je v současné době nejpoužívanější analyzátor formálně definovaného jazyka, jelikož jej lze velmi snadno zkonstruovat a všechny syntaktické konstrukce vyšších programovacích jazyků lze za pomocí LL gramatiky rozpoznávat.

Definice 15 *Říkáme, že LL syntaktický analyzátor je typu LL(∗), pokud při rozpoznávání jazyka není omezen pevně danou hodnotou parametru k [26].*

Sémantická analýza

Na fázi *lexíkální* a *syntaktické* analýzy navazuje fáze nazvaná *Sémantická analýza*, která je vykonávána *sémantickým analyzátorem*. Vstupem je simulace *konstrukce derivačního stromu* z fáze *syntaktické analýzy*, výstupem je *abstraktní syntaktický strom*[15]. Znázornění činnosti *sémantického analyzátoru* je na obrázku 2.5 [11].



Obrázek 2.5: Znázornění činnosti sémantického analyzátoru.

Abstraktní syntaktický strom vzniká procházením *derivačního stromu*. Do stromové struktury jsou doplňovány implicitní konverze a jsou odstraňovány gramatické konstrukce, které už pro nás v této fázi nemají žádný význam. Dochází zde ke kontrolám sémantických aspektů programu. Takovými aspekty může být *kontrola typů*, *kontrola deklaraci proměnných*, *kontrola definic funkcí*, *redefinice funkcí* nebo *testování dřívějšího výskytu nějakého symbolu*. Při provádění kontrol může sémantický analyzátor provádět implicitní konverze konverze datových typů [16]. Sémantické kontroly jsou nejčastěji reprezentovány jako procedury volané ze syntaktické analýzy. *Sémantická analýza* používá pro kontroly různé datové struktury, do kterých si ukládá informace o analyzovaném kódu. Mohou to být zásobníky, seznamy nebo tabulky symbolů. Nejčastěji používanou datovou strukturou je *tabulka symbolů*. *Tabulka symbolů* se používá pro uložení záznamu proměnných nebo funkcí. Překlad různých paradigmat si však žádá různé datové struktury. Zde se neřídíme žádnými striktními pravidly, ale požadavky cílového jazyka a architekturou překladače, která je u každého jazyka zcela individuální.

Při použití *syntaxí řízeného překladu* nenavazuje sémantická analýza přímo na syntaktickou analýzu, ale jednotlivé sémantické rutiny jsou volány za pomocí sémantického analyzátoru, který řídí celý proces překladu. Při selhání sémantické kontroly dojde v této fázi k ukončení překladu a hlášení sémantické chyby.

Generování vnitřního kódu

Na *sémantickou analýzu* navazuje fáze *generování vnitřního kódu*, kterou provádí *generátor vnitřního kódu*. Překladač či interpret v této fázi vytváří *nízkoúrovňový kód*, jehož znakem

[15] Abstraktní syntaktický strom je reprezentací programu ve formě stromu, který je ale již skutečně sestaven ve formě datové struktury. Tato struktura je určena k pozdějšímu zpracování.

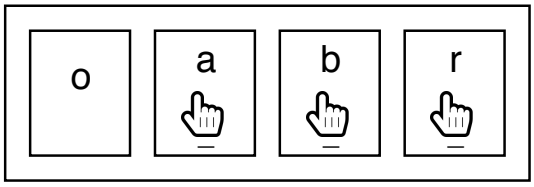
[16] Při nekompatibilních datových typech se sémantický analyzátor pokusí provést konverzi. Benevolence konverzí záleží na mnoha aspektech. v případě, že jazyk poskytuje slabou typovou kontrolu je benevolentnost vyšší než u silně typové kontroly. V případě nemožnosti tuto konverzi provést vrací sémantický analyzátor sémantickou chybu.

je *jednoduchost* a *jednotnost*. Jednotnost je důležitá zejména z toho důvodu, že více různých programovacích jazyků může využívat stejný vnitřní kód. V takovém případě mohou překladače těchto jazyků sdílet všechny navazující fáze překladu[17].

Kód musí být snadno sestavitelný a snadno přeložitelný do cílového kódu. Nejčastěji vytvářený vnitřní kód má podobu takzvaného tříadresného kódu. Pro generování tříadresného kódu existují tři základní přístupy [12]:

1. syntaktický analyzátor vytváří abstraktní syntaktický strom, který je převeden na tříadresný kód,
2. syntaktický analyzátor vytváří posfixovou reprezentaci programu, která je převedena na tříadresný kód,
3. syntaktický analyzátor vytváří tříadresný kód přímo.

Instrukce vyjádřená za pomocí tříadresného kódu má dle dostupné literatury [12] standardizovanou strukturu. Znázornění struktury je na obrázku 2.6.



Obrázek 2.6: Znázornění standardizované struktury tříadresného kódu.

Kde význam jednotlivých symbolů je následující [11]:

- **o** - Operátor, reprezentován například symbolem $(+, -, \%, ...)$,
- **a** - První operand,
- **b** - Druhý operand,
- **r** - Výsledek operace.

Vnitřní kód vytváříme z několika různých důvodů. Těmi nejčastějšími mohou být [11]:

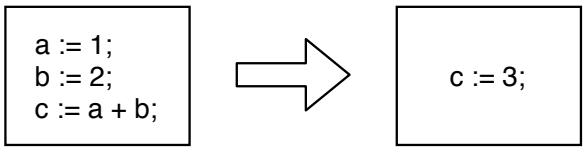
- možnost velmi snadno v následující fázi vnitřní kód optimalizovat,
- přímý překlad do cílového programu by byl velmi složitý a neprůhledný,
- vnitřní kód může být pro různé vstupní jazyky jednotný,
- vytváření vnitřního kódu zjednoduší generování cílového kódu.

[17] Příkladem takového překladače je například překladač *gcc*, který podporuje překlad z více různých jazyků. Nejznámějšími z nich jsou *C*, *C++* a *Objective-C*.

Optimalizace

Další fází je *optimalizace vnitřního kódu*, která je vykonávána *optimalizátorem vnitřního kódu*. Vstupem je *vnitřní kód* a výstupem je *optimalizovaný vnitřní kód*. Úkolem *optimalizátoru* je upravit vnitřní kód tak, aby byl efektivnější. Takto upravený vnitřní kód se nazývá *optimalizovaný vnitřní kód*. *Optimalizovaný vnitřní kód* musí být funkčně ekvivalentní s *vnitřním kódem*. Ukázka dvou ekvivalentních kódů je na obrázku 2.7.

Optimalizace vnitřního kódu probíhá většinou po blocích kódu nazývaných *základní bloky*. Základní blok je sekvence příkazů, které jsou v tomto pořadí vždy provedeny. Neexistuje skok dovnitř ani ven z této sekvence. Fáze optimalizace je volitelná. Může být v některých případech z překladu zcela vypuštěna. Většina překladačů obsahuje optimalizátor, u kterého si uživatel může za pomocí parametrů nastavit, zda a na jaké úrovni jej využije[18]. Eliminace optimalizací může velmi zkrátit dobu překladu. Jelikož uživatelé vysokoúrovňových jazyků obvykle píší velmi neefektivní kód, je optimalizace v současně době v rámci vývoje překladače nejvíce diskutovaným tématem. Při procesu optimalizace se využívá různých technik, od grafových algoritmů, umělé inteligence až po velmi pokročilé experimentální metody typu psychologie uživatele, které jsou nicméně zatím stále ve fázi raného vývoje [11].



Obrázek 2.7: Ukázka možné optimalizace vnitřního kódu.

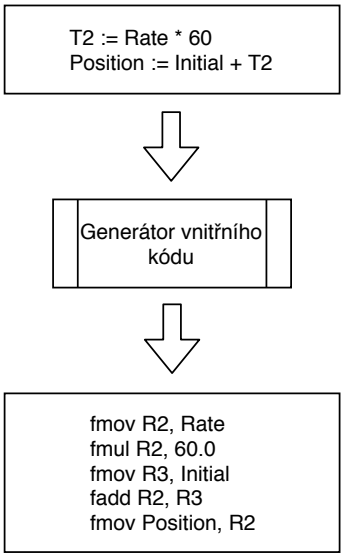
Generování cílového kódu

Poslední fází překladu je *generování cílového kódu*, což provádí generátor cílového kódu. Vstupem je optimalizovaný vnitřní kód a výstupem je cílový kód v cílovém jazyce. Program, který je výstupem překladače, musí být ekvivalentní s cílovým programem v cílovém jazyce. Ve většině případů je cílovým jazykem jazyk symbolických instrukcí neboli *assembler*. Může se však jednat o jakýkoliv programovací jazyk (definice 10). Fáze generování cílového kódu je typická pouze pro překladače. Ukázka generování cílového kódu je na obrázku 2.8.

Interpret na rozdíl od překladače cílový kód negeneruje. Místo toho provádí přímo vykonávání optimalizovaného vnitřního kódu tak, že na výstup generuje interpretovaný výstup vstupního programu.

Generátor cílového kódu musí především zvládnout přesnou syntax cílového jazyka. Musí generovat kód bez lexikálních, syntaktických a sémantických chyb a s ohledem na cílovou architekturu. V případě generování assembleru vzniká kód, který je silně závislý na cílové architektuře. Pro každou architekturu je tak třeba generovat kód podle specifických kritérií.

[18] Dobrým případem takového překladače je například *gcc*. Zde si uživatel může vybrat ze tří úrovní optimalizace, přičemž nejvyšší (experimentální) úroveň nemusí v některých případech zajistit zcela korektní překlad.



Obrázek 2.8: Znázornění možného generování cílového kódu.

2.4 Generování částí překladače

Při tvorbě překladače můžeme použít dva různé přístupy. První, dnes již méně typickou možností, je, že vytváříme celý překladač na zelené louce. Tento způsob předpokládá, že si všechny formální modely implementujeme sami v programovacím jazyce, který pro vývoj překladače používáme. Tento způsob je sice o hodně pracnější a nepřináší příliš mnoho výhod, v praxi se však občas stále používá. Jako příklad můžeme uvést programovací jazyk *C#*. Důvody, které k tomu vedou, jsou především absolutní kontrola nad zdrojovým kódem, možnost neomezeného přizpůsobení požadavkům vyvíjeného jazyka a lepší možnosti implementace optimalizací. Tyto výhody často u velkých firem převýší přednosti, které přináší možnost druhá. Tou je použití specializovaných nástrojů, které na základě dobře srozumitelného a čitelného popisu ve formě gramatik, regulárních výrazů či jiných formalismů některé části překladače generuje. Následující seznam popisuje ty nejužívanější z nich:

1. **LEX** je program sloužící ke generování *lexikálních analyzátorů*. Je běžnou součástí unixových operačních systémů a stal se nedílnou součástí standardu POSIX. Na vstupu čte specifikaci *lexikálního analyzátoru* zapsanou pomocí *regulárních výrazů* a na výstupu k němu poskytuje zdrojový kód zapsaný v programovacím jazyce *C*, který tento analyzátor implementuje [9].

2. **YACC** je program sloužící pro generování *syntaktických analyzátorů*. Byl vyvinutý jako implicitní *syntaktický analyzátor* pro platformu Unix. *Syntaktický analyzátor* vygenerovaný programem YACC potřebuje ještě *lexikální analyzátor* vygenerovaný například programem LEX nebo Flex. Vstupem je seznam gramatických pravidel a výstupem je zdrojový kód zapsaný v programovacím jazyce C implementující tato pravidla jako LR syntaktický analyzátor [5].

3. **ANTLR** nebo také **AN**other **T**ool for **L**anguage **R**ecognition (sekce 6.3) je nástroj pro rozpoznávání jazyka. Obsahuje v sobě více nástrojů. Na vstup přijímá gramatiku v rozšířené Backusově–Naurově formě (sekce 2.1), a na výstupu dokáže generovat všechny druhy analyzátorů z prvních tří fází překladu (sekce 2.3), tedy lexikální,

syntaktické a částečně také sémantické analýzy. Především je to nástroj pro generování lexikálního analyzátoru a nástroj pro generování syntaktického analyzátoru typu $LL(*)$ (definice 15). Od výše zmíněných nástrojů se liší především tím, že poskytuje pokročilé nástroje pro vizualizaci derivačních stromů nebo možnost detekce chyb v jednotlivých fázích překladu.

ANTLR dokáže na výstup generovat zdrojový kód v širokém spektru jazyků dle libosti uživatele. Nejčastěji je spojován s programovacími jazyky *Java* (sekce 6.1), *Python* a *C++*.

Kapitola 3

Software pro matematické výpočty

Následující kapitola poskytuje základní přehled o softwaru, který se běžně používá pro zadávání profesionálních matematických výpočtů. U jednotlivých nástrojů jsou uvedeny jejich základní způsoby a případy použití. Dále jsou popsány způsoby zápisu, který tyto nástroje používají. Pro placené nástroje je v práci také přiložen přehled základních licencí, se kterými lze aplikaci pořídit.

3.1 Matlab, GNU Octave

Matlab je inženýrský nástroj sloužící k realizaci profesionálních matematických výpočtů, analýzu dat, vizualizaci, vývoj algoritmů a technickou simulaci. Je využíván vědci a studenty po celém světě. Při řešení problémů bývá vždy první volbou. Je považován za standard. Nástroj je dostupný pro operační systém Windows, Linux i OS X. Balík *Matlab* se v základní podobě skládá ze dvou částí. První z nich je skriptovací programovací jazyk a druhou interaktivní grafické programové rozhraní. Zásadní vlastností Matlabu je fakt, že veškeré objekty jsou považovány za prvky pole (matice). v těchto strukturách lze uchovávat nejen reálná čísla, ale i komplexní čísla, proměnné, obrázky, zvuk nebo video [15].

Jádro aplikace *Matlab* je vyvíjeno jako vysoce modulární. Vývojáři a komunitou je poskytnuto rozsáhlé spektrum modulů, které rozsah jeho schopností dále rozšiřují do nejrůznějších odvětví použití. Nejznámějším modulem je nadstavba *Simulink* pro modelování a simulaci dynamických systémů. Poskytuje uživateli možnost velmi rychle vyvíjet modely dynamických soustav. Tyto modely jsou zadávány ve formě blokových schémat.

Matlab je poskytován jako komerční aplikace. Výsledná částka, kterou je třeba za aplikaci zaplatit, se může lišit podle účelu, za kterým hodláme aplikace pořídit. Seznam typů licencí, ve kterých lze aplikaci získat je popsán v rámci přílohy D.

Výpočetní jazyk aplikace Matlab

Jazyk, který interpret zabudovaný do aplikace *Matlab* zpracovává, lze klasifikovat jako skriptovací programovací jazyk. Je inspirován více různými programovacími jazyky, z nichž přebírá různé vlastnosti a různá programovací paradigmata. Může být zadáván buďto ze souboru obsahujícího zdrojový kód, z grafického uživatelského rozhraní nebo v rámci interaktivní konzole. Jedná se o jazyk multi-paradigmativní.

Obecně lze jazyk klasifikovat jako imperativní[1]. Jazyk je typovaný. Jeho interpret poskytuje slabou typovou kontrolu. Velkou zajímavostí je citlivost na velikost písmen u identifikátorů. Matlab není celkově citlivý na velikost písmen. Je citlivý na velikost písmen u názvů proměnných a vestavěných funkcí. U funkcí uložených v externím souboru není citlivý na velikost písmen na platformě *UNIX*, ale je citlivý na velikost písmen na platformě *Windows*.

GNU Octave

Program *Matlab* je celosvětově nejpoužívanější software pro řešení matematických výpočtů. Jedná se však o komerčně dostupnou aplikaci s velmi vysokou pořizovací cenou. To lidé preferující svobodný software nejsou ochotni akceptovat. V rámci svobodného software vznikl v roce 1988 program GNU Octave, který je částečně kompatibilní s programem *Matlab*. Program GNU Octave obsahuje velké množství nástrojů z oblasti nelineárních rovnic, integrování a diferenciálních rovnic. Na rozdíl od programu *Matlab* se těší velké podpoře od komunity svobodného software, která pro něj vyvíjí velké množství modulů [1].

3.2 TKSL, FOS

Aplikace *TKSL* slouží primárně k řešení soustav obyčejných diferenciálních rovnic [2]. K jejich řešení využívá unikátní metody Taylorovy řady [4] objasněné v habilitační práci docenta Kunovského vydané na fakultě Informačních technologií Vysokého učení technického v Brně. Je implementována v programovacím jazyce *C/C++*, díky čemuž umožňuje běh na všech moderních operačních systémech a zaručuje maximální možnou rychlost výpočtu.

Při řešení diferenciálních rovnic užívá *víceslovné aritmetiky* [3] a postupné inkrementace řádu metody. Díky tomu je dosaženo požadované přesnosti.

FOS

Aplikace *FOS* je reimplementací nástroje *TKSL* (sekce 3.2). Je implementována v jazyce *C++* s knihovnou *MPFR* [3]. Uživatelské rozhraní je implementováno jako webový nástroj [29], který k aplikaci na serveru přistupuje vzdáleně za pomocí zasílání HTTP požadavků. Lze ji volně využívat, jelikož je licencována pod MIT licencí. Výpočet zde probíhá na straně serveru a webový prohlížeč slouží pouze jako vizualizační nástroj. Poskytuje velmi názornou vizualizaci výsledku. Výsledek lze vizualizovat za pomocí videa, grafu a tabulky.

Výpočetní jazyk FOS

Jazyk, pomocí něhož je výpočet do aplikace FOS zadáván, lze klasifikovat jako jazyk deklarativní[4]. Zdrojový kód pro aplikaci *FOS* se skládá z několika sekcí. Sekce jsou následující:

[1] Při imperativním programování popisujeme řešení problému za pomocí posloupnosti příkazů, které přesně určují postup řešení.

[2] Aplikace TKLS se snaží všechny zadávané problémy převést na soustavu diferenciálních rovnic a tu poté řešit. Jedním z problémů řešených tímto způsobem je například výpočet určitého integrálu. Zájemce odkazuji na bakalářskou práci Ing. Petra Veigenda, která uvedenou problematiku rozebírá do hloubky [28].

[3] Víceslovná aritmetika je rozšířením klasické aritmetiky. Umožňuje na n-bitovém počítači pracovat s hodnotami, které mají i více než n bitů. Lze tedy na 8 bitovém počítači sčítat i 16 bitová čísla. V případě aplikace TKSL jde o rozšíření mantisy při výpočtech s plovoucí řadovou čárkou, díky čemuž získává výpočet extrémně vysokou přesnost.

[4] Při deklarativním programování říkáme zápisem pouze *co* se má udělat. Nikoliv *jak* se to má udělat.

1. **setup** - Tato sekce slouží pro nastavení parametrů výpočtu. Jedná se o blok konstant uvozený klíčovým slovem *setup*. Jednotlivé konstanty jsou odděleny středníkem. Název konstanty a její hodnota jsou pak odděleny znakem `'='`.

2. **graph/video** - v této sekci se nachází nastavení výstupu. Výstup může být dvojího druhu: *graf* nebo *video*. Struktura jednoho bloku označujícího výstup je velmi podobná jako u bloku *setup*. Název bloku je zde však uvozen klíčovým slovem *graph*.

3. **výpočet** - Obsahuje zápis výpočtu. Výpočet je zpravidla zadán pomocí funkcí a soustav diferenciálních rovnic prvního řádu. Lze však použít i mnohem bohatší konstrukce. Jejich význam lze nalézt v nápovědě aplikace *FOS* [29].

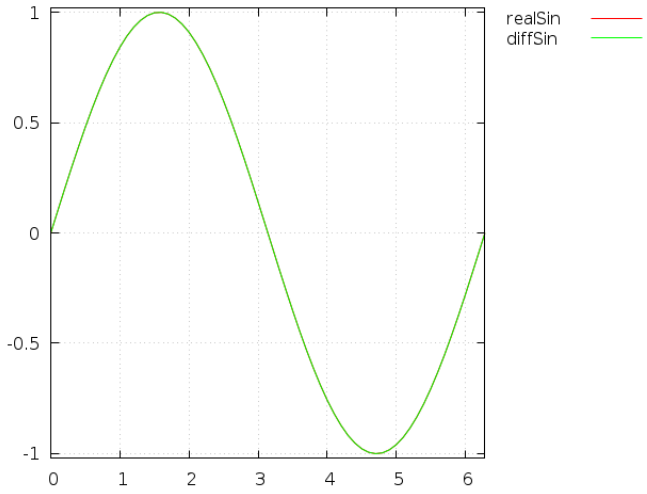
Následují ukázka demonstruje použití výpočetního nástroje *FOS*. Blokem *setup* je nastavena přesnost výpočtu na $1 \cdot e^{-15}$, délka kroku na 0.1. Čas, ve kterém má být simulace ukončena je nastaven na hodnotu $2 \cdot \pi$. Blokem *graph* jsou nastaveny parametry grafického výstupu. Do grafu jsou v tomto případě vloženy dvě funkce, *realSin* a *diffSin*. V poslední části jsou tyto funkce definovány. První z těchto funkcí je zadána přímo jejím matematickým přepisem, druhá za pomoci soustavy dvou diferenciálních rovnic prvního řádu pomocí niž je generována. Výsledkem výpočtu jsou dvě překrývající se funkce, což ukazuje vysokou přesnost výpočtu.

```
setup {
    dt = 0.1;
    tmax = 2 * PI;
    eps = 1e-15;
}

graph {
    show = realSin, diffSin;
}

realSin = sin(t);

diffSin' = z & 0;
z' = -diffSin & 1;
```

Výstupem je graf (obrázek 3.1) a tabulka hodnot (tabulka 3.1).

t	**realsin**	**diffsin**
0	0	0
0.1	0.09983341665	0.09983341665
⋮	⋮	⋮
6.283185307	3.531964132e-18	3.531964132e-18

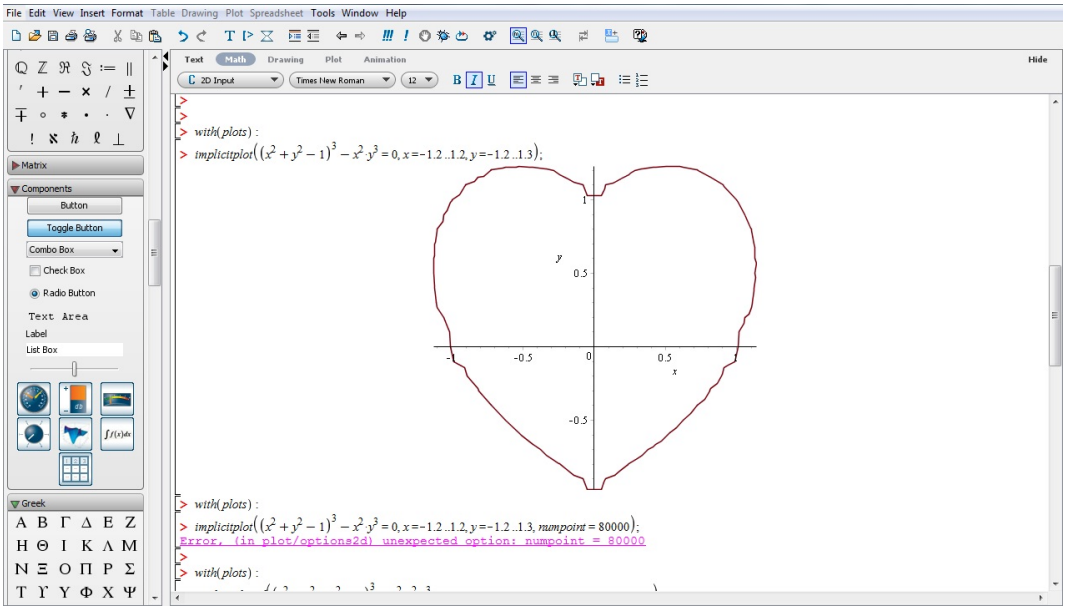
Tabulka 3.1: Ukázka výstupu aplikace *FOS* ve formě tabulky.



Obrázek 3.1: Ukázka výstupu aplikace *FOS* ve formě grafu.

3.3 Maple

Maple je počítačový program patřící do skupiny programů *CAS*, tedy Computer Algebra Systems. Je to program pracující se symbolickými výrazy pro numerické výpočty. Může být používán jako nástroj pro výpočty a také jako programovací jazyk, zčásti podobný Pascalu. Systém *MAPLE* je určen pro symbolické výpočty, má zázemí pro numerické řešení diferenciálních rovnic a hledání integrálů. Má také pokročilé grafické funkce. Rozdíl mezi programem *MAPLE* a předchozími je především v tom, že program *MAPLE* umožňuje vysoce interaktivní možnosti zadávání výpočtu bez znalosti programovacího jazyka, editaci matematických dokumentů a mnoho dalšího.



Obrázek 3.2: Ukázka grafického uživatelského rozhraní aplikace MAPLE.

Kapitola 4

Návrh výpočetního jazyka

Následující kapitola přesně a formálně popisuje jazyk *Calc2019*. V sekci 4.1 jsou popsány obecné vlastnosti jazyka. V následujících sekcích je poté jazyk rozebrán z různých hledisek. Zejména pak z hlediska lexikální výstavby a struktury zdrojového kódu z hlediska syntaktické analýzy.

4.1 Obecné vlastnosti

Jazyk *Calc2019*, je vysokoúrovňový programovací jazyk určený pro zadávání a řešení matematických výpočtů. Jedná se o jazyk interpretovaný. Z toho důvodu jej není problém provést na libovolné platformě, která podporuje spuštění interpretu. Je inspirován již existujícími jazyky, především pak jazyky *Python*, *Haskel* a *FOS* (sekce 3.2). Tyto jazyky jsou hojně rozšířené a umožňují užití vlastností, které jsou pro nás velmi výhodné. Prvky výsledného jazyka jsou silně inspirovány funkcionálními paradigmaty. Prakticky vše, co lze v jazyce vyjádřit, je považováno za funkci. Veškeré řídící konstrukce, matematické operace, definice proměnných prostředí a příkazy jsou interně reprezentovány jako funkce a podle toho je s nimi nakládáno.

4.2 Lexikální analýza

Interpret jazyka *Calc2019* podporuje pouze standardní ASCI znaky. Jiné než ASCI znaky mohou být použity, avšak pouze v rámci komentářů. Při zpracování zdrojového kódu jsou vynechávány bílé znaky. Za bílé znaky jsou interpretem považovány znaky popsané následujícím fragmentem:

```
fragment BLANK    : [\r\n\t ]+ ;
```

Jazyk *Calc2019* není citlivý na velikost písmen. Všechna písmena z alfanumerických lexém jsou interně reprezentována jako malá písmena. Následující dvě lexémy jsou tedy ekvivalentní.

```
IdEnTiFiCaToR
```

Je ekvivalentní s

```
identificator
```

Jelikož jsou lexémy jazyka *Calc2019* dokumentovány pomocí zápisu pro generátor nástroje *ANTLR* (sekce 6.3), je třeba určitým způsobem zakódovat, že na daném místě může být jak velké, tak i malé písmeno. Toho je dosaženo odkazem na fragment zkráceně označující malé i velké písmeno. Například hodnota `T` označuje dvojici písmen `'T', 't'`. Od tohoto odstavce bude v zápisech vždy velké písmeno popisovat tuto dvojici.

Komentáře

Komentáře jsou při zpracování zdrojového kódu ignorovány. Jelikož není třeba zpracovávat jejich obsah, mohou na rozdíl od zdrojového kódu obsahovat libovolné znaky i mimo rozsah ASCII tabulky. Jazyk *Calc2019* podporuje komentáře dvojího druhu. Jsou jimi komentáře *jednořádkové* a *víceřádkové*. Komentáře nemohou být do sebe vzájemně vnořovány.

Jednořádkový komentář

Za jednořádkový komentář je považováno vše, co začíná znakem `#` a končí znakem konce řádku. Je tedy popsán následující lexémou:

```
SINGLE_LINE_COMMENT    : '#' ~[\r\n] ;
```

Víceřádkový komentář

Za víceřádkový komentář je považována libovolná posloupnost symbolů započatá sekvencí `/*` a končící sekvencí `*/`. Mezi těmito sekvencemi znaků se může vyskytovat libovolná (i prázdná) posloupnost symbolů. Víceřádkový komentář je popsán následující lexémou:

```
MULTILINE_COMMENT : '/*' .* '*/' ;
```

Literály

Literály jsou hodnoty, které přímo zadáváme do zdrojového kódu za účelem jejích zpracování. Jazyk *Calc2019* nám umožňuje použití následujících literálů:

Celé číslo vyjadřuje hodnotu celého kladného čísla. Na lexikální úrovni není identifikováno znaménko. Je definováno jako neprázdná posloupnost číslic. Formálně jej lze zapsat následující lexémou:

```
INTEGER_LITERAL   : ('0'..'9')+ ;
```

Desetinné číslo vyjadřuje hodnotu kladného čísla s desetinnou částí. Je definováno jako neprázdná posloupnost číslic obsahující desetinnou část a volitelný exponent. Povolené zápisy jsou totožné se zápisy z programovacího jazyka *Java*. Přesnou definici určuje následující lexéma:

```
FLOAT_LITERAL     : ('0'..'9')+ '.' ('0'..'9')* EXPONENT?
                  | '.' ('0'..'9')+ EXPONENT? | ('0'..'9')+ EXPONENT
                  ;

fragment EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+
                  ;
```

Logická hodnota vyjadřuje pravdivostní hodnotu. Může nabývat dvou hodnot, pravda a nepravda. Hodnoty jsou reprezentovány za pomocí klíčových slov `true` a `false`. Formálně zapsáno:

```
LOGIC_VALUE        : (T R U E | F a~L S E)
                   ;
```

Vektor je strukturovaný datový typ. Vyjadřuje posloupnost hodnot v *přesně definovaném pořadí*. Hodnoty ve vektoru se mohou opakovat. Vektor je heterogenní datová struktura[1]. Vektor může být i prázdný. Struktura vektoru je kvůli bezkontextové struktuře definována až na syntaktické úrovni, nicméně více zapadá do kontextu literálů. Místo formální definice vektoru za pomocí lexémy tedy uvádím několik příkladů vektoru tak, jak je lze přímo do jazyka zapsat:

```
[3, 2, 42]
[3.8, 3, true]
[3.9, 3, [3, 5, []]]
[[[[[]]]]]
```

Množina je strukturovaný datový typ. Vyjadřuje množinu hodnot, jejichž pořadí není definováno a jejíž hodnoty se nemohou opakovat. V případě, kdy do množiny vložíme hodnoty, které se v ní již vyskytují, nebudou znovu vloženy, ale bude nám k dispozici tato hodnota pouze jednou. Množina může být také prázdná. Jelikož množina je definována až na syntaktické úrovni, místo popisu lexémy uvádím několik zápisů množiny:

```
{3, 2, 1}
{3, 3.1, true}
{{3, 2}, true, {}}
{{{{{{}}}}}}
```

Vnořování datových typů Jednotlivé strukturované datové typy lze do sebe bez problémů vnořovat. Do vektorů lze vkládat jako prvky množiny, do množin lze jako prvky vkládat vektory. Tyto množiny a vektory jsou poté považovány za jednu ze složek strukturovaného datového typu, pro který platí stejná pravidla jako u složek skalárních. Ukázky vnořování datových struktur demonstruje následující fragment kódu.

```
{3, [3, 2, 1], [[[3]]]}
{3, [2, {4, [2, {4, [42, {21, [21, 45]}]}]}]}
[{[{[{[{[{}]}]}]}]}]
```

Identifikátory

Identifikátor je neprázdná posloupnost malých písmen, velkých písmen a číslic začínající malým či velkým písmenem. Identifikátor začínající číslem není považován za validní. Dále jsou za nevalidní považovány identifikátory náležící do množiny klíčových slov (sekce 4.2).

```
IDENTIFIER         : ('a'..'z' | 'A'..'Z' )('a'..'z' | 'A'..'Z' | '0'..'9')*
```

[1] Jednotlivé složky vektoru mohou být různého datového typu. Navíc je možné, aby složkou vektoru byl vektor.

Klíčová slova

Klíčová slova jsou rezervované identifikátory, které mají význam pouze v určitém kontextu. Uživatelsky definovaný identifikátor (sekce 4.2) nesmí spadat do množiny klíčových slov. Jazyk *Calc2019* podporuje následující klíčová slova:

```
in, else, not, equals, true, false, less, bigger, equals,
xor, or, typeof, and, is
```

Operátory

Operátor je v programovacích jazycích symbol používaný ve výrazech, který předepisuje provedení nějaké (nejčastěji matematické nebo logické) operace s hodnotami (operandy) zapsanými ve svém okolí. Z lexikálního hlediska jsou v jazyce *Calc2019* vyčleněny jako operátory následující sekvence symbolů. Jako operátory mohou být použity také některá klíčová slova (sekce 4.2). Jejich sémantický význam je definován v tabulce operátorů (tabulka 4.2).

```
'\'', '.', '%', '+', '-', '*', '/', '^', '!', '|', '(', ')',
'[', ']', '{', '}', '_', ',', ':', '==', '=', '!=', '=!',
'>', '<', '<=', '=<', '>=', '=>', '<-'
```

4.3 Syntaktická výstavba jazyka

Celý zdrojový kód je pouze lineární sekvencí příkazů jdoucích logicky po sobě. Definujeme 4 typy příkazů. Pro oddělení příkazů není určena žádná syntaktická konstrukce. Lze využít libovolné posloupnosti bílých znaků, zejména pak znaku konce řádku. Syntaktický analyzátor sám dokáže identifikovat a rozdělit jednotlivé příkazy.

Definiční příkaz

Definiční příkaz slouží k nastavení vlastností výpočtu. Může se jednat o nastavení proměnných prostředí, případně o nastavení vlastností jednotlivým modulům. Pomocí takto nastavených vlastností nastavujeme hodnoty, které jednotlivé moduly pro své potřeby zpracovávají. Dopsáním dalšího modulu je možné další příkazy rezervovat pro své použití. Definiční příkaz se skládá z názvu modulu, kterému chceme definovat vlastnosti. Dále následuje seznam vlastností uzavřených do složených závorek. Vlastnosti jsou odděleny čárkou. Jedna vlastnost sestává z dvojice $(jméno, hodnota)$, jméno je od hodnoty odděleno posloupností dvou znaků `<-` nebo klíčovým slovem `is`. Hodnoty mohou nabývat všech podporovaných datových typů. Tyto typy jsou *Celé číslo*, *Desetinné číslo*, *Pravdivostní hodnota*, *vektor*, *množina* a *textový řetězec*. Následující blok kódu demonstruje použití všech datových typů

```
module {
    name1 <- 10,
    name2 <- 10.0,
    name3 <- true.
    name4 <- [1, 2, 3],
    name5 <- {4, 5, 6},
    name6 <- "euler"
}
```

Interpretace hodnot je závislá na cílovém modulu. Modul neznámé vlastnosti ignoruje a ty, které se nacházejí v doméně, kterou umí zpracovávat, zpracuje.

Matematický výraz

Význam operátorů i jejich zápis je silně uzpůsoben použití jazyka. Jsou navrženy tak, aby co nejvíce připomínaly matematický zápis. Význam operátorů je shrnut v následující tabulce. Výraz je konečná posloupnost symbolů obsahující volání funkcí, konstanty a operátory (sekce 4.2). Pro operace jsou vybrány stejné symboly, které se běžně v matematice používají. Symboly vyjadřující matematickou operaci mají navíc více ekvivalentních zápisů pro to, aby výsledek byl více čitelný. Tabulka 4.1 obsahuje všechny operátory, které lze v jazyce *Calc2019* použít. Tabulka 4.2 obsahuje úplný seznam operací, které lze použít.

TYPEOF : typeof	EQUALS : ('==' || equals)
NOT_EQUALS : ('~=' || '=~')	LESS : ('<' || less)
NOT_LESS : ('~<' || <~)	BIGGER : ('>' || bigger)
BIGGER_EQUAL : ('>=' || '=>')	LESS_EQUAL : ('<=' || '=<')
AND: ('&&' || and)	OR: ('||' || or)
NOT : ('~' || not)	XOR : ('^^' || xor)
LESS_EQUAL : ('<=' || '=>')	IN : in
FACTORIAL : '!'	EXP : "
ADD : '+'	SUBTRACT : '-'
MULTIPLY : '*'	DIVIDE : '/'
DERIVE : ' ' '	

Tabulka 4.1: Přehled všech podporovaných operandů.

Definice funkcí

Jelikož je celý programovací jazyk *Calc2019* založen na funkcionálních paradigmatech, jsou všechny příkazy jazyka definovány pomocí funkcí. Funkcemi jsou nahrazeny řídící konstrukce, jako funkce jsou reprezentovány také veškeré matematické operace a jako funkce jsou reprezentovány i proměnné. Funkce mohou být zadávány pomocí několika možných způsobů zápisu.

Funkce jako matematický výraz

Funkce může být zapsána jako matematický výraz. v takovém případě je funkce jednořádkovou konstrukcí skládající se z názvu funkce, seznamu argumentů uzavřených do závorek, symbolu přiřazení a matematického výrazu, ve kterém mohou být použity názvy vstupních argumentů. Jako symbol oddělující definici hlavičky funkce a její tělo lze použít dvě různé ekvivalentní posloupnosti. Jsou jimi klíčové slovo `IS` a lexéma `<-`. Následující fragment demonstruje tento způsob zápisu.

```
funkce1(x) <- x + 1
funkce2(y) is log(y)
```

Operátor	**Význam operace**	**Výsledná hodnota**
LITERAL	Hodnota literálu	Hodnota literálu
expr TYPEOF type	Ověření datového typu	true/false
expr1 EQUALS expr2	Stejnost dvou hodnot	true/false
expr1 NOT_EQUALS expr2	Negovaná stejnost dvou hodnot	true/false
xpr1 LESS expr2	První hodnota menší než druhá	true/false
xpr1 NOT_LESS expr2	Negace první hodnta menší než druhá	true/false
xpr1 BIGGER expr2	První hodnota větší než druhá	true/false
xpr1 NOT_BIGGER expr2	Negace první hodnota větší než druhá	true/false
xpr1 BIGGER_EQUAL expr2	První hodnota větší nebo rovna druhé	true/false
xpr1 LESS_EQUAL expr2	První hodnota menší nebo rovna druhé	true/false
xpr1 AND expr2	Logický součin dvou hodnot	true/false
xpr1 OR expr2	Logický součet dvou hodnot	true/false
xpr1 XOR expr2	Logický exkluzivní součet dvou hodnot	true/false
NOT expr2	Logická operace negace	true/false
(expr)	Prioritní vyhodnocení výrazu	expr
| expr |	Absolutní hodnota z výrazu	expr
expr1 IN expr2	Náležitost množina	true/false
expr1 NOT IN expr2	Negace náležitosti množině	true/false
expr FACTORIAL expr	Výpočet faktoriálu	integer
expr1 EXP expr2	Operace mocnina	expr
expr1 ADD expr2	Součet dvou hodnot	expr
expr1 SUBTRACT expr2	Rozdíl dvou hodnot	expr
expr1 MULTIPLY expr2	Součin dvou hodnot	expr
expr1 DIVIDE expr2	Podíl dvou hodnot	expr
SUBTRACT expr	Záporná hodnota	expr
ADD expr	Kladná hodnota	expr
IDENTIFIER	Hodnota funkce bez argumentů	expr
IDENTIFIER(args)	Hodnota funkce s argumenty	expr
IDENTIFIER DERIVE	Derivace	funkce

Tabulka 4.2: Úplný seznam navržených operátorů a výsledek, který poskytují.

Definiční obor funkce lze velmi snadno omezit pomocí omezující funkce. Omezující funkce je funkce jedné neznámé, která vrací pravdivostní hodnotu. V případě, že funkce vrátí hodnotu `true`, může proběhnout výpočet hodnoty vykonáním těla funkce. v opačném případě je vrácena konstanta `undefined`. Návrh jazyka obsahuje také několik předdefinovaných funkcí, které tuto činnost vykonávají. Jsou jimi funkce `is_N(x)`, `is_Z(x)`, `is_R(x)`, `is_BOOL(x)`, `is_VECTOR(x)` a `is_SET(x)`. V následující ukázce jsou definovány tři funkce, `funkce1` má jeden argument, jehož hodnota je omezena na reálná čísla. `funkce2` má jeden argument, jehož hodnota je za pomocí dříve definované funkce `suda` omezena na sudá čísla.

```
sude(x in is_Z): is_BOOL <- x % 2 == 0
funkce1(x in is_R) <- x + 1
funkce2(x in sude) is x * 2
```

Stejným způsobem, kterým omezujeme definiční obor funkce, můžeme omezit také obor hodnot funkce. V případě, že se funkce pokusí vrátit hodnotu, která by nebyla v souladu s oborem hodnot, dojde k chybě za běhu.

```
funkce1(x in is_BOOL) : is_BOOL <- NOT x
```

Jazyk *Calc2019* neobsahuje žádné syntaktické konstrukce pro tvorbu proměnných. Proměnné lze však velmi dobře emulovat jako funkce, které nemají žádné vstupní argumenty. V takovém případě je dokonce možné zcela vynechat závorky okolo potenciálních argumentů. Zápis takové funkce lze provést za pomocí dvou ekvivalentních zápisů.

```
funkce3 <- (20 + 31) * 6
funkce4() <- (20 + 31) * 6
```

Funkce s podmíněnými bloky

Činnost funkce lze rozdělit na jednotlivé bloky. Tyto bloky lze poté vykonávat podmíněně. Pomocí této skutečnosti lze velmi dobře emulovat podmínky a cykly, které známe z imperativních programovacích jazyků. Funkce s podmíněnými bloky má následující zápis:

```
funkce5(x in is_N) <- {
    x % 2 == 0 <- x + 1
    true <- x + 2
}
```

V případě, že pro kombinaci vstupních hodnota není nalezen odpovídající blok, je vrácena hodnota `undefined`. Tato hodnota může být vrácena i při striktním omezení datového typu. Následující funkce tak pro sudá čísla vrací hodnotu `true`. Pro všechna ostatní čísla vrací konstantu `undefined`.

```
funkce6(x in is_N) <- {
    x % 2 == 0 <- true
}
```

Bloky lze samozřejmě do sebe libovolně vnořovat. Maximální úroveň zanoření není žádným způsobem programově omezena. Následující ukázka demonstruje výpočet absolutní hodnoty pro všechna celá čísla. Stejného efektu by bylo možné dosáhnout také za pomocí omezení definičního oboru. v takovém případě by ale při zavolání s hodnotou špatného datového typu program skončil s běhovou chybou.

```
funkce7(x) <- {
    x typeof is_N <- {
        x > 0 <- x
        x < 0 <- (-x)
    }
}
```

Diferenciální rovnice

Požadavkem na jazyk *Calc2019* byl také modul pro řešení diferenciálních rovnic. Jelikož jazyk *Calc2019* si zakládá na uživatelské přívětivosti a maximálním možném napodobení skutečného matematického zápisu, byla pro zápis diferenciální rovnice potřeba speciální gramatika. Řešení diferenciálních rovnic tedy nebylo navrženo jako knihovna, ale jako přímá součást jazyka. Modul umí řešit lineární diferenciální rovnici vyššího řádu.

Diferenciální rovnice zapisujeme přímo ve tvaru, ve kterém je uvažujeme jako matematici bez jakýchkoliv úprav. Diferenciální rovnice se skládá ze dvou částí. První částí je samotná rovnice a druhou jsou počáteční podmínky. Tyto dvě části jsou odděleny znakem `'&'`. První část je složena ze členů diferenciální rovnice. Ty jsou ve tvaru :

```
expr * IDENTIFIER(')*
```

Počet symbolů `'` určuje řád derivace [7]. Maximální řád derivace není nijak omezen. Parametry diferenciální rovnice lze nastavit pomocí definičního příkazu (sekce 4.3), který má následující tvar:

```
differential {
    resulting_function <- "x",
    enforcing_function <- "y",
    step_size <- 0.1,
    time_max <- 1,
    method <- "runge_kutta"
}
```

Kde `resulting_function` je jméno hledané funkce a `enforcing_function` je jméno vynucující funkce. Parametr `method` určuje metodu použitou pro realizaci integrátoru. v rámci demonstrační aplikace byly implementovány tři numerické metody:

- Eulerova metoda [8] při použití konstanty `euler`
- Metoda Runge–Kutta 4. řádu [8, 20] při použití konstanty `runge_kutta`
- Metoda Adams-Bashforth [8] při použití konstanty `adams_bashforth`

Výsledná diferenciální rovnice (např. druhého řádu) poté může být zapsána jako:

```
37*y''' + 22*y'' = 89 & y''(0) = 1, y''(0) = 0, y'(0) = 0, y(0) = 0
```

Chceme li získat výsledek diferenciální rovnice, jednoduše jako příkaz použijeme název funkce. Název bude interpretován jako výraz a vyhodnocen. v současné verzi jazyka nelze výsledek diferenciální rovnice začlenit do výrazu, nebo s ním provádět další matematické operace. Nelze tak k výsledku přičíst konstantu a požadovat vykreslení.

Kapitola 5

Srovnání jazyka Calc2019 s existujícími řešeními

Následující kapitola poskytuje srovnání jazyka *Calc2019* s řešeními, která poskytují konkurenční platformy. Cílem kapitoly je ukázat pozitivní vlastnosti jazyka, které mohou uživateli ulehčit rozhodnutí, zda pro své potřeby použít jazyk *Calc2019*, nebo se poohlédnout po jiném řešení.

5.1 Operace

Matematické operace jsou ve většině programovacích jazyků implementovány jako funkce. Znaky, které běžně matematici používají pro označení matematických operací jsou často použity pro zápis jiných syntaktických konstrukcí. Uveďme si několik příkladů.

Absolutní hodnota

V programu *Matlab* (sekce 3.1) je potřeba pro získání absolutní hodnoty volat specializovanou funkci. Kód, pomocí něhož absolutní hodnotu z literálu vložíme do proměnné by vypadal takto:

```
y = abs(-5)
```

V jazyce *Calc2019* lze pro výpočet absolutní hodnoty použít znak `'|'`, který mnohem více odpovídá matematickému zápisu:

```
y = |-5|
```

Další operace

Na podobném principu pracují také další matematické operace. Všechny jsou maximálně uzpůsobeny tomu, aby věrně napodobovaly skutečný matematický zápis. Operandy navíc mají více ekvivalentních zápisů, které lze mezi sebou libovolně zaměňovat. Následující zápisy jsou ekvivalentní:

```
a <= b      a ~> b
a =< b      a >~ b
```

Díky tomu si uživatel může zvolit tvar operandu, který je mu více blízký, a ten používat. Konkurenční platformy tuto možnost standardně nenabízí.

5.2 Zápis diferenciálních rovnice

Zápis diferenciálních rovnic konkurenční platformy často podporují ve formě soustavy diferenciálních rovnice prvního řádu. Do této soustavy je třeba tyto rovnice transformovat. Některé aplikace, například *Matlab* (sekce 3.1) podporují také zadání diferenciální rovnice pomocí blokového schématu. Budeme řešit následující diferenciální rovnici:

$$37 \cdot y''' + 22 \cdot y'' = 89$$

Pro rovnici jsou definovány následující počáteční podmínky:

$$\begin{aligned} y''(0) &= 0, \\ y'(0) &= 0, \\ y(0) &= 0. \end{aligned}$$

Při řešení této rovnice pomocí aplikace *FOS* (sekce 3.2) je třeba tuto rovnici převést na soustavu obyčejných diferenciálních rovnic, a poté řešit. Zápis by mohl vypadat například následovně:

```
setup {
    dt = 0.1;
    tmax = 5;
    eps = 1e-9;
    coef01 = ((-22)/37);
    coef02 = 87/37;
}

graph {
    show = [y];
}

p3y = coef01*p2y + coef02;
p2y' = p3y & 0;
py' = p2y & 0;
y' = py & 0;
```

Tuto soustavu je však třeba ručně získat. Jazyk *Calc2019* podporuje zápis diferenciální rovnice s lineárními koeficienty přímo. Zápis by vypadal takto:

```
differential {
    resulting_function <- "xyz",
    step_size <- 0.1,
    time_max <- 5,
    method <- "euler"
}

37*y''' + 22*y'' = 89 & y''(0) = 1, y'(0) = 0, y(0) = 0
```

Interpret jazyka z této soustavy získá koeficienty přímo a provede interně výpočet převedením na soustavu diferenciálních rovnic prvního řádu.

Kapitola 6

Použité technologie a implementace demonstrační aplikace

V následující kapitole budou popsány technologie, které byly použity pro implementaci demonstrační aplikace, jež je realizačním výstupem této bakalářské práce. Budou uvedeny důvody zvolení daných technologií a budou také diskutovány důvody, ze kterých technologie vyhovují požadavkům pro implementaci aplikace. Budou také rozebrány nedostatky a omezení použitých technologií. Dále bude specifikována struktura a implementace demonstrační aplikace, která je realizačním výstupem bakalářské práce.

6.1 Programovací jazyk

Při výběru programovacího jazyka jsem vycházel ze svých znalostí a zkušenosti a vybíral z několika kandidátů. V úvahu připadaly programovací jazyky *Java*, *C++*, *PHP*, *Javascript* a *Python*. Největší zkušenost mám s programovacími jazyky *PHP* a *JavaScript*. Velkou nevýhodou je však velmi nízký výkon a časově omezený běh programu na straně serveru. Programovací jazyk *JavaScript* navíc neposkytuje téměř žádné prověřené a v reálných případech použitelné nástroje pro rozpoznávání jazyka[1]. Pro zajištění spolehlivého provozu by tedy bylo potřeba běh výpočtu zpracovávat na straně serveru a o výsledek si žádat pomocí HTTP požadavků. Na podobném principu pracuje například webové rozhraní aplikace *FOS*, viz sekce 3.2. Programovací jazyky *C++* a *Python* jsem zavrhl z důvodu velmi malého množství zkušeností.

I přes nižší výkon jsem se tedy rozhodl vyvinout demonstrační aplikaci v programovacím jazyce *Java*. *Java* je objektově orientovaný, interpretovaný, robustní programovací jazyk představený firmou Sun Microsystems v roce 1995. Jde o jeden z nejpoužívanějších programovacích jazyků vůbec. Popularita tohoto jazyka sice v posledních letech mírně klesá, stále se ale jedná o jeden z nepoužívanějších jazyků pro vývoj komerčních aplikací. Většina velkých podnikových, zejména pak bankovních aplikací je založena právě na programova-

[1] Existuje implementace nástroje *ANTLR* (sekce 6.3) pro *JavaScript* [17], jedná se však o řešení, které stále trpí nedostatky nové technologie. Lze však očekávat, že během několika následujících měsíců či let dojde k opravě všech porodních bolestí nové technologie a nástroj bude v jazyce *JavaScript* stejně dobře použitelný, jako je tomu v jiných programovacích jazycích.

cím jazyce *Java*. Jeho největší předností je přenositelnost bajtového kódu[2]. Ten může být spuštěn na různých zařízeních počínaje stolními počítači, servery, chytrými telefony, televizemi, konče vestavěnými systémy nebo automobily [19]. Rozsáhlost a robustnost technologie *Java*[3] dovoluje vytvořit téměř libovolnou aplikaci.

Hlavním důvodem použití programovacího jazyka *Java* je jednoduchost, vysoká podpora pro objektově orientovaný návrh, který umožní velmi dobrou rozšiřitelnost aplikace v budoucnosti, podpora moderních knihoven pro tvorbu grafických uživatelských rozhraní a velmi rozsáhlá aktivní komunita. Dobrým důvodem je také multiplatformnost výsledné aplikace, která může být po provedení malých úprav přenositelná například na mobilní zařízení, televizory nebo vestavěné systémy [22]. Pro budoucí rozvoj aplikace, kdy bude třeba řešit především rychlost je také dobré zmínit možnost spouštění nativního kódu obaleného pomocí standardních objektů jazyka *Java*[4].

6.2 Grafické uživatelské rozhraní

V rámci demonstrační aplikace bylo třeba implementovat *grafické uživatelské rozhraní*. *Java* v tomto ohledu poskytuje několik možností. Jsou jimi technologie *AWT*, *Swing* a *JavaFX*. Zde byla volba velmi jednoduchá. Od roku 2009 jsou technologie *AWT* a *Swing* považovány za zastaralé. Je doporučeno je postupně nahrazovat modernějšími technologiemi z důvodu zastavení podpory pro opravy bezpečnostních děr a nové aplikace už vyvíjet pouze za pomocí technologie *JavaFX*.

JavaFX je open source softwarová platforma nové generace pro vytváření desktopových aplikací s bohatým a moderně vypadajícím grafickým uživatelským rozhraním. Jejím cílem je především to, aby výsledná aplikace byla z hlediska uživatelské přívětivosti jednoduše a intuitivně použitelná. Byla vytvořena pro programovací jazyk Java za účelem nahrazení zastaralého swingu. *JavaFX* má podporu pro stolní počítače na systémech *Windows*, *Linux* a *macOS*. *JavaFX* poskytuje téměř neomezené možnosti přizpůsobení vzhledu uživatelského rozhraní. Poskytuje podporu stylování komponent uživatelského rozhraní pomocí značkovacího jazyka *CSS*, možnost definice komponent vlastních a podporu nových návrhových vzorů (především pak *MVC* a *MVVM*) [25]. Je také velmi oblíbená pro rozsáhlou podporu moderních prvků, jako například průhlednost, animace nebo responzivní design.

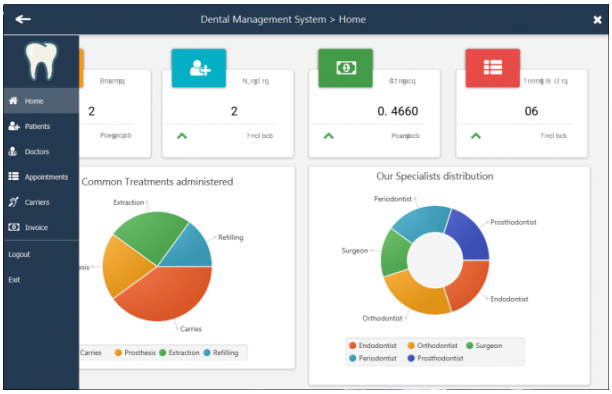
Možnosti návrhu

Návrh grafického uživatelského rozhraní pomocí technologie *JavaFX* (sekce 6.2) může probíhat dvěma způsoby. Prvním z nich z nich je definice uživatelského rozhraní pomocí souboru s příponou `.fxml`. Tento formát souboru byl vytvořen firmou Oracle Corporation. Jedná se o pro člověka srozumitelný zápis, který může být velmi snadno generován jako výstup editoru grafického uživatelského rozhraní. Takový editor je součástí každého solidního vývojového prostředí, které se dnes pro vývoj v absolutní většině případů používá. Existuje i velké množství externích vizuálních editorů, které editování tohoto souboru podporují. Jedním z nich je například *Scene Builder*, jehož vývoj zaštiťuje firma Gluon. Zápis jednoduché

[2] Zdrojový kód programovacího jazyka *Java* je kompilován do mezikódu, který nazýváme bajtový kód.

[3] Java není pouze programovací jazyk. Jedná se o rozsáhlý ekosystém sestávající z programovacího jazyka, překladače, virtuálního stroje a spousty dalších podpůrných nástrojů pro ladění kódů a další vývojářské potřeby.

[4] *Java Native Interface* je rozhraní, které programátorovi umožňuje přístup ke knihovnám vyvinutým v kompilovaných jazycích (sekce 2.2). To přináší znatelné zvýšení rychlosti, které může v optimálních případech dosahovat i rychlosti nativní aplikace.



Obrázek 6.1: Ukázka graficky bohatého uživatelského vytvořeného pomocí technologie JavaFX [14].

scény pomocí souboru *FXML* demonstruje následující fragment kódu. Jde o jednoduchý formulář obsahující jedno tlačítko vložené do jednoho kontejneru:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.AnchorPane?>
<AnchorPane prefHeight="400.0" prefWidth="600.0"
    xmlns="http://javafx.com/javafx/8.0.121"
    xmlns:fx="http://javafx.com/fxml/1"
    fx:controller="interpret.Ahoj">
   <children>
      <Button layoutX="269.0" layoutY="187.0" mnemonicParsing="false"
      text="Button" />
   </children>
</AnchorPane>
```

Druhou možností je tvorba grafického uživatelského rozhraní pomocí přímého vytváření objektů ve zdrojovém kódu. Objekty jsou skládány pomocí návrhového vzoru kompozice a tvoří tak strom, který reprezentuje grafické uživatelské rozhraní aplikace. Tento přístup k tvorbě grafického uživatelského rozhraní pochází ze starších technologií typu *Swing*, kde se jednalo o jedinou možnost tvorby rozhraní.

V praxi se běžně používá kombinace obou přístupů, kdy většina prvků aplikace je definována pomocí `.fxml` souboru a komplikovanější prvky, u kterých není možné tento přístup použít, jsou poté extra dodefinovány pomocí objektového přístupu.

6.3 ANTLR

Pro aplikaci bylo třeba vytvořit demonstrační interpret deklarativního jazyka, který poskytuje velmi vysokou úroveň abstrakce. Jelikož tvorba interpretu je netriviální záležitost, rozhodl jsem se použít pro tvorbu některých úvodních fází interpretace nástroj, který provádí jejich syntézu z formálního popisu. Takový nástroj dovoluje odstínit se od nízkoúrovňových záležitostí a řešení opakujících se problémů, poskytne nám možnosti pro ladění interpretu a usnadní vývoj do takové míry, že se můžeme zaměřit především na návrh jazyka jako takového.

Výběr nástroje

Při výběru nástroje byla určena následující kritéria:

- kompatibilita s programovacím jazykem *Java*,
- jednoduchost použití,
- aktivní vývoj,
- rozsáhlá a aktivní komunita,
- obsáhlá dokumentace.

Všechna kritéria bez výhrady splňuje nástroj *ANTLR*. Konkurenční technologie *LEX* a *YACC* nebyly vybrány z důvodu nedostatečné podpory pro programovací jazyk *Java* a nízkého množství podpůrných nástrojů.

ANTLR neboli **A**nother **T**ool for **L**anguage **R**ecognition[5] je nástroj, který umožňuje generovat zdrojový kód pro první fáze překladu, tedy *lexikální analyzátor*, *syntaktický analyzátor* a v omezené formě i *sémantický analyzátor*. Nástroj generuje syntaktický analyzátor typu $LL(*)$ (definice 15). Překonává ostatní nástroje především tím, že lze všechny podporované fáze překladu popsat jednotně v rámci jediného souboru s příponou *.g4*. Jiné nástroje jako LEX nebo YACC se specializují pouze na jednu fázi překladu. Dále ANTLR poskytuje rozsáhlé možnosti vizualizace a ladění překladače. Jako výstup může poskytnout zdrojový kód v několika různých programovacích jazycích dle libosti uživatele. Nabízí podporu pro jazyky *Java*, *C++*, *C#*. Na podpoře pro další programovací jazyky se velmi aktivně pracuje.

Způsob použití

Nástroj zpracovává gramatiku v *rozvinuté Backus-Naurově formě* (sekce 2.1). Jsme tedy schopni velmi efektivně a v krátkém čase gramatiku jazyka zapsat a odladit. Výstupem činnosti tohoto nástroje je zdrojový kód implementující analyzátor jazyka, který vstupní gramatika popisuje.

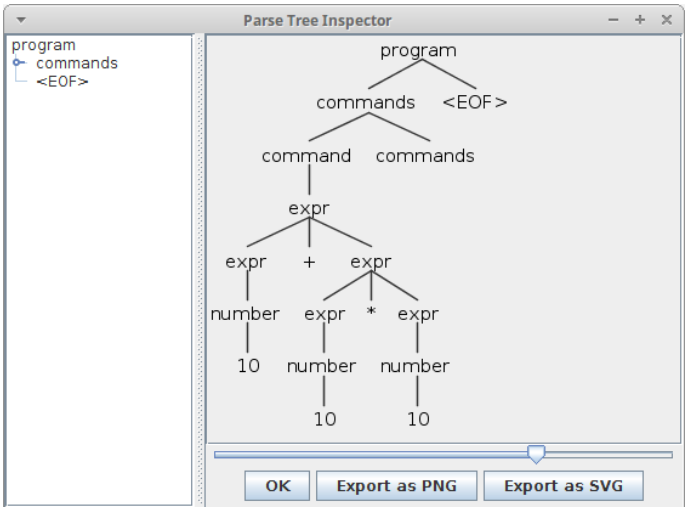
Následující fragment kódu [18] ukazuje zápis jednoduché gramatiky pro zpracování sekvence jednoduchých matematických výrazů. v první části je definováno jméno gramatiky. Za pomocí tohoto jména se poté na tuto gramatiku můžeme odkazovat. Ve druhé části je definován popis syntaktického analyzátoru. V poslední části jsou definována lexikální pravidla. Jelikož v rámci jednoho souboru definujeme jak lexikální, tak syntaktická pravidla, platí zde jedno omezení. Názvy syntaktických pravidel jsou psány malými písmeny a názvy jednotlivých lexém jsou psány velkými písmeny.

```
// Jméno gramatiky
grammar Expr;

// Syntaktická pravidla
prog    : (expr NEWLINE)*
        ;
expr    : expr ('*'|'/') expr
```

[5] Na českých webových serverech lze *ANTLR* někdy nalézt také pod českým překladem *Jiný nástroj pro rozpoznávání jazyka*.

```
        | expr ('+'|'-') expr
        | INT
        | '(' expr ')'
        ;

// Lexikální pravidla
NEWLINE : [\r\n]+
        ;
INT     : [0-9]+
        ;
```

ANTLR umí vstupní gramatiku zpracovávat několika různými způsoby. Na rozdíl od ostatních nástrojů také poskytuje velké množství nástrojů pro ladění gramatiky a její optimalizace. Nejvýznamnější nástroj, kterým se ANTLR odlišuje od jiných nástrojů, je nástroj pro vizualizaci derivačního stromu. *ANTLR* v tomto případě na vstup přijímá dvojici *(gramatika, zdrojový soubor s programem)*. Na základě této dvojice vizualizuje derivační strom demonstrující použitá gramatická pravidla. Ukázku použití nástroje pro vizualizaci derivačního stromu můžete vidět na obrázku 6.2.



Obrázek 6.2: Ukázka vizualizace derivačního stromu nástrojem *ANTLR GUI*.

Výstup

ANTLR poskytuje na výstup několik souborů pojmenovaných na základě jména gramatiky. Prvním z nich je `<jmeno_gramatiky>Lexer.java` obsahující implementaci lexikálního analyzátoru. Druhým souborem je `<jmeno_gramatiky>Parser.java`. Ten obsahuje implementaci syntaktického analyzátoru. Předpřipravené třídy, které nám *ANTLR* poskytuje, poté využíváme k ovládání těchto těchto modulů a jejich vzájemnému provázání. Zpravidla vytváříme proud tokenů, který použijeme jako vstup pro syntaktický analyzátor (sekce 6.3).

Další soubory jsou závislé na vstupních argumentech. *ANTLR* umí generovat výstup založený na dvou různých přístupech. *Visitor* a *Listener*. Použití každého z přístupů má své klady a zápory a závisí na tom, k čemu přesně ANTLR chceme použít.

Při použití *Listeneru* jsou metody reprezentující syntaktická pravidla volány automaticky a naším úkolem je pouze definovat akce, které se mají při jejich zavolání provést.

v případě *Visitoru* je volání vnořených pravidel ponecháno v plné režii programátora. Při použití *Listneru* nemohou metody reprezentující pravidla gramatiky vrátit hodnotu. U *Visitoru* může metoda reprezentující pravidlo vrátit hodnotu libovolného datového typu. *Listener* používá explicitně zásobník přidělený na haldě, zatímco *Visitor* používá k zanořování do hlouběji volaných pravidel rekurzivní volání metod potomka. To může vést k výjimkám *StackOverFlow* u velmi hlubokých zanoření. Proto se při reálném použití nástroje *ANTLR* vyplatí provést detailní analýzu a zvolit přístup, který bude plně vyhovovat potřebám našeho řešení. Následující ukázka demonstruje typický způsob použití nástroje při programování v jazyce *JAVA*. v ukázce je použit přístup *Visitor*.

```
CalcLexer calcLexer = new CalcLexer(charStream);
CommonTokenStream tokenStream = new CommonTokenStream(calcLexer);
CalcParser calcParser = new CalcParser(tokens);
ParseTree parseTree = calcParser.program();
new DeriveTreeVisitor().visit();
```

V kódu lze snadno identifikovat vytvoření lexikálního analyzátoru ze vstupního proudu znaků[6], vytvoření proudu tokenů, jeho napojení na syntaktický analyzátor a zahájení syntaxí řízeného překladu, který využívá přístup *Visitor* (sekce 6.3).

6.4 LaTeX

LaTeXje balík maker programu TeX, který umožňuje autorům textů sázet a tisknout svá díla ve velmi vysoké typografické kvalitě. Vstupem pro sázecí systém LaTeX je prostý textový dokument obohacený o transformační příkazy, které definují význam a vzhled jimi ohraničených úseků textu, podobně jako například ve značkovacím jazyce *HTML*. Takto vytvořený textový soubor je poté možné přeložit do výstupního formátu dvi. Z tohoto formátu je poté možné provést konverzi do standardně rozšířeného formátu PDF. Systém je dostupný pro všechny nejpoužívanější operační systémy. Jeho největší výhodou je, že sám dokáže velmi dobře hlídat typografické aspekty textu. Uživateli tak téměř nedává možnost dopustit se nejčastějších typografických chyb. Výstupní dokument je absolutně nezávislý na operačním systému, na kterém byl proveden překlad. Pro dosažení vysoké typografické úrovně LaTeX používá své vlastní fonty. Další lze doplňovat za pomocí balíčků [13].

LaTeX vznikl především pro sazbu vědeckých a odborných publikací obsahujících velmi mnoho matematických vzorců a technických zápisů. Tyto zápisy není efektivní vytvářet a modifikovat za pomoci vizuálních editorů. Ukázalo se, že zápis formou strukturovaného značkovacího jazyka je pro tento typ dokumentů mnohem výhodnější. Na obrázku 6.3 můžete vidět rovnici vysázenou za pomoci technologie LaTeX.

$$\int_1^\infty \frac{1}{x^2}\,dx = \left[-\frac{1}{x}\right]_1^\infty = 1$$

Obrázek 6.3: Ukázka rovnice vysázené za pomocí LaTeXu.

[6] Proud vstupních znaků může být vytvořen ze standardního vstupu, souboru, socketu a dalších zdrojů poskytujících textová data.

KaTeX

KaTeX je nejrychlejší vykreslovací jádro pro matematické výrazy zapsané ve formátu pro interpret LaTeXu. Je implementován v programovacím jazyce *JavaScript*. Původně byl navržen pouze pro internetové aplikace, ale komponenty moderních programovacích jazyků umožňují použití i pro desktopové aplikace. KaTeX používá pro vykreslování data binding, a proto nepotřebuje pro znovuvykreslení obnovit aplikaci. Při libovolné změně vstupních dat dojde k překreslení. Nemá žádné závislosti a může být proto snadno integrován do libovolného projektu. KaTeX podporuje také vykreslování na straně serveru. v takovém případě produkuje výstup jako prostý HTML dokument [2].

6.5 Implementace aplikace

Demonstrační aplikace, která je realizačním výstupem této práce je založena na technologiích popsaných v předchozích sekcích této kapitoly. Následující odstavce použití těchto technologií konkretizují a uvádějí do souvislostí.

Struktura aplikace

Aplikace se skládá z několika oddělených částí, jimiž jsou:

- interpret,
- grafické uživatelské rozhraní,
- textové uživatelské rozhraní.

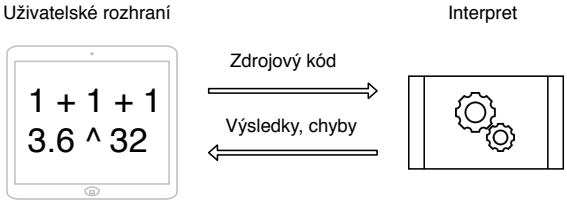
Obecná struktura interpretu

Implementace interpretu je situována do balíčku `interpret`. Je v rámci aplikace implementován jako oddělený modul, který na vstupu přijímá zdrojový kód zapsaný v jazyce *Calc2019* a na výstupu poskytuje v případě úspěšné interpretace kolekci výsledků a v případě neúspěšné interpretace kolekci chyb. Výsledek je implementován jako potomek objektu `Result` a chyba je implementována jako objekt `IntepretError`. Interpret je od zbytku aplikace zcela izolován, a lze jej proto použít zcela nezávisle. Zpracování výstupu interpretu je úkolem pro grafické, případně textové uživatelské rozhraní. Jelikož je interpret implementován zcela odděleně a nezávisle na zbytku aplikace, lze do implementace doplnit libovolné další uživatelské rozhraní[7]. Interpret používá ke své činnosti syntaxí řízený překlad, který svoji činností sestavuje abstraktní syntaktický strom, který je posléze interpretován třídou `ASTMachine`. Činnost celého interpretu zapouzdřuje třída `Interpret`, která nám také poskytuje rozhraní pro ovládání interpretu. Skrz toto rozhraní je možné s interpretem komunikovat.

Lexikální a syntaktický analyzátor

Z důvodu urychlení vývoje a snížení počtu programátorských chyb jsem fázi *lexikální* a *syntaktické* analýzy založil na nástroji *ANTLR* (sekce 6.3). To mi umožnilo věnovat mnohem více úsilí návrhu gramatiky a mnohem méně méně času samotnému programování. Práce

[7] V případě úspěchu aplikace se počítá s vytvořením uživatelského rozhraní pro mobilní telefony, tablety a další zařízení. Programovací jazyk Java 6.1 toto velmi snadno umožňuje.



Obrázek 6.4: Znázornění činnosti aplikace.

s nástrojem *ANTLR* je velice pohodlná. Výsledná gramatika programovacího jazyka je přiložena na CD v adresáři `\grammar`. Ke gramatice je přiložen také soubor `Makefile` a sada testovacích programů v adresáři `\cd\testing_programs`, pomocí kterých lze činnost gramatiky testovat. Za pomocí příkazu:

```
make test FILE_NAME=NAME
```

kde `NAME` je jméno testovacího souboru bez cesty a přípony dojde k provázání s nástrojem *ANTLR GUI* a vykreslení derivačního stromu zdrojového kódu ze souboru `Name`. Podmínkou korektního provedení vizualizace ovšem je, aby se ve stejném adresáři nacházel také soubor `antlr.jar` obsahující knihovnu nástroj *ANTLR* ve verzi 4. Pro získání nástroje lze použít následující příkaz:

```
make get_antlr
```

Dojde k automatickému stažení nástroje a jeho správnému pojmenování.

Sémantický analyzátor

Sémantická analýza je implementována jako seznam rutin volaných ze syntaktického analyzátoru při vykonávání syntaxí řízeného překladu. Dochází zde k sestavování abstraktního syntaktického stromu, který je poté přímo bez žádných dalších úprav interpretován za pomocí třídy `ASTMachine`. Abstraktní syntaktický strom se skládá ze tří částí:

- **Úložiště definic** z definičních příkazů je reprezentováno za pomocí třídy `PropertiesStorage`. Každý modul má poté přístup ke svým definicím.
- **Úložiště funkcí** reprezentováno za pomocí třídy `ListOfFunctions` je tabulka symbolů sloužící pro uchování funkcí. Do tabulky jsou uchovávány jak funkce předdefinované, tak funkce uživatelské. Jelikož jazyk *Calc2019* považuje proměnné za bezparametrické funkce, jsou taktéž ukládány do tohoto úložiště.
- **Seznam příkazů pro interpret** je datová struktura typu `List` uchovávající příkazy, které jsou později interpretovány interpretem. Příkazy mohou být algebraické výrazy nebo zadání diferenciálních rovnic. Při budoucím rozšíření aplikace není problém dopsat jakékoliv další rozšíření pro další příkazy.

Interpretace

Interpret je implementován za pomocí třídy `Interpret`. Tato třída slouží také jako černá skříňka, pomocí které s interpretem komunikujeme. Interpret je implementován jako cyklus,

který prochází seznam příkazů jeden po druhém a postupně je vykovává. Může přitom nahlížet do dalších datových struktur, které jsou součástí abstraktního syntaktického stromu. Na výstup poskytuje v případě korektní interpretace kolekci výsledků. V opačném případě vrací kolekci chyb.

Uživatelské rozhraní

Výsledná aplikace poskytuje dva způsoby ovládání. *Grafické* a *textové* uživatelské rozhraní. Plnou funkcionalitu poskytuje pouze grafické uživatelské rozhraní. Textová forma je určena pouze pro interpretaci programu obsahující algebraické výrazy, a poskytuje pouze textový výstup. Neposkytuje téměř žádnou vizualizaci výsledků.

Pro implementaci grafického uživatelského rozhraní bylo užito technologie *JavaFX* a návrhového vzoru *MVC*[8]. Soubory `FXML` reprezentující jednotlivé formuláře jsou situovány do adresáře `resources`. Kontrolery obsluhující formuláře jsou třídy jazyka *Java* uložené v balíčku `gui.controllers`. Za model potom považován samotný interpret. Poslední použitou technologií je technologie *HTML*. Té je užito pro zobrazení nápovědy, která je uživateli zobrazena za pomocí komponenty *WebView*. Detailní průvodce grafickým uživatelským rozhraním je v příloze B.

V rámci uživatelského rozhraní je také možné měnit lokalizaci aplikace a barvy použité pro zvýrazňování syntaxe. Překlady a soubor s nastavením tedy bylo třeba perzistentně uchovávat. v jazyce Java se pro tyto účely běžně používají *Properties soubory*. Jedná se o způsob uložení dat v několika různých formátech[9]. Knihovna jazyka Java nám poté umožňuje s těmito soubory pracovat jednotným způsobem. Při spuštění aplikace je důležité, aby v adresáři byly také adresáře s lokalizacemi a nastavením. v opačném případě nedojde ke korektnímu spuštění.

Ladění výpočtu

Původní ideou bylo také implementovat nástroj pro ladění, který by poskytoval možnost krokování výpočtu a interaktivní zobrazování aktuálních hodnot z běhové reprezentace. Jelikož časová dotace pro implementaci aplikace byla omezená, nebylo možné toto realizovat. Ladění bylo tedy implementováno pouze za pomocí chybových hlášek, které uživateli zobrazuje uživatelské rozhraní v panelu *Běhové informace*. Jsou rozlišovány lexikální chyby, syntaktické chyby, sémantické chyby a chyby za běhu.

[8] Uvažoval jsem také o návrhovém vzoru *MVVM*. Ten lze za určitých okolností pomocí *JavaFX* implementovat, ovšem pouze ve velmi omezené formě.

[9] Pro *Properties soubory* se nejčastěji používají formáty XML a PROPERTIES.

Kapitola 7

Závěr

Cílem práce bylo navrhnout nový programovací jazyk sloužící pro zadávání a řešení matematických výpočtů. Pro tento návrh bylo třeba implementovat interpret a demonstrační uživatelské rozhraní usnadňující zápis programu a umožňující efektivní vizualizaci výsledků. Dále bylo třeba srovnat efektivitu tohoto řešení s konkurenčními řešeními z hlediska uživatelské přívětivosti.

Jazyk, který v rámci práce vznikl byl pojmenován *Calc2019*. Jedná se o interpretovaný programovací jazyk, který je kombinací několika existujících programovacích jazyků (*Python*, *Haskel* a *FOS*), ze kterých přebírá některé užitečné aspekty a přidává nové. Sytaxe a sémantika jazyka je maximálně uzpůsobena pro zadávání matematických výpočtů.

Dále vznikla demonstrační aplikace, která podmnožinu tohoto jazyka implementuje. Implementace sestává ze tří částí. První a zároveň nejdůležitější částí je interpret, který podmnožinu tohoto jazyka vykonává. Druhou částí je grafické uživatelské rozhraní umožňující zadávání výpočtů v grafickém režimu a vizualizaci výsledků. Třetí částí je textové uživatelské rozhraní, které umožňuje spouštění výpočtu z konzole.

V demonstrační aplikaci je možné vyzkoušet konstrukce a aspekty navrženého jazyka. Přiložené CD obsahuje mimo jiné ukázky zdrojových kódů, které názorně demonstrují správné použití výsledného jazyka. v grafickém uživatelském rozhraní je možné výsledky vizualizovat do uživatelsky přívětivé podoby ve formě grafů a tabulek. Dále je uživateli umožněn export výsledků do většiny nejpoužívanějších formátů (*JPG*, *PNG*, *BMP*).

Cíle práce byly splněny. Jelikož byla v rámci práce z důvodu omezené časové dotace implementována pouze podmnožina navrženého jazyka demonstrující jeho klíčové vlastnosti, navazujícím cílem této práce je implementovat kompletní specifikaci a rozšířit možnosti grafického uživatelského rozhraní.

Během dokončování práce vzniklo velké množství nápadů, které by mohly v budoucnu aplikaci zajímavým způsobem rozšiřovat. Jedná se například o výpočty v módu klient-server, provádění náročných výpočtů v nižších kompilovaných programovacích jazycích typu *C* či *C++*, nebo možnost zadávání diferenciálních rovnic za pomocí blokových schémat. Jelikož časová dotace pro tvorbu demonstrační aplikace byla omezená, nebylo možné všechny tyto nápady realizovat. V případě zájmu je však možné počítat s budoucím rozšířením aplikace různými směry a naplnění všech zmíněných vizí.

Literatura

[1] Eaton, J.: GNU Octave: Programová dokumentace. 2018.
URL https://octave.org/doc/interpreter/

[2] Eisenberg, E.; Alpert, S.: KaTeX: Aplikační programátorské rozhraní. Březen 2019.
URL https://katex.org/docs/api.html

[3] Hanrot, G.: Knihovna MPFR pro vývoj v jazyce C++. 2009.
URL https://www.mpfr.org/

[4] Jiří, K.: Moderní metoda taylorovy řady. FEKT VUT, 1994, habilitační práce.

[5] Johnson, S. C.: Yacc: Nástroj pro generování syntaktických analyzátorů.
URL http://dinosaur.compilertools.net/#yacc

[6] Kovár, M.: Diskrétní matematika. Březen 2014.
URL http://matika.umat.feec.vutbr.cz/inovace/texty/IDA/CZ/IDA_plna_verze_CZ.pdf

[7] Krupková, V.: Matematická analýza: Studijní opora. Prosinec 2012.

[8] Kubíček, M.: *Numerické metody a algoritmy*. Praha: Vysoká škola chemicko-technologická, 2005, ISBN 80-7080-558-7.

[9] Lesk, M.; Schmidt, E.: Lex: Generátor lexikálních analyzátorů.
URL http://dinosaur.compilertools.net/lex/index.html

[10] MathWorks: Matlab: Licence. 2019.
URL https://www.mathworks.com/pricing-licensing.html

[11] Meduna, A.: *Automaty a jazyky: Teorie a aplikace*. Springer, 2000, ISBN 9781852330743.

[12] Meduna, A.: Formální jazyky a překladače: Studijní opora. Prosinec 2012.

[13] Mittelbach, Frank: Úvod do systému LaTeX. 2019.
URL https://www.latex-project.org/

[14] Mlayah, D.: Systém řízení schůzek v JavaFx. 2019.
URL https://javafx.orionbricetechnologies.co.ke/product/appointment-management-system-in-javafx/

[15] Moore, H.: *MATLAB pro inženýry (pátá edice)*. Pearson, 2017, ISBN 0134589645.

[16] Parewa Labs: Interpret proti Kompilátoru: Rozdíly mezi interpretem a kompilátorem. 2016.
URL https://www.programiz.com/article/difference-compiler-interpreter

[17] Parr, T.: ANTLR: Implementace pro Javascript. Březen 2019.
URL
https://github.com/antlr/antlr4/blob/master/doc/javascript-target.md

[18] Parr, T.: ANTLR: Programová dokumentace. Únor 2019.
URL https://github.com/antlr/antlr4/blob/master/doc/index.md

[19] Pecinovský, R.: *Java 9: Kompletní příručka jazyka*. Praha: Grada Publishing, 2018, ISBN 978-80-271-0715-5.

[20] Peringer, J.: Modelování a simulace: Studijní opora. Prosinec 2012.

[21] Přispěvatelé wikipedie: Backusova–Naurova forma: Wikipedie, Otevřená encyklopedie. 2018.
URL https://cs.wikipedia.org/wiki/Backusova%E2%80%93Naurova_forma

[22] Přispěvatelé wikipedie: Vestavný systém: Wikipedie, Otevřená encyklopedie. 2018.
URL https://en.wikipedia.org/wiki/Embedded_system

[23] Přispěvatelé wikipedie: Chomského hierarchie: Wikipedie, Otevřená encyklopedie. 2019.
URL https://cs.wikipedia.org/wiki/Chomsk%C3%A9ho_hierarchie

[24] Přispěvatelé wikipedie: LL gramatiky: Wikipedie, Otevřená encyklopedie. 2019.
URL https://en.wikipedia.org/wiki/LL_grammar

[25] Singh, P.: MVC a MVVM: Rozdíly mezi návrhovými vzory. Červen 2019.
URL https:
//medium.com/mobidroid/difference-between-mvc-and-mvvm-456ec67181f6

[26] Terence, P.; Kathleen, S. F.: LL(*): The Foundation of the ANTLR Parser Generator. 2011, [Online; posted 25-Januar-2019].
URL https://www.antlr.org/papers/LL-star-PLDI11.pdf

[27] Vavrečková, S.: Syntaxí řízený překlad. Únor 2016.
URL https://docplayer.cz/46652831-Syntaxi-rizeny-preklad.html

[28] Veigend, P.: *Semi-analytické výpočty určitých integrálů*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2011.
URL http://www.fit.vutbr.cz/study/DP/BP.php?id=12730

[29] Veigend, P.: FOS: webové rozhraní. Květen 2019.

[30] Češka, M.; Vojnar, T.: Teoretická informatika: Studijní opora. Prosinec 2018.

Příloha A

Obsah přiloženého CD

```
/
├── dist/ ............................Přeložená a zabalená aplikace.
│   ├── calc2019.jar .................Spustitelný soubor aplikace.
│   ├── translations/ ................Lokalizace aplikace.
│   │   └── *.xml ....................Jednotlivé jazyky.
│   ├── setting/ .....................Nastavení aplikace.
│   │   └── *.xml ....................Soubory s nastavením.
│   ├── help/ ........................Nápověda aplikace.
│   │   └── *.html ...................Soubory s nápovědou.
├── doc/ .............................Generovaná projektová dokumentace.
│   └── index.html ...................Hlavní soubor projektové dokumentace.
├── grammar ..........................Adresář s gramatikou.
│   ├── testing_programs .............Testovací zdrojové kódy pro
│   │                                 demonstraci gramatiky.
│   ├── Calc.g4 ......................Zdrojový kód gramatiky.
│   ├── Makefile .....................Soubor usnadňující překlad
│   │                                 a testování gramatiky.
├── src/ .............................Kopie pracovního adresáře.
│   ├── pom.xml ......................Projektový soubor pro plugin Maven.
│   ├── LICENCE ......................Licence práce.
│   ├── src/ .........................Zdrojové kódy aplikace
│   │   ├── main/ ....................Zdrojové kódy jednotlivých tříd
│   │   └── test/ ....................Jednotkové testy
├── test_programs/ ...................Odladěné testovací programy pro
│                                     interpret.
├── text/ ............................Text práce.
│   └── xkobel02.pdf .................Text práce ve formátu PDF.
└── text_source/ .....................Zdrojové soubory textu bakalářské
    │                                 práce vysázené v systému LaTeX.
    ├── obrazky-figures/ .............Grafické součásti práce.
    ├── zadani.pdf ...................Zadání práce ve formátu pdf.
    ├── bibliography.bib .............Normovaná bibliografie práce.
    ├── content.tex ..................Přílohy práce ve formátu pro LaTeX.
    └── prilohy.tex ..................Soubor usnadňující překlad práce.
```

Příloha B

Návod pro použití aplikace

Tato příloha popisuje základní použití aplikace.

B.1 Překlad

Aplikaci je možné přeložit za pomocí *Java Development Kit* a nástroje *Maven*. Překlad je možné provést dvěma způsoby.

1. **Pouze za pomocí nástroje Maven** Všechny konfigurace pro *Maven* jsou uloženy v souboru `pom.xml`. Překlad lze provést zavoláním příkazu `mvn package`. Výstup je generován do složky `target`. V adresáři je vytvořen soubor `bakalarka-1.0-SNAPSHOT-jar-with-dependencies.jar`, který reprezentuje cílovou aplikaci. Pro spuštění je třeba soubor spustit v adresáři, ve kterém se nachází i adresáře s nastavením, lokalizacemi a nápovědou.

2. **Pomocí nástroje Maven a souboru Makefile** Překlad si lze zjednodušit použitím pomocného souboru *Makefile*. Příkazem `make output` dojde k překladu a generování adresáře `output`. V adresáři se po úspěšném provedení nachází aplikace se všemi závislostmi.

B.2 Spuštění jednotkových testů

Některé části aplikace jsou pokryty jednotkovými testy. Veškeré jednotkové testy lze na přiloženém CD nalézt v adresáři `/src/src/test/java`. Nachází se zde několik jednotkových testů na funkci vnitřních částí interpretu a grafické uživatelské rozhraní. Testy je možné spustit pomocí sestavovacího nástroje *Maven* pomocí nástedujícího příkazu:

```
mvn test
```

B.3 Spuštění

Aplikaci je možno před spuštěním získat dvěma způsoby:

1. Přeložením zdrojových kódů, viz. B.1,

2. Z adresáře dodaného na přiloženém CD. Ten se nachází v adresáři `/dist`, viz. příloha A.

Aplikaci je možné spustit ve dvou režimech. V režimu *grafického uživatelského rozhraní* a *textového uživatelského rozhraní*.

B.4 Textové uživatelské rozhraní

Textové uživatelské rozhraní je velmi jednoduché. Jeho jediným úkolem je přijmout vstup a na standardní výstup vypsat v případě úspěchu výsledky. V případě neúspěchu na standardní chybový výstup vypíše chybu a skončí s návratovým kódem 1. Pro interpretaci zdrojového kódu v textovém uživatelském režimu lze použít následující příkaz:

```
java -cp .:Calc2019.jar main.CLIMain ZDROJOVY_SOUBOR
```

`Calc2019.jar` je zde jméno souboru s aplikací, a `ZDROJOVY_SOUBOR` je jméno souboru, který má být interpretován. Při použití textového uživatelského rozhraní je jméno zdrojového souboru povinné.

B.5 Grafické uživatelské rozhraní

Po spuštění grafického uživatelského rozhraní dojde k otevření hlavního okna aplikace. Ukázka hlavního okna a rozložení prvků v pracovním okně je na obrázku B.1. Pro spuštění aplikace v módu s grafickým uživatelským rozhraním lze použít následující příkaz.

```
java -cp .:Calc2019.jar main.Main [ SEZNAM_ZDROJOVYCH_SOUBORU ]
```

`Calc2019.jar` je zde jméno souboru s aplikací, a `SEZNAM_ZDROJOVYCH_SOUBORU` je seznam souborů, které mají být otevřeny. Seznam souborů je zadáván volitelně. V případě, že není zadán žádný soubor, dojde pouze k otevření hlavního okna aplikace. V případě, že jsou soubory zadány, dojde k otevření hlavního okna aplikace a zároveň jejich otevření v režimu editace.

Hlavní okno aplikace je velmi jednoduché. Pro ovládání aplikace lze použít *hlavní menu* a *tělo aplikace*.

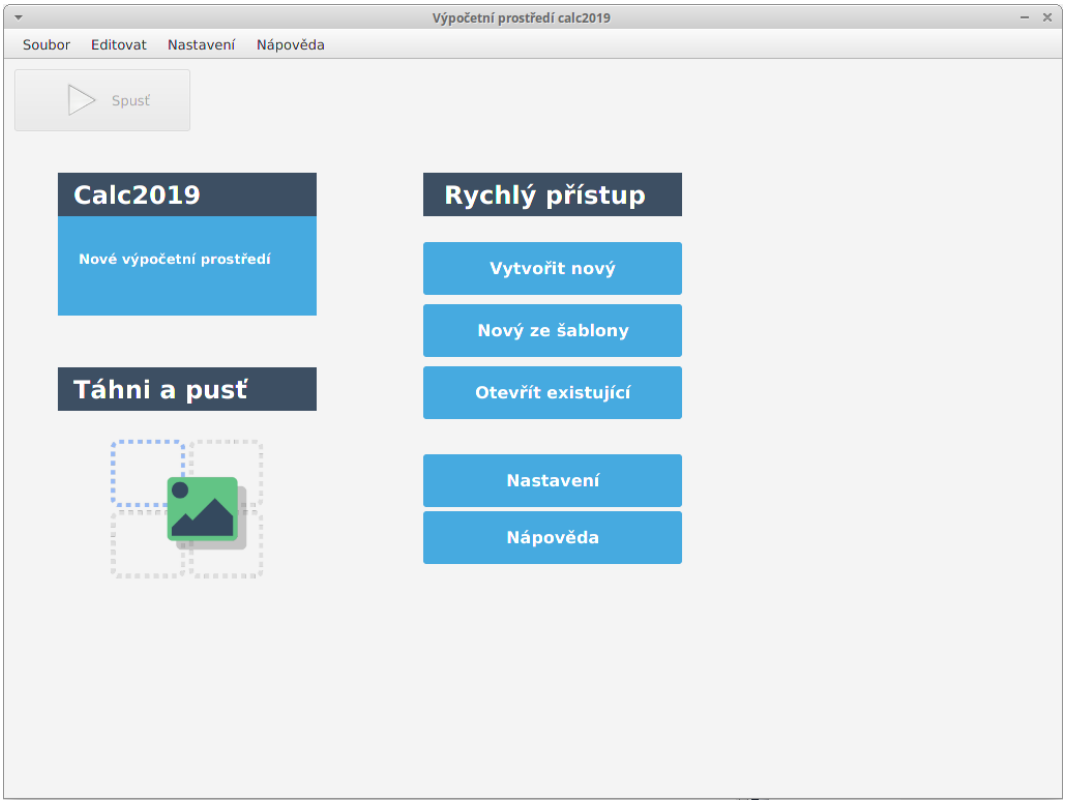
Hlavní menu

Hlavní menu se skládá ze čtyř položek. Jsou jimi:

- Soubor - Poskytuje základní práci se soubory, tedy jejich vytvoření, otevření, uložení a ukončení aplikace,
- Editovat - Poskytuje možnosti editace aktuálního souboru, tedy posouvání se zpět a vpřed v historii editací, kopírování, vyjmutí a vložení,
- Nastavení - Otevírá dialogové okno poskytující nastavení aplikace,
- Nápověda - Zobrazuje nápovědu a základní informace o aplikaci a použité licenci.

Tělo aplikace

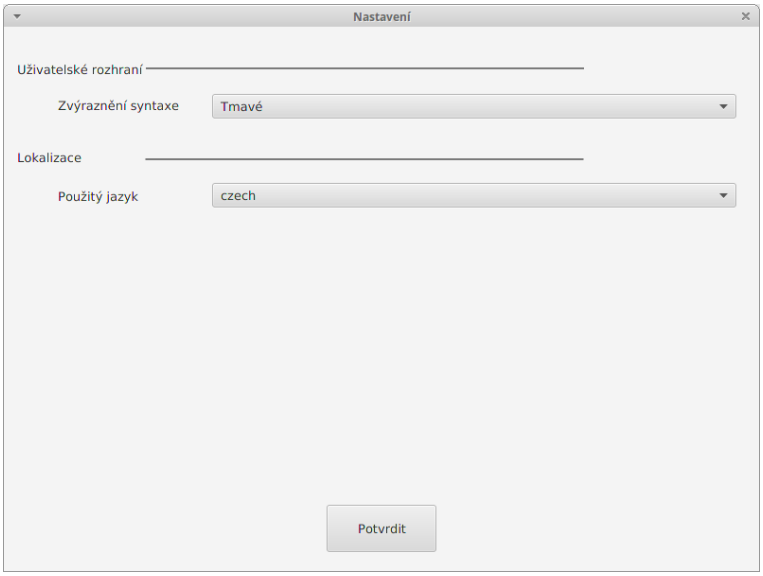
Tělo aplikace se skládá z několika tlačítek, kterými je možné jednoduše provést základní operace se soubory, spuštění aplikace a zobrazení nápovědy. Soubor je také možné otevřít pomocí metody *drag and drop*. Soubor stačí tažením myši přetáhnout do okna aplikace, a dojde k jeho otevření.



Obrázek B.1: Ukázka hlavního okna aplikace.

B.5.1 Nastavení

Nastavení je reprezentováno jedním dialogovým oknem. Dialogové okno je znázorněno na obrázku B.2.



Obrázek B.2: Ukázka dialogového okna umožňující nastavení aplikace.

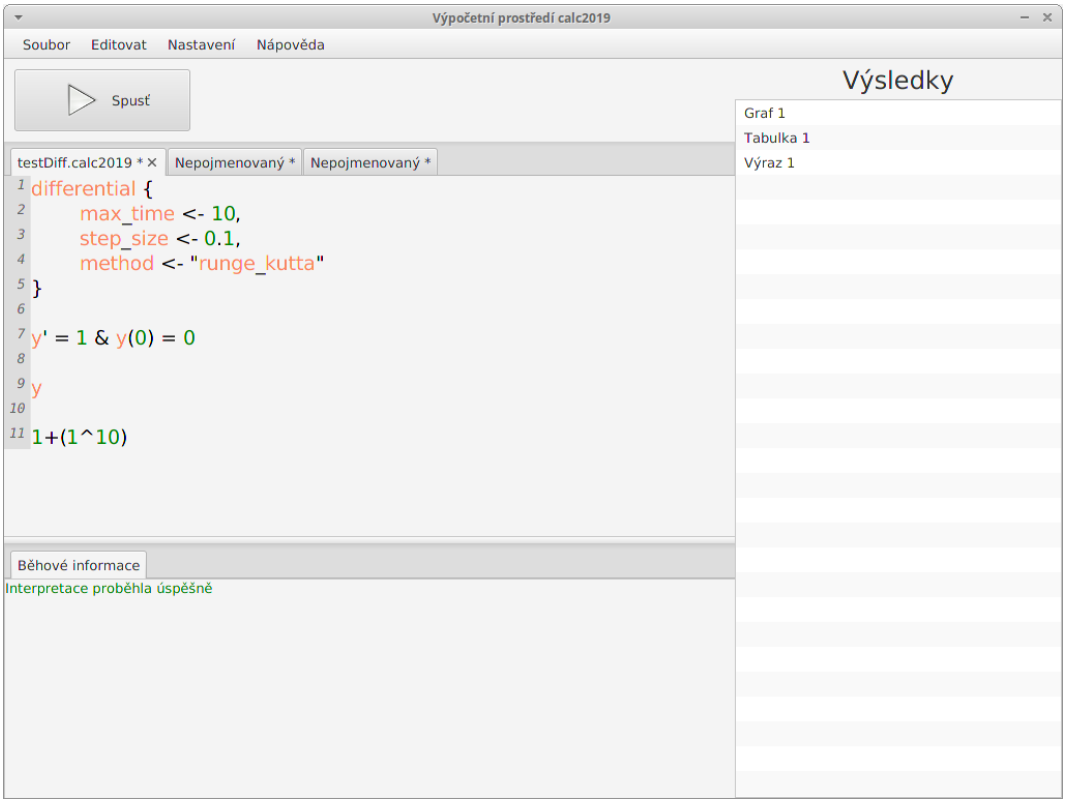
V nastavení můžeme modifikovat dvě vlastnosti. Jsou jimi:

- jazyk aplikace,
- zvýrazňování syntaxe.

K aplikování nastavení dojde okamžitě. Veškeré elementy grafického uživatelského rozhraní jsou implementovány pomocí návrhového vzoru *observer*, díky čemuž není třeba aplikaci restartovat.

B.5.2 Editování zdrojového souboru

Po otevření souboru nebo jeho vytvoření aplikace přejde do režimu editace. Rozložení hlavního okna aplikace se změní. Ukázka hlavního okna v režimu editace je na obrázku B.3.



Obrázek B.3: Ukázka editačního režimu aplikace.

V editačním režimu je hlavní okno rozděleno na několik částí:

- tlačítko *Spusť* slouží ke spuštění aktuálního programu,
- prostor pod tlačítkem *Spusť* je vyčleněn pro editaci zdrojového kódu. Najednou je možné mít otevřených více zdrojových souborů. K ovládání lze použít také klávesové zkratky:
  - `ctrl + n` - vytvoření nového souboru,
  - `ctrl + c` - kopírování aktuálně označeného textu do schránky,

  - `ctrl + v` - vložení aktuálně označeného textu ze schránky,
  - `ctrl + w` - uzavření aktuálně otevřeného a aktivního souboru,
  - `ctrl + tab` - posunutí na následující kartu.

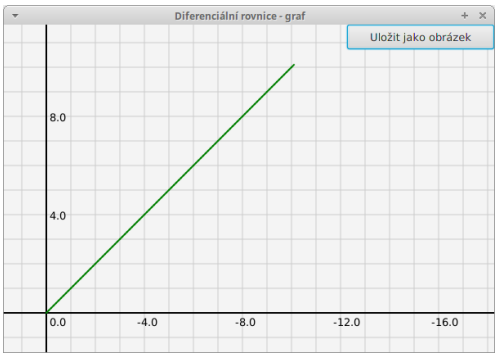
- *Běhové informace* slouží pro vizualizaci informací od interpretu. Slouží pro základní ladění výpočtu. V případě úspěšné interpretace se zobrazí zeleným písmem hlášení `'Interpretace proběhla úspěšně'`, v případě chyby při interpretaci se zobrazí červeným písmem chybové hlášení.

- *Výsledky* zobrazují seznam výsledků vzniklý interpretací.

B.5.3 Vizualizace výsledků

Po dvojkliku myší na výsledek v seznamu výsledků dojde k jeho vizualizaci. Aplikace podporuje tři vizualizační nástroje:

- **graf** slouží k vykreslení grafu funkce, která je výsledkem diferenciální rovnice. Výsledek je možné uložit jako obrázek do multimediálních formátů (*JPG*, *BMP*, *PNG*). Ukázka vizualizace jednoduché diferenciální rovnice je na obrázku B.4.



Obrázek B.4: Ukázka vizualizace výsledku diferenciální rovnice z obrázku B.3 za pomocí grafu.

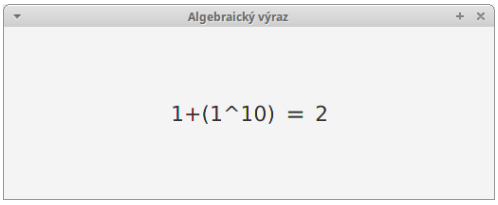
- **tabulka** slouží k zobrazení výsledku diferenciální rovnice za pomocí tabulky. Ukázka tabulky je na obrázku B.5.

Diferenciální rovnice - tabulka

Time	y
0.0	2.0
1.0	3.0
2.0	4.0
3.0	5.0
4.0	6.0
5.0	7.0
6.0	8.0
7.0	9.0
8.0	10.0
9.0	11.0
10.0	12.0

Obrázek B.5: Ukázka vizualizace výsledku diferenciální rovnice za pomocí tabulky.

- **algebraický výraz** slouží pro vizualizaci výsledku algebraického výrazu. Ukázka vizualizace algebraického výrazu je na obrázku B.6.



Obrázek B.6: Ukázka vizualizace výsledku algebraického výrazu.

Příloha C

Experimenty s programy zapsanými v jazyce Calc2019

Následující příloha ukazuje na konkrétních příkladech použití jazyka *Calc2019* při řešení reálných problémů.

C.1 Výpočet faktoriálu za pomocí funkce

Faktoriál lze velmi snadno vyčíslit za pomocí rekurzivního volání funkce. Toho lze v jazyce *Calc2019* velmi snadno dosáhnout. Následující program vyčísluje výraz 10!:

```
factorial(x in is_N) is {

    x in {0, 1} <- 1

    true <- x * factorial(x - 1)

}

factorial(10)
```

Výstupem je hodnota

```
3628800
```

C.2 Generování konstantní funkce

Problém generování funkce si lze snadno vyjádřit za pomocí *diferenciální rovnice*. Hledáme takovou diferenciální rovnici, jejíž řešení je funkce, kterou chceme generovat. Uvažujme generování konstantní funkce. Ta má tvar:

$$y(x) = c$$

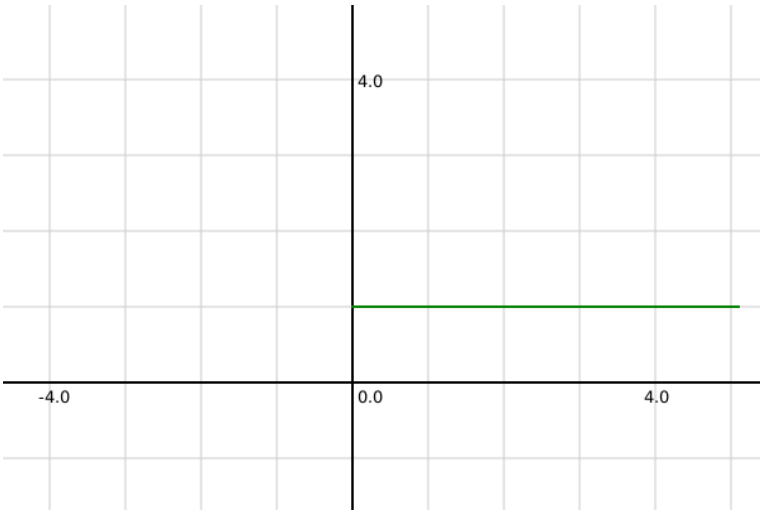
Tuto funkci lze velmi snadno generovat za pomocí obyčejné diferenciální rovnice prvního řádu:

$$y' = 0 \quad \& \quad y(0) = c$$

Program, který tuto rovnici řeší pro $c = 1$ zapsaný v jazyce *Calc2019* má následující zápis:

```
differential{
    step_size <- 0.1,
    time_max <- 5,
    method <- "euler"
}

y' = 0 & y(0) = 1

y
```

Výsledek simulace je na obrázku C.1.



Obrázek C.1: Znázornění výsledku generování funkce $y = 2$.

C.3 Generování lineární funkce

Předchozí příklad lze velmi snadno zobecnit na generování funkce lineární. Uvažujme funkci:

$$y = c1 \cdot x + c2$$

kde

- $c1$ je směrnice funkce a
- $c2$ je posunutí této funkce po ose y.

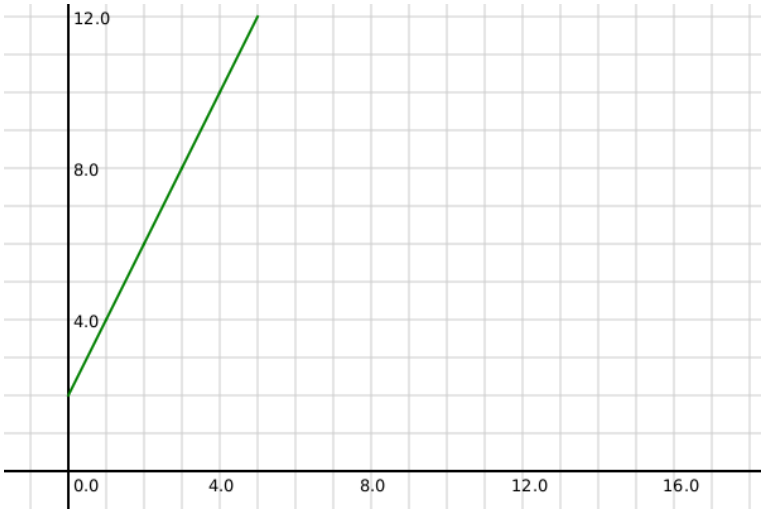
Tuto funkci lze reprezentovat za pomocí následující diferenciální rovnice:

$$y' = c1 \quad \& \quad y(0) = c2$$

Pro hodnoty parametrů $c1 = 2$ a $c2 = 2$ lze problém zapsat v jazyce *Calc2019* následujícím programem:

```
differential {
    step_size <- 1,
    time_max <- 5,
    method <- "euler"
}

y' = 2 & y(0) = 2

y
```

Výsledný graf je na obrázku C.2



Obrázek C.2: Znázornění výsledku generování funkce $y = 2 \cdot x + 2$.

C.4 Generování funkce e^x

Uvažujme funkci:

$$y = e^x$$

Tuto funkci derivujme. Výsledek je:

$$y' = e^x$$

Kombinací těchto dvou vztahů získáváme diferenciální rovnici:

$$y' = y$$

K rovnici je ještě třeba doplnit počáteční podmínku. Tu získáme dosazením do původní funkce:

$$y(0) = e^0 = 1$$

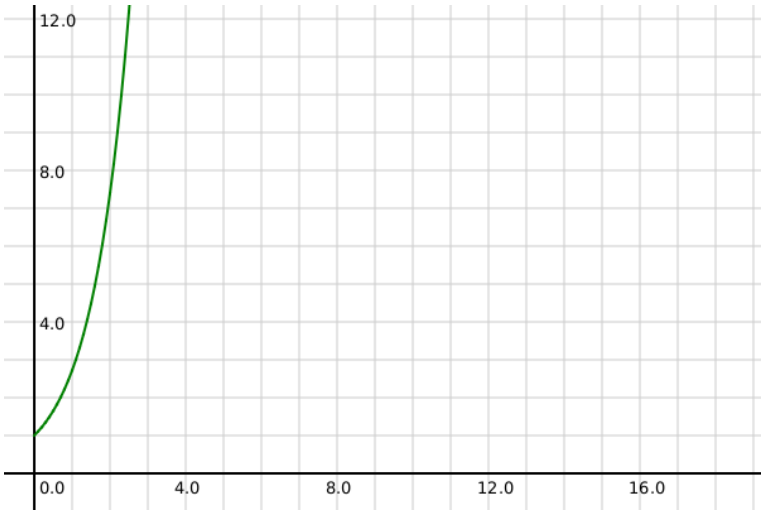
Získáváme tedy diferenciální rovnici:

$$y' = y \quad \& \quad y(0) = 1$$

Tuto rovnici řeší následující program v jazyce *Calc2019*:

```
differential{
    step_size <- 0.1,
    time_max <- 5,
    method <- "euler"
}

y' - y = 0 & y(0) = 1

y
```

Výsledek simulace je na obrázku C.3.



Obrázek C.3: Znázornění výsledku generování funkce $y = e^x$.

C.5 Generování funkcí $sin(x)$

Uvažujme funkci $y = sin(x)$. Vyjádřeme si první a druhou derivaci této funkce:

$$
\begin{aligned}
y &= sin(x) \\
y' &= cos(x) \\
y'' &= -sin(x)
\end{aligned}
$$

Kombinací těchto vztahů můžeme získat diferenciální rovnici:

$$y'' = -y$$

K rovnici doplníme počáteční podmínky:

$$
\begin{aligned}
y(0) &= sin(0) = 0 \\
y'(0) &= cos(0) = 1
\end{aligned}
$$

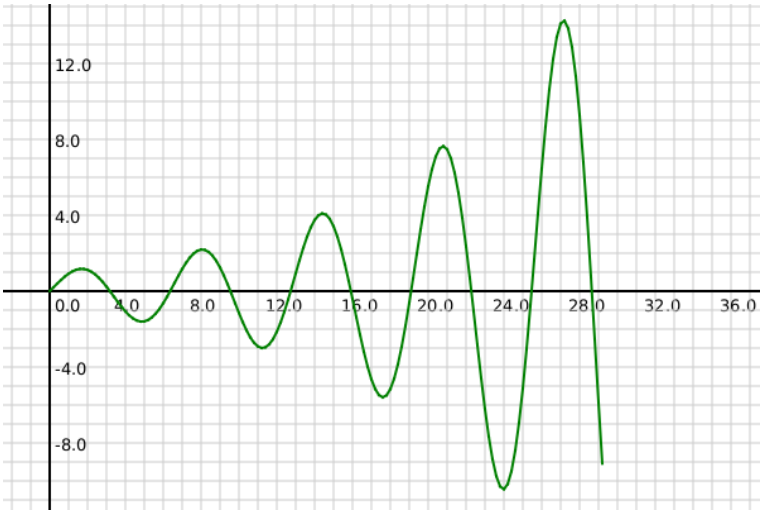
Dostáváme diferenciální rovnici s počátečními podmínkami:

$$y'' + y = 0 \quad \& \quad y(0) = 0, \;\; y'(0) = 1$$

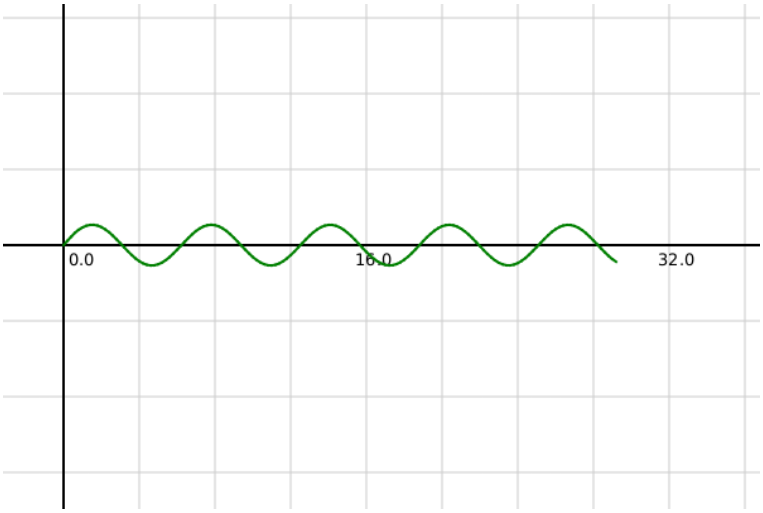
Tuto rovnici řeší následující kód v jazyce *Calc2019*

```
differential {
    step_size <- 0.2,
    time_max <- 29,
    method <- "euler"
}

y'' + y = 0 & y(0) = 0, y'(0) = 1

y
```

Výsledek simulace při použití *Eulerovy metody* je na obrázku C.4. Na obrázku lze vypozorovat velmi rychle narůstající chybu výpočtu. To ukazuje velmi nízkou přesnost *Eulerovy metody*. Na obrázku C.5 je znázorněna simulace stejného problému za pomocí metody *Adams-Bashforth*. Zde vidíme přesnost násobně vyšší.



Obrázek C.4: Znázornění nepřesného řešení generování funkce $y = sin(x)$ pomocí *Eulerovy metody.*



Obrázek C.5: Znázornění přesného řešení generování funkce $y = sin(x)$ pomocí metody *Adams-Bashforth.*

Příloha D

Licence aplikace Matlab

Matlab je aplikace, která je vyvíjena a poskytovaná komerčně. Následující příloha dává přehled o základních druzích licencí [10], ve kterých je aplikace distribuována.

- **Standard** je licence, kterou zvolíme v případě, že chceme software obsluhovat, instalovat a spravovat sami. Tento druh licence je jediný možný způsob, jak program *Matlab* užívat v rámci společnosti za účelem zisku.
- **MATLAB Home** je licence určená výhradně pro osobní a nekomerční použití. V této licenci by měl být užíván k ověřování nápadů a realizaci soukromých projektů u nadšenců. Bývá používána také pro úpravu fotografií. Licence neumožňuje práci ve prospěch jakékoliv organizace. Lze ji použít za účelem zisku, avšak pouze jako jednotlivec.
- **Individuální studentská licence** je licence určená pro soukromý provoz na osobních počítačích studentů. Licence je nepřenosná a je určená pouze pro studentské účely. Nelze ji použít za účelem zisku.
- **PASS licence** Společnost *MathWorks* se rozhodla podpořit vzdělávání na středních a vysokých školách speciálním typem celoškolní multilicence *PASS*. Tento typ licence umožňuje využívat software všem učitelům a žákům dané vzdělávací instituce za výhodných finančních podmínek. Nelze ji použít za účelem zisku.