VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

PROSTŘEDÍ PRO PODPORU MODELŮ FORMÁLNÍCH JAZYKŮ

ENVIRONMENT FOR FORMAL LANGUAGE MODELS SUPPORT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. JAN JUDA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ZBYNĚK KŘIVKA, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

148305

Ústav:	Ústav informačních systémů (UIFS)
Student:	**Juda Jan, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Vývoj aplikací
Název:	**Prostředí pro podporu modelů formálních jazyků**
Kategorie:	Algoritmy a datové struktury
Akademický rok:	2022/23

Zadání:

1. Prozkoumejte existující nástroje pro práci s pokročilými/nestandardními formálními modely (např. řízené gramatiky, skákající modely, Watson-Crick modely apod.). Prostudujte alespoň 3 existující modulární systémy/rámce (např. OSGi) včetně možností využití modulů implementovaných v různých programovacích jazycích.
2. Na základě konzultací s vedoucím proveďte návrh rozšiřitelného modulárního systému sloužícího pro experimentální implementace algoritmů nad pokročilými/nestandardními formálními modely (případně rozšiřte vhodný existující systém).
3. Systém implementujte a jeho modulárnost/rozšiřitelnost demonstrujte implementací alespoň 3 algoritmů pro alespoň 2 zcela odlišné formální modely.
4. Vytvořené moduly otestujte a zhodnoťte modulárnost implementovaného systému.
5. Navrhněte další rozšíření systému.

Literatura:

- A. Meduna: Automata and Languages, Springer, 2000.
- P. Horáček, A. Meduna, M. Tomko: Handbook of Mathematical Models for Languages and Computation. The Institution of Engineering and Technology, 2020. ISBN 978-1-78561-659-4.
- R. S. Hall, K. Pauls, S. McCulloch, D. Savage: OSGi in Action - Creating Modular Applications in Java. Manning, 2011, ISBN 9781933988917.
- S. H. Rodger, T. W. Finley: JFLAP - An Interactive Formal Languages and Automata Package, 2005. Dostupné na: https://www2.cs.duke.edu/csed/jflap/jflapbook/jflapbook2006.pdf [cit. 2021-10-08]

Při obhajobě semestrální části projektu je požadováno:
Student má již splněný SEP.

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

Vedoucí práce:	**Křivka Zbyněk, Ing., Ph.D.**
Vedoucí ústavu:	Kolář Dušan, doc. Dr. Ing.
Datum zadání:	1.11.2022
Termín pro odevzdání:	17.5.2023
Datum schválení:	26.10.2022

Abstrakt

Cílem této práce je vytvořit podpůrné prostředí pro spouštění algoritmů a výpočet typických problémů na pokročilých modelech formálních jazyků. Toto prostředí je realizováno jako modulární aplikace s využitím OSGi rámce, u které autor dbá na snadnou integraci nových uživatelem definovaných modelů formálních jazyků a algoritmů pro tyto modely, a to bez nutnosti úprav či opětovného překladu ostatních částí prostředí včetně jeho jádra.

Pro demonstrování svých možností podpůrné prostředí obsahuje implementaci konečného automatu, bezkontextové gramatiky, n-zásobníkového m-páskového automatu, obecného skákajícího konečného automatu, Watson-Crickova konečného automatu, problému členství a algoritmů obecně schopných řešit problém členství na automatech a gramatikách.

Abstract

The goal of this thesis is to create an environment that supports advanced formal language models including the computation of algorithms for typical problems. This environment is developed as a modular application using OSGi framework, that focuses on easy integration of new user-defined formal language models and algorithms for these models. The modularity provides the possibility to integrate a new user-defined language model and the corresponding algorithms without necessity to recompile the rest of the system. Therefore, a new module can be seamlessly plugged in into the system.

As a demonstration of its capabilities, the environment supports the models of finite automata, context free grammars, n-pushdown m-tape automata, general jumping finite automata, Watson-Crick finite automata, the membership problem and algorithms that are generally able to solve membership problem on automata and grammars.

Klíčová slova

formální jazyky, modely formálních jazyků, automat, gramatika, algoritmy, problém členství, uživatelem definované automaty a gramatiky

Keywords

formal languages, formal language models, automaton, grammar, algorithms, membership problem, user-defined automata and grammars

Citace

JUDA, Jan. *Prostředí pro podporu modelů formálních jazyků*. Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

Prostředí pro podporu modelů formálních jazyků

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doktora Zbyňka Křivky. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jan Juda
16. května 2023

Poděkování

Chtěl bych zde poděkoval panu doktoru Zbyňku Křivkovi za vedení mé diplomové práce. Také musím za podporu poděkovat všem členům své rodiny, a především mojí milované mamince, která mě podporovala ze všech nejvíc, a která se už nemůže dočkat, až z jejího syna bude inženýr.

Dále bych rád poděkoval i všem mým kamarádům a přátelům, a to hlavně za to, že tu pro mě stále jsou i přes to, že jsem na ně v poslední době neměl moc času. Nebojte přátelé, váš Pán Jeskyně se už brzy vrátí!

Nakonec bych chtěl poděkovat i sobě, že jsem se dokopal k tomu tuto práci dokončit a snad i úspěšně odpromovat.

Obsah

Kapitola 1

Úvod

Tato práce pojednává o problematice pokročilých modelů formálních jazyků a o vývoji modulární aplikace poskytující prostředí pro podporu těchto modelů a algoritmů nad těmito modely. Teorie formálních jazyků spadá do oboru teoretické informatiky. Formálním jazykem se myslí množina řetězců nad určitou abecedou, přičemž abeceda je konečná množina znaků.

Výsledky této práce by měly pomoci výzkumným pracovníkům, kteří se zabývají teoretickou informatikou. Konkrétně by aplikace měla sloužit jako možnost otestovat si nově navržené modely formálních jazyků, jejich vlastnosti a algoritmy nad těmito modely.

Dílo je rozděleno do šesti kapitol: Úvod, Potřebná teorie a znalosti, Návrh řešení, Implementace, Zhodnocení práce a Závěr. Kapitolu Úvod právě čtete. V kapitole Potřebná teorie a znalosti provádím shrnutí teorie od základních definic jazyka a jazykových operací až po popis pokročilých modelů. V Návrhu aplikace se věnuji tomu, jak byla aplikace navržena, studuji existující nástroje pro práci s modely formálních jazyků, popisuji, proč jsem se rozhodl implementovat vlastní řešení, a shrnuji informace o technologiích využitých při tvorbě aplikace. Ve čtvrté kapitole Implementace je podrobně vysvětlena a popsána výsledná aplikace a její jednotlivé moduly. Zhodnocení výsledků a modulárnosti výsledného systému provádím v kapitole Zhodnocení práce. V Závěru shrnuji čeho bylo v práci docíleno, v jakém rozsahu práce odpovídá zadání, a také zde nastiňuji různá možná rozšíření této práce, včetně možností vypracování bakalářských či diplomových prací rozšiřující výslednou aplikaci.

Kapitola 2

Potřebná teorie a znalosti

Jelikož je celá tato práce postavena na teorii formálních jazyků, tak mi přijde nezbytné zde v dostatečné míře tuto teorii popsat a nadefinovat všechny použité odborné termíny. V této kapitole se tedy věnuji základům teorie formálních jazyků spolu s důležitými pojmy, probírám vybrané standardní a pokročilé modely formálních jazyků a popisuji problém členství.

2.1 Formálních jazyky a modely

Zde shrnuji základní pojmy z teorie formálních jazyků a uvádím příklady dobře známých modelů formálních jazyků. Při psaní této podkapitoly jsem vycházel především z knih Automata and Computability [4] a Automata and Languages: Theory and Applications [5]. Též jsem se inspiroval učebním textem k teoretické informatice [11].

Teorie formálních jazyků vychází z matematiky a pro její pochopení je nutné se orientovat v teorii množin, grafů a relací. Konkrétně zde čtenář narazí na pojmy jako množiny, množinové operace, relace, funkce, grafy, typy grafů a další jim blízké pojmy. Znalost těchto pojmů a jejich definic budu brát jako nutný základ a jejich vysvětlení se v této práci nebudu věnovat.

Abecedy a řetězce

Než bude možné vysvětlit, co je to jazyk, bude nutné definovat základní pojmy abeceda, symbol a řetězec a několik nezbytných operací, pracujících s těmito pojmy.

Nechť **abeceda** (angl. *alphabet*) je neprázdná konečná množina prvků. Tyto označujeme jako **symboly**. Jako příklad abecedy si můžeme uvést abecedu $\Sigma = \{a, b, c, d, 0, 1, 2\}$. Abecedy obecně mohou být i nekonečné, ale těmi se v této práci nebudu zabývat.

Konečná sekvence symbolů w ve tvaru

$$w = a_1 a_2 a_3 ... a_n$$

kde $a_i \in \Sigma$ a $i = 1, 2, ..., n$, se nazývá **řetězec** (angl. *string*) nad abecedou Σ. Množina všech řetězců nad abecedou Σ se označuje Σ^*. Tato množina obsahuje navíc ještě tzv. **prázdný řetězec**, který značíme ε. Jedna z významných vlastností řetězce je **délka řetězce** w, která se definuje jako $|w| = n$. Prázdný řetězec neobsahuje žádný symbol a jeho délka je tedy 0 ($|\varepsilon| = 0$). Příkladem řetězce nad abecedou Σ mohou být například řetězce *ab*, *abcd*2*acdc*, 101, *cdcdcd* a podobně.

Nad řetězci se dále definuje několik dalších operací, a to:

- **Konkatenace řetězců** (zřetězení): Jsou-li w a x dva řetězce nad abecedou Σ, pak binární operace konkatenace těchto řetězců je řetězec wx (a značí se též wx). Tato operace není komutativní (záleží na pořadí operandů).

  Například uvedu řetězce $w = ab$ a $x = 101$. Jejich konkatenace wx se rovná řetězci $ab101$, ovšem konkatenace xw se rovná řetězci $101ab$.

  o konkatenaci dále platí:

  - Konkatenace $w\varepsilon$ se rovná konkatenaci εw, a to se rovná řetězci w (například $ab\varepsilon = \varepsilon ab = ab$). Jinak řečeno konkatenace má jednotkový prvek ε.
  - Konkatenace je asociativní, tedy platí $w(xy) = (wx)y$.
  - Délka konkatenace wx je součet délek řetězců w a x, tedy $|wx| = |w| + |x|$.

- **Reverzace řetězce**: Unární operace reverzace řetězce $w = a_1, a_2, ..., a_{n-1}, a_n$ se značí w^R a definuje se jako $w^R = a_n, a_{n-1}, ..., a_2, a_1$.

  Například $ab^R = ba$.

- **Mocnina řetězce**: N-tá mocnina řetězce w je unární operace w^n, kde $n \geq 1$, je definovaná následovně: pro $n = 0$: $w^0 = \varepsilon$ a pro $n \geq 1$: $w^n = ww^{n-1}$.

  Například $abc^1 = abc$, $ab^2 = abab$, $012^0 = \varepsilon$, $da^6 = dadadadadada$.

- **Podřetězec**: Jsou-li w a x dva řetězce nad abecedou Σ, pak w je podřetězec x, pokud platí, že existují řetězce y_1 a y_2 nad abecedou Σ takové, že $y_1 w y_2 = x$.

  Pokud navíc platí, že $w \neq \varepsilon$ a $w \neq x$, tak se w nazývá **vlastní podřetězec** řetězce x.

  Například řetězec cab je (vlastní) podřetězec řetězce $aabcabaaac$.

- **Prefix řetězce**: Jsou-li w a x dva řetězce nad abecedou Σ, pak w je prefix x, pokud platí, že existuje řetězec y nad abecedou Σ takový, že $wy = x$.

  Pokud navíc platí, že $w \neq \varepsilon$ a $w \neq x$, tak se w nazývá **vlastní prefix** řetězce x.

  Například řetězec $abba$ je (vlastní) prefix řetězce $abba00122cc$.

- **Sufix řetězce**: Jsou-li w a x dva řetězce nad abecedou Σ, pak w je sufix x, pokud platí, že existuje řetězec y nad abecedou Σ takový, že $wy = x$.

  Pokud navíc platí, že $w \neq \varepsilon$ a $w \neq x$, tak se w nazývá **vlastní sufix** řetězce x.

  Například řetězec 101 je (vlastní) sufix řetězce $abba00122cc$.

- **Počet symbolů** a **v řetězci**: Pro určení počtu výskytů konkrétního symbolu v řetězci se zavádí operace $\#_a(w)$ definovaná jako počet symbolů $a \in \Sigma$ v řetězci $w \in \Sigma^*$. Počet výskytů jakéhokoliv symbolu v řetězci ε je vždy nula.

  Například $\#_c(acdc) = 2$, $\#_2(0010) = 0$, $\#_a(abc) = 1$, $\#_b(\varepsilon) = 0$.

2.1.1 Jazyky a jazykové operace

Formálním jazykem nad abecedou Σ se označuje každá množina L, pro kterou platí $L \subseteq \Sigma^*$. Takový jazyk, stejně jako množinu, je možné popsat výčtem prvků či výrazem, například množiny $L_0 = \varnothing$, $L_1 = \{a, ab, ac, ad\}$, $L_2 = \{w : |w| = 3\}$, $L_3 = \{w : 101 \text{ je sufixem } w\}$, $L_4 = \{a^n b^n c^n : n >= 0\}$ a $L_5 = \Sigma^*$ jsou všechno jazyky nad abecedou Σ.

Můžeme rozlišovat dva typy formálních jazyků, a to **konečné jazyky** a **nekonečné jazyky**. Konečné jazyky jsou takové jazyky, které obsahují konečný počet řetězců. Ostatní jazyky jsou potom nekonečné. Z výše uvedených jazyků jsou jazyky L_0, L_1 a L_2 konečné a jazyky L_3, L_4 a L_5 jsou nekonečné.

Jelikož jsou formální jazyky množiny, tak na nich lze provádět všechny množinové operace jako sjednocení ($\cup$), průnik ($\cup$), rozdíl ($\setminus$), doplněk ($\overline{L}$) a další. Protože je jazyk definován nad abecedou Σ, tak doplněk jazyka se definuje jako rozdíl oproti množině Σ^*, tedy doplněk $\overline{L}$ jazyka L je $\overline{L} = \Sigma^* \setminus L$ (znak $\setminus$ je množinové mínus, tedy rozdíl dvou množin). Kromě těchto operací se na formálních jazycích definují další operace a to:

- **Konkatenace jazyků** (zřetězení): Jsou-li L_1 a L_2 dva jazyky nad abecedou Σ, pak binární operace konkatenace těchto dvou jazyků L_1 a L_2, L_1L_2, se definuje jako

  $$L_1L_2 = \{wx : w \in L_1 \wedge x \in L_2\}$$

  Tato operace podobně jako konkatenace řetězců není komutativní.

  Konkatenace jazyků má jednotkový prvek a to jazyk $\{\varepsilon\}$, tedy $L\{\varepsilon\} = \{\varepsilon\}L = L$. Dále má konkatenace jazyků i nulový prvek a to $\varnothing$, tedy $L\varnothing = \varnothing L = \varnothing$.

  Například $L_1 = \{0, 1\}$ a $L_2 = \{a, b\}$, pak $L_1L_2 = \{0a, 0b, 1a, 1b\}$.

- **Rezervace jazyka**: Unární operace reverzace jazyka L se značí L^R a definuje se jako

  $$L^R = \{w^R : w \in L\}$$

  Například $L = \{abc, acdc\}$, pak $L^R = \{cba, cdca\}$.

- **Mocnina jazyka**: N-tá mocnina jazyka L je unární operace L^n, kde $n \geq 1$, je definovaná následovně

  $$n = 0 : L^0 = \{\varepsilon\}$$
  $$n \geq 1 : L^n = LL^{n-1}$$

  Například $L = \{1, bb\}$, pak $L^0 = \{\varepsilon\}$, $L^1 = L = \{1, bb\}$, $L^2 = \{11, 1bb, bb1, bbbb\}$.

- **Iterace jazyka**: Unární operace iterace jazyka L^* a pozitivní iteraci jazyka L^+ na jazyku L jsou definovány jako

  $$L^* = \bigcup_{n=0}^{\infty} L^n$$
  $$L^+ = \bigcup_{n=1}^{\infty} L^n$$

  Tyto dvě operace mají na jazyku L následující vlastnosti:

  - $L^* = L^+ \cup \{\varepsilon\}$.
  - $L^+ = LL^* = L^*L$.

  Například $L = \{a, b\}$, pak $L^* = \{\varepsilon, a, b, aa, ab, ba, bb, ...\}$ a $L^+ = \{a, b, aa, ab, ba, bb, ...\}$.

Reprezentace jazyků a jejich modely

V tomto oddíle byly ukázány dva způsoby popisu formálního jazyka, a to výčet prvků a matematické výrazy. Tyto způsoby však s rostoucí komplexností popisovaných jazyků naráží na značné problémy. Výčtem prvků, to jest všech řetězců (slov) v jazyce, lze popsat pouze jazyky konečné a u rozsáhlejších z nich to již může být výrazný problém. Pro představu si stačí položit například otázku, kolik slov může mít běžný programovací jazyk. A co třeba jazyk anglický? Výčet prvků jednoduše kromě triviálních příkladů nepřichází v úvahu. **Konečnost** je tedy jeden ze základních požadavků na reprezentaci jak konečných tak nekonečných formálních jazyků.

Matematické výrazy nám sice dovolují reprezentovat i nekonečné jazyky, ovšem snaha o definování struktury komplexního formálního jazyka díky své značné náročnosti s sebou rychle přinese otázku, zda neexistuje lepší způsob na popisování formálních jazyků. Právě této otázce se budu věnovat dále v tomto textu při popisu **modelů formálních jazyků**.

2.1.2 Regulární výrazy a konečné automaty

Regulární výrazy[1] jsou jednoduchou reprezentací formálních jazyků. Jazyky, které značí, se nazývají **regulární jazyky**. Regulární výrazy ovšem nejsou schopny vyjádřit libovolný formální jazyk, proto platí, že množina regulární jazyků je vlastní podmnožinou množiny všech možných formálních jazyků. I přes tento nedostatek jsou ale regulární výrazy velmi rozšířené a hojně prakticky využívané. Mezi praktické použití se řadí například vyhledávání v řetězcích či porovnávání vzorů (angl. pattern matching). Důvod, proč jsou regulární výrazy takto rozšířené navzdory své omezené vyjadřovací síle, je ten, že jejich zápis je vcelku jednoduchý, principiálně snadno pochopitelný a jejich vyjadřovací kapacita je většinou dostatečná pro danou praktickou aplikaci.

Nechť existuje abeceda Σ, pak regulární výrazy nad touto abecedou spolu s jimi reprezentovanými jazyky jsou definovány pomocí následujících pravidel:

- $\varnothing$ je regulární výraz reprezentující prázdnou množinu, tedy prázdný jazyk.
- ε je regulární výraz reprezentující jazyk obsahující pouze řetězec ε, tedy jazyk $\{\varepsilon\}$.
- a, kde platí $a \in \Sigma$, je regulární výraz reprezentující jazyk obsahující pouze řetězec a, tedy jazyk $\{a\}$.
- Pokud r a s jsou regulární výrazy reprezentující jazyky L_r a L_s, potom platí následující:
  - rs je regulární výraz reprezentující konkatenaci jazyků L_r a L_s, jazyk L_{rs} reprezentovaný výrazem rs se tedy rovná $L_{rs} = L_r L_s$.
  - $r + s$ je regulární výraz reprezentující sjednocení jazyků L_r a L_s, jazyk L reprezentovaný tímto výrazem se tedy rovná $L = L_r \cup L_s$.
  - r^* je regulární výraz reprezentující iteraci jazyka L_r, jazyk L reprezentovaný tímto výrazem se tedy rovná $L = L_r^*$.

Složený regulární výraz je pak možné reprezentovat uzávorkováním jeho částí, například výraz $t = (s((r + s)^*))r$. Pro přehlednější zápis se ale definuje následující priorita operací:

$$r* > rs > r + s$$

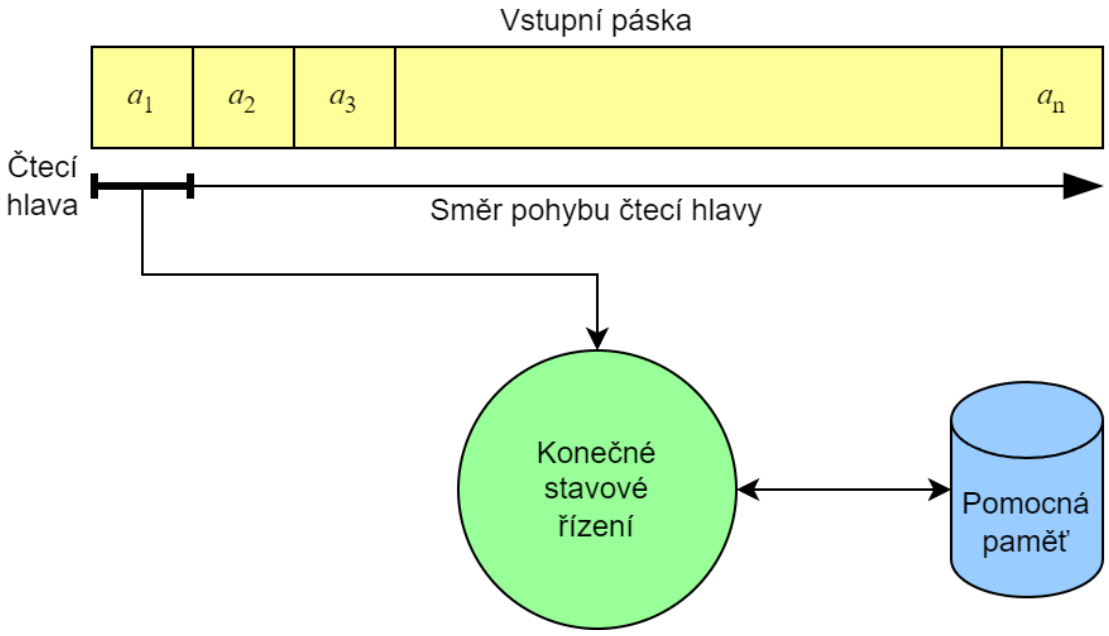
[1] regulární výraz - anglicky *regular expression*, často zkracováno jako „regex“

S takto definovanou prioritou je poté možné výraz t přepsat jako $t = s(r + s)^*r$, přičemž část $r+s$ stále musí být v závorce, jelikož operace * má vyšší prioritu a odstraněním závorky bychom tedy změnili význam. Konkatenace rs se v některé literatuře též označuje pomocí symbolu . jako $r.s$.

V praktickém využití se zápis regulárních výrazů rozšiřuje o řadu dalších pravidel, které slouží ke zkrácení zápisu regulárního výrazu. Typicky se jedná o takzvaný syntaktický cukr (angl. syntactic sugar), což je označení pro rozšíření možností zápisu regulárních výrazů (či v jiném kontextu jakékoliv jiné reprezentace formálního jazyka), které však nerozšiřují vyjadřovací sílu regulárních jazyků (či jiné reprezentace).

Konečný automat

Konečný stavový automat je jednoduchý model založený na konečné množině **stavů** a konečné množině **přechodových pravidel** mezi těmito stavy. Množina formálních jazyků reprezentovatelných konečnými automaty je stejná jako množina jazyků reprezentovaná regulárními výrazy (regulárních jazyků). Pak říkáme, že konečné automaty a regulární výrazy mají **stejnou vyjadřovací sílu**.



Obrázek 2.1: **Schéma obecného automatu**

Obecný automat se skládá z konečného stavového řízení, vstupní pásky, čtecí hlavy a pomocné paměti (viz obrázek 2.1). Automat funguje tak, že čtecí hlava přečte znak ze vstupní pásky a posune se doprava. Stavové řízení pak na základě aktuálního stavu a načteného symbolu provede přechod dle přechodového pravidla a změní svůj stav. Stavové řízení též může využít pomocné paměti pro ukládání informací. Jednotlivé typy automatů se liší podle funkcí čtecí hlavy a pomocné paměti. Čtecí hlava některých automatů může kromě přečtení jednoho znaku a následného posunu doprava například číst více znaků či žádný a různě měnit svoji pozici na čtecí pásce. Co se pomocné paměti týče, tak některé automaty využívají například zásobník, a jiné zase žádnou pomocnou paměť nemají. Tyto funkce se pak promítají ve formátu zápisu přechodových pravidel. Jelikož se automaty snaží

přijmout nebo-li akceptovat řetězec na vstupní pásce, tak jsou označovány jako **akceptační modely**.

Konečný automat vždy čte právě jeden symbol ze vstupní pásky, nemá žádnou pomocnou paměť a jeho čtecí hlava neumožňuje změnu pozice (kromě posunu o jedna doprava při přečtení znaku).

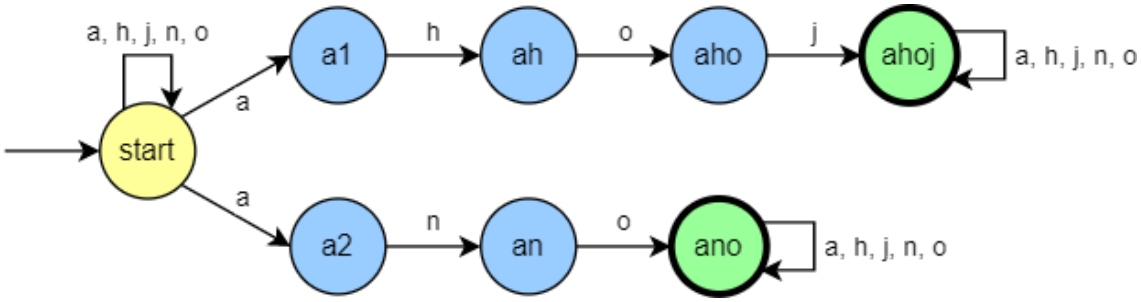
Konečný automat se formálně definuje jako pětice:

$$M = (Q, \Sigma, R, s, F)$$

kde Q je konečná množina stavů, Σ je vstupní abeceda, R je konečná množina přechodových pravidel, $s \in Q$ je počáteční stav a $F \subseteq Q$ je konečná množina koncových stavů. Název stavu může být delší, než jeden symbol.

Množina přechodových pravidel R je definována jako relace z $Q \times \Sigma$ do 2^Q, kde 2^Q symbolizuje potenční množinu množiny Q (to jest množina všech možných podmnožin množiny Q). Pravidla pak lze zapisovat jako $(qa, T) \in R$, kde $q \in Q$, $a \in \Sigma$ a $T \subseteq Q$. Pozorný čtenář si všimne, že jestliže $T \subseteq Q$, pak platí, že $T \in 2^Q$. Pro přehlednější možnost zápisu pravidel se definuje formát $qa \to T$, jenž se často zjednodušuje na zápis $qa \to t$ pro $\forall t \in T$. Tento zápis budu používat i ve zbytku textu. Pravidlo $s0a \to s1, s2, s3$, kde $s0, s1, s2, s3 \in Q$ a $a \in \Sigma$, lze pak zapsat jako tři pravidla $s0a \to s1$, $s0a \to s2$ a $s0a \to s3$.

Konečné automaty lze reprezentovat buď formálním zápisem, čímž se myslí definovat celou pětici (Q, Σ, R, s, F), přičemž pravidla se popíší výčtem prvků či tabulkou, a nebo schématem zobrazujícím stavy a přechody mezi nimi. Na obrázku 2.2 je zobrazen příklad schématu konečného automatu M spolu s vysvětlením zápisu.



Obrázek 2.2: **Schéma konečného automatu M.** Na schématu lze vidět konečný automat M. Každý stav automatu je zobrazen kolečkem, v němž se nachází jméno daného stavu. Šipky mezi stavy reprezentují přechody, tedy například šipka označená písmenem h vedoucí ze stavu `a1` do stavu `ah` značí pravidlo `a1`$h \to$ `ah`. Počáteční stav `start` se značí prázdnou šipkou, jenž nevede ze žádného stavu. Koncové stavy `ahoj` a `ano` se značí tučným ohraničením. Pro zjednodušení orientace jsou stavy barevně rozlišeny, a to tak, že počáteční stav je žlutý, koncové stavy jsou zelené a ostatní stavy jsou modré. Ze schématu lze vyčíst i minimální abecedu konečného automatu M, a to $\Sigma = \{a, h, j, n, o\}$.

Automat M ze schématu 2.2 přijímá všechny řetězce na abecedě $\Sigma = \{a, h, j, n, o\}$, které obsahují buď podřetězec `ahoj`, nebo podřetězec `ano`. Lze si všimnout, že automat M je nedeterministický, jelikož při čtení znaku `a` ve stavu `start` nelze deterministicky rozhodnout, zda automat přejde do stavu `a1`, `a2`, či zda zůstane ve stavu `start`. Tyto tři přechody bychom mohli zapsat pomocí pravidel `start`$a \to$ `a1`, `start`$a \to$ `a2` a `start`$a \to$ `start`, a nebo zjednodušeně jako jedno pravidlo `start`$a \to$ `a1`, `a2`, `start`. Přechodová pravidla automatu

	a	h	j	n	o
start	{start, a1, a2}	{start}	{start}	{start}	{start}
a1	∅	{ah}	∅	∅	∅
ah	∅	∅	∅	∅	{aho}
aho	∅	∅	{ahoj}	∅	∅
ahoj	{ahoj}	{ahoj}	{ahoj}	{ahoj}	{ahoj}
a2	∅	∅	∅	{an}	∅
an	∅	∅	∅	∅	{ano}
ano	{ano}	{ano}	{ano}	{ano}	{ano}

Tabulka 2.1: **Tabulka přechodových pravidel konečného automatu M.**

M by též šla reprezentovat pomocí následující tabulky 2.1. Jazyk přijímaný automatem M lze vyjádřit také pomocí regulárního výrazu $(a+h+j+n+o)^*(ahoj+ano)(a+h+j+n+o)^*$.

Během výpočtu konečného automatu je nutné umět reprezentovat aktuální stav výpočtu, a pro tento účel nám slouží takzvaná **konfigurace** automatu. Konfigurace je řetězec $\kappa \in Q\Sigma^*$ a popisuje stav, ve kterém se automat nachází, a aktuální obsah pásky. Je-li stav $s \in Q$ počáteční stav automatu M a řetězec w je obsah vstupní pásky před zahájením výpočtu, pak se konfigurace sw označuje jako **počáteční konfigurace**. Je-li stav $f \in F$ jedním z koncových stavů automatu M a vstupní páska je aktuálně prázdná (obsahuje řetězec ε), pak se konfigurace $f\varepsilon$ označuje jako přijímající či **akceptující konfigurace**.

Nechť řetězce qax a tx jsou dvě konfigurace konečného automatu M, přičemž $q, t \in Q$, $a \in \Sigma$ a $x \in \Sigma^*$. Dále nechť $qa \to t \in R$ je přechodové pravidlo z R. Potom se říká, že automat M může provést **přechod** z konfigurace qax do konfigurace tx pomocí aplikace pravidla $qa \to t$. Tento přechod se značí symbolem $\vdash$, konkrétně $qax \vdash tx$. Přechod je formálně **relace přechodu** na množině $Q\Sigma^*$. Dále pokud κ je konfigurace automatu M, pak automat M může dosáhnout konfigurace κ provedením nula přechodů, což značíme $\kappa \vdash^0 \kappa$.

Nechť existuje sekvence přechodů $\kappa_{i-1} \vdash \kappa_i$ konečného automatu M ve tvaru

$$\kappa_0 \vdash \kappa_1 \vdash ... \vdash \kappa_n$$

kde $n \geq 1$, $i = 1, 2, ..., n$ a $\kappa_i \in Q\Sigma^*$ je konfigurace automatu M, potom se říká, že konfigurace κ_n je dosažitelná sekvencí přechodů o délce n z konfigurace κ_0, což se značí jako $\kappa_0 \vdash^+ \kappa_n$. Sekvence přechodů připouštějící provedení nula přechodů ($n = 0$) se zapisuje jako $\kappa_0 \vdash^* \kappa_n$ a začíná nultým přechodem $\kappa_0 \vdash^0 \kappa_0$.

Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat, pak formální jazyk L_M přijímaný konečným automatem M se definuje jako $L_M = \{w : w \in \Sigma^*, sw \vdash^* f\varepsilon, f \in F\}$. V zápisu akceptující konfigurace lze symbol ε reprezentující prázdný obsah vstupní pásky vynechat a zapsat tak místo $f\varepsilon$ jen f. Doporučuje se používat zápis $f\varepsilon$ v situacích, kde by nemuselo být jasné, zda f značí pouze stav f či konfiguraci $f\varepsilon$.

Příklad: Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat definovaný schématem 2.2 a nechť se na vstupní pásce nachází řetězec $joano$. Cílem výpočtu bude zjistit, zda řetězec $joano$ patří do jazyka přijímaného automatem M. Pro připomenutí abeceda tohoto automatu je $\Sigma = \{a, h, j, n, o\}$. Výpočet začíná počáteční konfigurací `start`$joano$.

V prvním kroku výpočtu přečte čtecí hlava symbol j. Jediné pravidlo mající pravou stranu `start`j je pravidlo `start`$j \rightarrow$ `start`. Automat tedy aplikuje toto pravidlo a provede přechod `start`$joano \vdash$ `start`$oano$, čímž se výpočet nachází v konfiguraci `start`$oano$.

Ve druhém kroku výpočtu lze obdobně aplikovat pouze pravidlo `start`$o \rightarrow$ `start`, a tudíž proběhne přechod `start`$oano \vdash$ `start`ano.

Zajímavá situace nastává ve třetím kroku výpočtu, kdy čtecí hlava přečte symbol a, zatímco je automat ve stavu `start`. V tuto chvíli je možné provést tři různé přechody. Jelikož je automat M nedeterministický, tak se současně provedou přechody `start`$ano \vdash$ `start`no, `start`$ano \vdash$ `a1`no a `start`$ano \vdash$ `a2`no a výpočet se rozdělí na tři alternativní větve.

Ve čtvrtém kroku automat přečte symbol n. Pro konfiguraci `a1`no neexistuje žádné pravidlo, které by přijalo symbol n, není tedy možné provést žádný přechod a tím pádem se v této větvi automat **zastaví** a tato větev výpočtu končí konfigurací `a1`no neúspěšně. Z konfigurace `start`no lze provést přechod `start`$no \vdash$ `start`o. Nyní už jde jasně vidět, že tato větev výpočtu též skončí neúspěchem, automat však musí výpočet i pro tuto větev dokončit. V poslední větvi s konfigurací `a2`no se provede přechod `a2`$no \vdash$ `an`o.

V posledním kroku výpočtu čte automat symbol o. Ve větvi s konfigurací `start`o se provede přechod `start`$o \vdash$ `start`ε. Přestože automat M přečetl celý obsah vstupní pásky, tak tato větev výpočtu končí neúspěchem, jelikož stav `start` není koncový a tudíž konfigurace `start`ε není přijímající konfigurace. Ve zbývající větvi výpočtu automat provede přechod `an`$o \vdash$ `ano`ε, čímž se nachází v přijímající konfiguraci `ano`ε. Celý výpočet pak končí úspěšně, jelikož automat M dosáhl přijímající konfigurace a řetěz $joano$ tedy patří do jazyka L_M přijímaného automatem M. Při výpočtu se zdá, že automat M správně „uhádne“, do kterého stavu přejít v kroku tři po načtení symbolu a. Tohle zdání je způsobené právě nedeterminismem automatu M.

2.1.3 Gramatiky

Jedním z dobře známých způsobů reprezentace formálních jazyků je takzvaná gramatika. Gramatika je generativní model využívající abecedu neterminálních symbolů a abecedu terminálů symbolů spolu s přepisovacími pravidly k popsání struktury jazyka.

Abeceda terminálních symbolů, zkráceně **terminálů**, gramatiky je shodná s abecedou jazyka, který tato gramatika reprezentuje. Z terminálů se tedy skládají řetězce popisovaného jazyka.

Abeceda neterminálních symbolů, zkráceně **neterminálů**, slouží jako množina zástupných symbolů, přičemž každý neterminál představuje jiný syntaktický celek. Na neterminály se aplikují **přepisovací pravidla**, která přečtou a spotřebují daný neterminál a na jeho pozici vygenerují řetězec terminálů a neterminálů, přičemž cílem je získat řetězce složené čistě z terminálních symbolů – tedy řetězce popisovaného jazyka. Díky této generativní schopnosti, kde gramatika vychází z jednoho **počátečního neterminálu** a generuje delší řetězce, se gramatika označuje jako **generativní model**.

Formálně se gramatika G definuje jako tato čtveřice:

$$G = (N, \Sigma, P, S)$$

kde N je konečná množina neterminálů, Σ je konečná množina terminálů, P je konečná množina přepisovaných pravidel popsaných níže a $S \in N$ je počáteční neterminální symbol gramatiky. Dále platí, že množina neterminálů a množina terminálů jsou disjunktní, tedy $N \cap \Sigma = \varnothing$.

Definice množiny přepisovacích pravidel P se liší dle typu gramatiky. Obecně se dá říct, že přepisovací pravidlo má tvar (α, β), ke α je přepisovaný řetězec na jehož místo je umístěn zapisovaný řetězec β. Pravidlo poté zapisujeme jako $\alpha \to \beta$. Řetězec α, tvořící levou stranu pravidla, musí obsahovat alespoň jeden neterminál, neboť řetězec tvořený pouze terminály nelze přegenerovat. Nejobecnější definice řetězce α je tedy $\alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*$. Pro řetězec β, tvořící pravou stranu pravidla, žádná obdobná podmínka neexistuje, a tím je tedy obecně definován jako $\beta \in (N \cup \Sigma)^*$. Spojením těchto definic lze sestavit nejvíce obecnou definici množiny přepisovacích pravidel P, tak, že množina P je podmnožinu kartézského součinu:

$$P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$$

Aplikace přepisovacího pravidla se nazývá přímé odvození či **přímá derivace** a tvoří mezi řetězci stejnojmennou relaci. Řetězce $\gamma\alpha\delta$ a $\gamma\beta\delta$ jsou v relaci přímé derivace, označované $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$, právě tehdy, když řetězce γ a δ jsou libovolné řetězce z $(N \cup \Sigma)^*$ a v množině pravidel P existuje pravidlo $\alpha \to \beta$, kterým lze přepsat podřetězec α v řetězci $\gamma\alpha\delta$ na řetězec β. Též můžeme říct, že řetězec $\gamma\beta\delta$ byl přímo odvozen (derivován) z řetězce $\gamma\alpha\delta$ s využitím pravidla $\alpha \to \beta$.

Nechť řetězce γ a δ jsou libovolné řetězce z $(N \cup \Sigma)^*$, a pokud existuje sekvence přímých derivací $\alpha_{i-1} \Rightarrow \alpha_i$ ve tvaru

$$\gamma = \alpha_0 \Rightarrow \alpha_1 \Rightarrow ... \Rightarrow \alpha_n = \delta$$

kde $n \geq 1$, $i = 1, 2, ..., n$ a $\alpha_i \in (N \cup \Sigma)^*$, tak řetězce γ a δ jsou v relaci **derivace** označované jako $\gamma \Rightarrow^+ \delta$. Ekvivalentně je možné definovat relaci derivace jakožto tranzitivní uzávěr relace přímé derivace $\Rightarrow$, z čehož vychází i její značení $\Rightarrow^+$. **Délku derivace** definujeme jako počet přímých derivací v derivační sekvenci, a ten je roven číslu n. Dále je možné využít značení $\Rightarrow^n$ pro n-tou mocninu relace přímé derivace $\Rightarrow$.

Každý řetězec $\alpha \in (N \cup \Sigma)^*$, který je derivován z počátečního symbolu, tedy platí $S \Rightarrow^* \alpha$, se označuje jako **větná forma**. Větná forma složená pouze z terminálních symbolů se nazývá **věta**, či věta v gramatice.

Nechť $G = (N, \Sigma, P, S)$ je gramatika G, pak L_G označuje **jazyk generovaný gramatikou** G nad abecedou Σ a generovaný jazyk L_G se definuje jako $L_G = \{w : w \in \Sigma^*, s \Rightarrow^+ w\}$. Nyní následuje pár příkladů.

Příklad 1: Nechť $G = (N, \Sigma, P, S)$ je gramatika G a řetězec $BADC$ je řetězec neterminálů získaný aplikací pravidel z P a $D \to AAAC \in P$ je přepisovací pravidlo z P, pak je možno řetězec $BAAAACC$ odvodit z řetězce $BADC$ aplikací pravidla $D \to AAAC$. Řetězce $BADC$ a $BAAAACC$ jsou v relaci přímé derivace (zapisuje se $BADC \Rightarrow BAAAACC$).

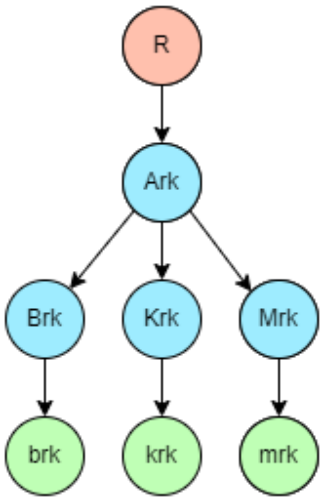
Příklad 2: Nechť $G = (N = \{A, B, K, M, R\}, \Sigma = \{b, k, m, r\}, P, R \in N)$ je gramatika G, přičemž množina pravidel P je

$$P = \{R \to Ark, A \to B, A \to K, A \to M, B \to b, K \to k, M \to m\}$$

pak jazyk L_G generovaný gramatikou G je $L_G = \{brk, krk, mrk\}$. Pro názornost zde rozepíši tento příklad do jednotlivých derivačních kroků. Počáteční symbol gramatiky G je neterminál R. Na tento neterminál lze aplikovat jediné pravidlo, a to $R \to Ark$. Provede se tedy přímá derivace $R \Rightarrow Ark$. Na řetězec Ark lze aplikovat pravidla $A \to B$, $A \to K$ a $A \to M$. Jelikož lze v jednom kroku aplikovat více pravidel, tak lze říci, že tato gramatika G je **nedeterministická**[2]. Provedou se tedy souběžně přímé derivace $Ark \Rightarrow Brk$,

[2]nedeterministický model = takový model, při jehož simulaci v alespoň jednom simulačním kroku nelze jednoznačně určit, jaké pravidlo se má provést, či jaký bude výsledek daného simulačního kroku

$Ark \Rightarrow Krk$ a $Ark \Rightarrow Mrk$. Na každý z dosud vygenerovaných řetězců Brk, Krk a Mrk lze vždy aplikovat jen jedno pravidlo ($B \to b$, $K \to k$ a $M \to m$). Aplikací těchto pravidel vzniknou přímé derivace $Brk \Rightarrow brk$, $Krk \Rightarrow krk$ a $Mrk \Rightarrow mrk$ a řetězce brk, krk a mrk. Žádný z těchto řetězců již neobsahuje žádný neterminál, čímž výpočet končí a jazyk L_G obsahuje právě tyto tři řetězce. Jednotlivé kroky derivace pro tento příklad jsou zobrazeny stromem na obrázku 2.3.



Obrázek 2.3: **Stromové schéma generování řetězců k příkladu č. 2.** Počáteční symbol je znázorněn červeně, řetězce v průběhu derivace modře a výsledné řetězce zeleně.

Pravá lineární gramatika

Nejjednodušším typem gramatiky je takzvaná pravá lineární gramatika. Tato gramatika $G = (N, \Sigma, P, S)$ povoluje pouze přepisovací pravidla ve tvaru

$$a \to wB \quad \text{nebo} \quad A \to w$$

kde $A, B \in N$ a $w \in \Sigma^*$. Název „pravá“ vychází z toho, že jediný neterminál, který se může nacházet na pravé straně přepisovacího pravidla, je vždy úplně vpravo. Tento typ gramatiky zde uvádím především z důvodu, že množina jazyků generovaná pravými lineárními gramatikami je shodná s množinou regulárních jazyků. Pravé lineární gramatiky tedy mají shodnou vyjadřovací sílu jako regulární výrazy či konečné automaty.

K pravým lineárním gramatikám existuje ekvivalent nazývaný levá lineární gramatika, která se od té pravé liší pouze v tom, že místo pravidel ve tvaru $A \to wB$ obsahuje pravidla ve tvaru $A \to Bw$. V těchto gramatikách se neterminál nachází na pravé straně přepisovacího pravidla vždy vlevo.

Jako příklad pravé lineární gramatiky je zde uvedena gramatika, která generuje stejný jazyk, jako konečný automat M ze schématu 2.2. Tato gramatika $G = (N, \Sigma, P, \texttt{start})$,

kde $N = \{\texttt{start}, \texttt{ahoj}, \texttt{ano}\}$, $\Sigma = \{a, h, j, n, o\}$ a množina P obsahuje pravidla

$\texttt{start} \to a\texttt{start}$,	$\texttt{ahoj} \to \varepsilon$,	$\texttt{ano} \to \varepsilon$,
$\texttt{start} \to h\texttt{start}$,	$\texttt{ahoj} \to a\texttt{start}$,	$\texttt{ano} \to a\texttt{start}$,
$\texttt{start} \to j\texttt{start}$,	$\texttt{ahoj} \to h\texttt{start}$,	$\texttt{ano} \to h\texttt{start}$,
$\texttt{start} \to n\texttt{start}$,	$\texttt{ahoj} \to j\texttt{start}$,	$\texttt{ano} \to j\texttt{start}$,
$\texttt{start} \to o\texttt{start}$,	$\texttt{ahoj} \to n\texttt{start}$,	$\texttt{ano} \to n\texttt{start}$,
$\texttt{start} \to ahoj\texttt{ahoj}$,	$\texttt{ahoj} \to o\texttt{start}$,	$\texttt{ano} \to o\texttt{start}$
$\texttt{start} \to ano\texttt{ano}$,		

Bezkontextová gramatika

Bezkontextová gramatika je dalším typem gramatiky. Povolená jsou pouze přepisovací pravidla ve tvaru

$$a \to \alpha \quad \text{kde} \quad a \in N \text{ a } \alpha \in (N \cup \Sigma)^*$$

Množina jazyků přijímaných bezkontextovými gramatikami se nazývá **bezkontextové jazyky**. Bezkontextové jazyky jsou nadmnožinou regulárních jazyků.

Hlavní rozdíl bezkontextové gramatiky oproti pravé lineární je v tom, že přepisovací pravidla bezkontextové gramatiky mohou na své pravé straně mít více než jen jeden neterminál. Při derivování z řetězce, který obsahuje více neterminálů, bezpochyby dojde na otázku, který neterminál z neterminálů v řetězci derivovat dále.

Například gramatika $H = (\{S, A\}, \{a, b, c\}, \{S \to AcAcA, a \to aA, a \to b\}, S)$ v prvním kroku provede derivaci $S \vdash AcAcA$. V dalším kroku přichází otázka, který ze tří neterminálů A derivovat dále. Odpověď, která se nabízí přímo sama, je použití nedeterminismu a rozdělení derivace do tří souběžných větví, přičemž v každé větvi se bude odvozovat právě jeden z neterminálů A. Toto je funkční a spolehlivá metoda, která přijde vhod u komplikovanějších gramatik, ale pro bezkontextové gramatiky existuje jednodušší způsob. Pro vysvětlení tohoto způsobu je nutné nejprve definovat nejlevější a nejpravější derivace.

Nechť $G = (N, \Sigma, P, S)$ je bezkontextová gramatika a řetězce $wA\beta$ a $w\alpha\beta$ jsou v relaci **nejlevější přímé derivace** (angl. leftmost derivation), označované $wA\beta \Rightarrow_{lm} w\alpha\beta$, právě tehdy, když $w \in \Sigma^*$, $A \in N$, $\alpha, \beta \in (N \cup \Sigma)^*$ a v množině pravidel P existuje pravidlo $A \to \alpha$. Derivace se nazývá nejlevější proto, že se vždy derivuje nejlevější neterminál v řetězci. Obdobně je možné definovat **nejpravější přímou derivaci** (angl. rightmost derivation), kde se jen prohodí podřetězce w a β v definici, tedy $\beta Aw \Rightarrow_{rm} \beta\alpha w$. Při použití nejpravější derivace se pak derivuje neterminál, který se v řetězci nachází nejvíce v pravo.

Analogicky, jako je v oddílu 2.1.3 definovaná relace derivace $\Rightarrow^+$, se definuje relace nejlevější derivace $\Rightarrow^+_{lm}$ a nejpravější derivace $\Rightarrow^+_{rm}$ jen s užitím relací nejlevější přímé derivace $\Rightarrow_{lm}$ a nejpravější přímé derivace $\Rightarrow_{rm}$ místo relace přímé derivace $\Rightarrow$.

Nechť $G = (N, \Sigma, P, S)$ je bezkontextová gramatika, pak jazyky $L_{G1} = \{w : w \in \Sigma^*, s \Rightarrow^+ w\}$, $L_{G2} = \{w : w \in \Sigma^*, s \Rightarrow^+_{lm} w\}$ a $L_{G3} = \{w : w \in \Sigma^*, s \Rightarrow^+_{rm} w\}$ jsou totožné. Nejlevější a nejpravější derivace tedy nijak nesnižují vyjadřovací sílu bezkontextových gramatik. Díky tomuto faktu si stačí pro generování řetězců zvolit nejpravější či nejlevější derivaci a při odvozování v situacích, kdy má jeden řetězec více neterminálů (viz příklad s výše popsanou gramatikou H), je již jasně dané pravidlo, který z neterminálů v řetězci derivovat.

Běžným nástroj pro zobrazení posloupnosti derivací je takzvaný **derivační strom**. Nechť $G = (N, \Sigma, P, S)$ je bezkontextová gramatika, řetězec κ je z množiny $(N \cup \Sigma)^*$

a $S = \beta_0 \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow ... \Rightarrow \beta_k = \kappa$ je derivace v gramatice G. Derivační strom této derivace je potom takový strom, který splňuje následující podmínky:

- Uzly derivačního stromy jsou označeny symboly z $(N \cup \Sigma)$, přičemž kořenový uzel je označen počátečním neterminálem S.
- Přímá derivace $\beta_{i-1} \Rightarrow \beta_i$, $i = 0, 1, ..., k$, kde
  - $\beta_{i-1} = \gamma A \delta$, $\gamma, \delta \in (N \cup \Sigma)^*, a \in N$
  - $\beta_i = \gamma \alpha \delta$
  - $(A \to \alpha) \in P$, $\alpha = X_1 X_2 ... X_n$

  je znázorněna právě n hranami (A, X_j) vedoucími z uzlu A do uzlů X_j, kde $j = 1, 2, ..., n$. Tyto hrany musí být seřazeny zleva doprava v pořadí $(A, X_1), (A, X_2), ...,$ (A, X_n).
- Symboly listových uzlů derivačního stromu vytváří zleva doprava řetězec κ.

Nechť $G = (N, \Sigma, P, S)$ je bezkontextová gramatika, existuje-li řetězec $w \in L_G$, jehož derivaci lze znázornit více než jedním derivačním stromem, pak se gramatika G označuje jako **nejednoznačná**. Pokud žádný takový řetězec w neexistuje, potom je gramatika G označena za **jednoznačnou**. Dále bezkontextový jazyk L označujeme za vnitřně nejednoznačný, pokud neexistuje žádná jednoznačná bezkontextová gramatika G, která by tento jazyk L generovala.

Příklad: Nechť $G = (\{S, field, id, rval, bool, int, string\}, \{':', '\{', '\}', ',', 'true', 'false'\}, P, S)$ je bezkontextová gramatika s přechodovými pravidly P:

$$
\begin{array}{lll}
s \to \{field\}, & field \to \{id : rval\}, & rval \to \{string\}, \\
field \to \{field, field\}, & rval \to \{bool\}, & bool \to \{true\}, \\
field \to \{field, field\}, & rval \to \{int\}, & bool \to \{false\}
\end{array}
$$

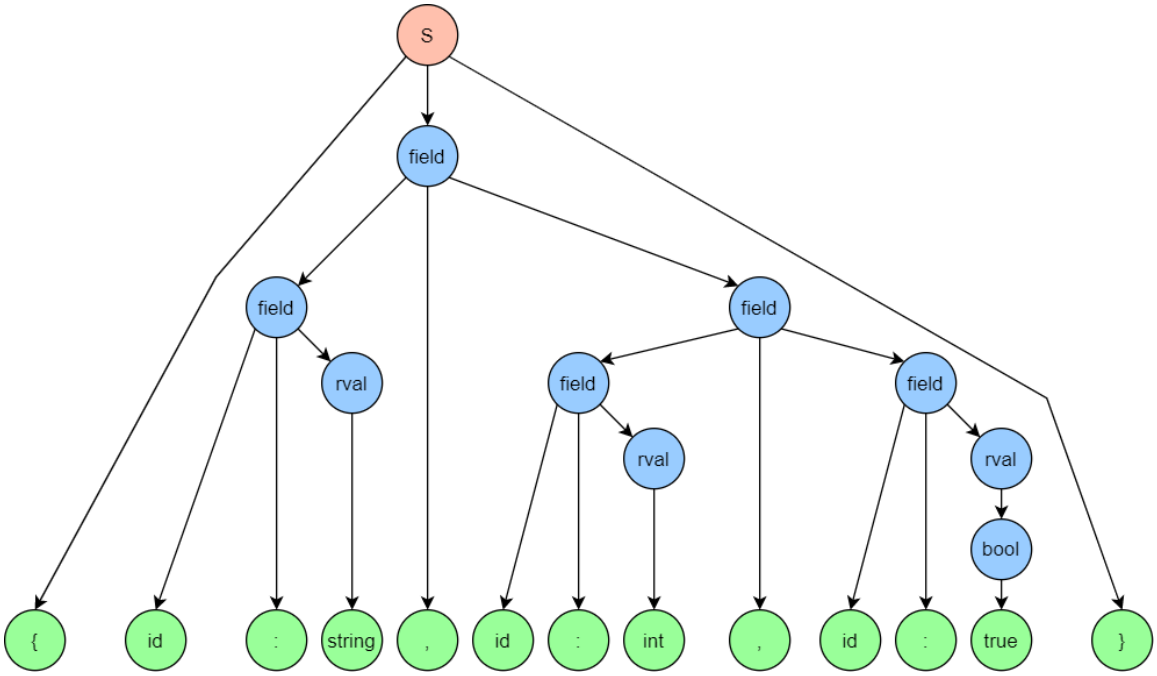
Derivace řetězce $\alpha =$ `{id:string,id:int,id:true}`, $\alpha \in (N \cup \Sigma)^*$, jde znázornit derivačním stromem na obrázku 2.4.

Chomského hierarchie a další typy gramatik

Chomského hierarchie jazyků, či také Chomského klasifikace jazyků, rozděluje množinu formálních jazyků do čtyř množin, které se označují jako typ 0, typ 1, typ 2 a typ 3. Mezi těmito typy jazyků platí vztah Typ 3 $\subset$ Typ 2 $\subset$ Typ 1 $\subset$ Typ 0. Jedná se o vlastní podmnožiny, proto je vždy možné najít jazyk typu 0, který není jazykem typu 1 a podobně.

Každému z Chomského typů formálních jazyků odpovídá gramatika, která tento jazyk generuje. Jazyky typu 3 jsou generovány pravými lineárními gramatikami, které jsou popsány výše v tomto oddílu. V textu jsou pravé lineární gramatiky následovány bezkontextovými gramatikami, které generují jazyky typu 2.

Gramatiky generující jazyky typu 1 se nazývají **kontextové gramatiky**. Pravidla kontextové gramatiky mají podobu $\alpha A \beta \to \alpha \gamma \beta$, kde $\alpha, \beta \in (N \cup \Sigma)^*$, $A \in N$ a $\gamma \in (N \cup \Sigma)^+$. Řetězcům α a β se říká kontext neterminálu A, z čehož je odvozen název kontextových gramatik. Množina pravidel P navíc může obsahovat i pravidlo ve tvaru $S \to \varepsilon$, pokud se počáteční neterminální symbol S nenachází na výsledné (pravé) straně žádného jiného



Obrázek 2.4: **Derivační strom řetězce `{id:string,id:int,id:true}`.** Počáteční symbol je znázorněn červeně, neterminály v průběhu derivace modře a listové symboly zeleně.

pravidla v P. Toto pravidlo dovoluje, aby řetězec ε mohl být součástí jazyka kontextové gramatiky.

Jazykům typu 0 odpovídají **neomezené gramatiky**, nebo též gramatiky typu 0. Název neomezené gramatiky vychází z toho, že jejich pravidla mají stejnou podobu $\alpha \to \beta$ jako pravidla v obecné definici gramatiky, kde $\alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*$ a $\beta \in (N \cup \Sigma)^*$, a nejsou tedy žádným způsobem omezena. Jazyky typu 0 obsahují všechny jazyky přijímané Turingovými stroji. Vyjadřovací síla neomezené gramatiky je tedy ekvivalentní k vyjadřovací síle Turingova stroje. Když je nějaký model silou ekvivalentní Turingově stroji, tak se říká, že je daný model **turingovsky úplný**.

Tuto část teorie formálních jazyků zde uvádím proto, že jsem při návrhu výsledné aplikace bral v potaz, aby bylo v aplikaci možné snadno definovat jakoukoliv ze standardních gramatik.

2.1.4 Zásobníkový automat

Zásobníkový automat je model s vyjadřovací silou ekvivalentní vyjadřovací síle bezkontextových gramatik. Zásobníkové automaty tedy přijímají bezkontextové jazyky.

Zásobníkový automat má velmi blízko ke konečnému automatu, který rozšiřuje použitím pomocné paměti (viz obrázek se schématem obecného automatu 2.1) v podobě právě jednoho zásobníku. Zásobníkový automat se formálně definuje jako sedmice

$$M = (Q, \Sigma, \Gamma, R, s, S, F)$$

kde Q je konečná množina stavů, Σ je vstupní abeceda, Γ je zásobníková abeceda, R je konečná množina přechodových pravidel, $s \in Q$ je počáteční stav, $S \in \Gamma$ je počáteční symbol na zásobníku a $F \subseteq Q$ je konečná množina koncových stavů.

Množina přechodových pravidel R je definována jako relace z $\Gamma \times Q \times (\Sigma \cup \{\varepsilon\})$ do $\Gamma^* \times Q$. Pravidla jsou pak zapisována ve tvaru $Aqa \to wt$, kde $A \in \Gamma$, $q, t \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ a $w \in \Gamma^*$. Díky použití této definice, kde $a \in \Sigma \cup \{\varepsilon\}$, lze vidět, že zásobníkový automat na rozdíl od konečného automatu povoluje i takzvané **epsilon přechody**. Epsilon přechod je takový přechod, u kterého automat ze vstupní pásky nečte žádný znak, respektive čte automat prázdný řetězec ε.

Konfigurace zásobníkového automatu je řetězec $\kappa \in \Gamma^* Q \Sigma^*$. Počáteční konfiguraci zásobníkového automatu se definuje jako konfigurace Ssw, kde $w \in \Sigma^*$ je řetězec na vstupní pásce před zahájením výpočtu.

Nechť řetězce $wAqax$ a wtx jsou dvě konfigurace zásobníkového automatu M, přičemž $w \in \Gamma^*$, $A \in \Gamma$, $q, t \in Q$, $a \in \Sigma$ a $x \in \Sigma^*$. Dále nechť $Aqa \to t \in R$ je přechodové pravidlo z R. Potom se říká, že automat M může provést **přechod** z konfigurace $wAqax$ do konfigurace wtx pomocí aplikace pravidla $Aqa \to t$. Tento přechod se značí symbolem $\vdash$, konkrétně $wAqax \vdash wtx$. Přechod je formálně **relace přechodu** na množině $\Gamma^* Q \Sigma^*$. Sekvence přechodů $\vdash^+$ a $\vdash^*$ se definuje po řadě jako tranzitivní uzávěr relace přechodu a tranzitivní reflexivní uzávěr relace přechodu.

Akceptující konfiguraci zapsanou jako $xf\varepsilon$ či xf a jazyk $L(M) = \{w | Ssw \vdash^* xf\}$ přijímaný zásobníkovým automatem M lze definovat jedním ze následujících tří způsobů pomocí toho, jak se definuje stav f a řetězec x:

1. pokud $f \in F$ a $x \in \Gamma*$, pak automat M přijímá jazyk $L(M)$ přechodem do koncového stavu
2. pokud $f \in Q$ a $x = \varepsilon$, pak automat M přijímá jazyk $L(M)$ vyprázdněním zásobníku
3. pokud $f \in F$ a $x = \varepsilon$, pak automat M přijímá jazyk $L(M)$ přechodem do koncového stavu a vyprázdněním zásobníku

Zásobníkové automaty přijímající jazyky těmito třemi způsoby jsou mezi sebou navzájem ekvivalentní, protože lze převést zásobníkový automat přijímající jedním způsobem na ekvivalentní zásobníkový automat přijímající jiným z těchto způsobů.

2.2 Pokročilé formální modely

V této podkapitole se věnuji popisu teorie o implementovaných pokročilých modelech formálních jazyků, a to konkrétně Watson-Crickově konečném automatu a obecnému skákajícímu konečnému automatu.

2.2.1 Watson-Crickův konečný automat

Watson-Crickův konečný automat je velmi podobný dvoupáskovému konečnému automatu, ale s tím rozdílem, že navíc obsahuje komplementární relaci ρ. Watson-Crickův automat byl navržen pro zpracování dvouvláknových řetězců DNA. Každé ze dvou vláken DNA je čteno jednou čtecí hlavou, proto je Watson-Crickův automat navržen se dvěma páskami. Důležitou vlastností Watson-Crickova automatu je to, že symboly na odpovídajících pozicích na vstupních páskách jsou vzájemně komplementární dle relace ρ podobně, jako jsou páry nucleotidů DNA komplementární dle Watson-Crickovy komplementarity. V tomto oddílu pojednávajícím o Watson-Crickově konečném automatu čerpám informace z prací [1] a [3].

Watson-Crickův konečný automat je formálně definován jako šestice:

$$M = (Q, \Sigma, \rho, R, s, F)$$

kde Q je konečná množina stavů, Σ je vstupní abeceda, $\rho \subseteq \Sigma \times \Sigma$ je komplementární **symetrická** relace, R je konečná množina přechodových pravidel, $s \in Q$ je počáteční stav a $F \subseteq Q$ je konečná množina koncových stavů. To, že je relace ρ symetrická, znamená $\forall a, b \in \Sigma,\ (a, b) \in \rho \implies (b, a) \in \rho$. Důležité je, že množina přechodových pravidel musí korespondovat s komplementární relací ρ. Takto definovaný Watson-Crickův konečný automat je obdobně jako standardní konečný automat obecně nedeterministický.

Pravidla z přechodové množiny R se zapisují jako $q(w_1, w_2) \to t$, kde $q, t \in Q$, $w_1, w_2 \in \Sigma^*$ a $T \subseteq Q$. Tento zápis budu v textu i nadále používat. Alternativně je možné se setkat i se zápisem pravidel ve tvaru $q\binom{w_1}{w_2} \to t$, což více odpovídá zápisu DNA. Formálně lze množinu přechodových pravidel R definovat jako relaci z $Q \times \Sigma^* \times \Sigma^*$ do Q.

Konfigurace Watson-Crickova konečného automatu je řetězec $\kappa \in Q\Sigma^*\Sigma^*$ zapisovaný ve formátu $q(u_1, u_2)$ kde $q \in Q$ a $u_1, u_2 \in \Sigma^*$. Použití závorek v zápisu je nutné pro definovaní hranice mezi dvěma řetězci. Je-li stav $s \in Q$ počáteční stav Watson-Crickova konečného automatu, řetězec w_1 je obsahem první vstupní pásky před zahájením výpočtu a řetězec w_2 je obsahem druhé vstupní pásky před zahájením výpočtu, pak se konfigurace $s(w_1, w_2)$ označuje jako **počáteční konfigurace** značená jako κ_0.

Nechť řetězce $q(u_1x_1, u_2x_2)$ a $t(x_1, x_2)$ jsou dvě konfigurace Watson-Crickova konečného automatu M, přičemž $q, t \in Q$ a $u_1, u_2, x_1, x_2 \in \Sigma^*$. Dále nechť $q(u_1, u_2) \to t \in R$ je přechodové pravidlo z R. Potom se říká, že automat M může provést **přechod** z konfigurace $q(u_1x_1, u_2x_2)$ do konfigurace $t(x_1, x_2)$ pomocí aplikace pravidla $q(u_1, u_2) \to t$. Tento přechod se značí symbolem $\vdash$, konkrétně $q(u_1x_1, u_2x_2) \vdash t(x_1, x_2)$. Sekvence přechodů a dosažitelnost konfigurace pak definujeme obdobně jako u konečného automatu.

Definují se množiny $\left[\begin{smallmatrix}\Sigma\\ \Sigma\end{smallmatrix}\right]_\rho = \left\{\left[\begin{smallmatrix}a\\ b\end{smallmatrix}\right] \mid a, b \in \Sigma, (a, b) \in \rho\right\}$ a $WK_\rho(\Sigma) = \left[\begin{smallmatrix}\Sigma\\ \Sigma\end{smallmatrix}\right]_\rho^*$. Množina $WK_\rho(\Sigma)$ se nazývá Watson-Crickova doména asociovaná s abecedou Σ a relací ρ. Dále se definuje zápis $\left[\begin{smallmatrix}w_1\\ w_2\end{smallmatrix}\right] = \left[\begin{smallmatrix}a_1\\ b_1\end{smallmatrix}\right]\left[\begin{smallmatrix}a_2\\ b_2\end{smallmatrix}\right]\ldots\left[\begin{smallmatrix}a_n\\ b_n\end{smallmatrix}\right] \in WK_\rho(\Sigma)$, kde $w_1 = a_1a_2\ldots a_n$ a $w_2 = b_1b_2\ldots b_n$. Tento zápis implikuje, že řetězce w_1 a w_2 mají stejnou délkou a jsou společně propojeny komplementární relací ρ tak, že platí $(a_i, b_i) \in \rho$ pro $1 \leq i \leq n$.

Je-li stav $f \in F$ jedním z koncových stavů automatu M, obě vstupní pásky jsou aktuálně prázdné (obsahují řetězec ε), konfigurace $f(\varepsilon, \varepsilon)$ je dosažitelná z počáteční konfigurace κ_0 a pro řetězce w_1 a w_2 z počáteční konfigurace κ_0 platí $\left[\begin{smallmatrix}w_1\\ w_2\end{smallmatrix}\right] \in WK_\rho(\Sigma)$, pak se konfigurace $f(\varepsilon, \varepsilon)$ označuje jako **akceptující konfigurace**.

Nechť $M = (Q, \Sigma, \rho, R, s, F)$ je Watson-Crickův konečný automat, pak formální jazyk L_M přijímaný automatem M se definuje jako $L_M = \{w_1 : w_1, w_2 \in \Sigma^*, s(w_1, w_2) \vdash^* f(\varepsilon, \varepsilon), f \in F, \left[\begin{smallmatrix}w_1\\ w_2\end{smallmatrix}\right] \in WK_\rho(\Sigma)$. Dle definice je vidět, že automat akceptuje pouze řetězec w_1. Řetězec w_2 není součástí jazyka, a tudíž není akceptovaný. Jeho účel je proto pouze pomocný.

2.2.2 Obecný skákající konečný automat

Skákající konečný automat je velmi podobný standardnímu konečnému automatu, liší se jen v tom, že s každým provedeným čtením může skočit na libovolnou pozici na vstupní pásce. Skok tedy znamená přesun hlavy na jinou pozici na pásce. Na kterou pozici bude skok proveden, je zcela nedeterministické rozhodnutí, takže tento model je už z principu velice nedeterministický. Počáteční pozice čtecí hlavy na pásce je též volena nedeterministicky, což vyplývá z definice relace skoku níže. Důležité je zdůraznit i to, že **každý symbol na vstupní pásce může být přečten maximálně jedenkrát**. V podstatě se dá říct že

symbol ze vstupní pásky po přečtení zmizí. Obecnějším typem skákající automatu s vyšší výpočetní silou, než má skákající konečný automat, je **obecný skákající konečný automat**. Obecný skákající konečný automat na rozdíl od skákajícího konečného automatu a standardního konečného automatu může v jenom kroku číst libovolný počet symbolů ze vstupní pásky, tedy nula až neomezeně mnoho. V tomto oddíle vycházím z díla [6].

Obecný skákající konečný automat se formálně definuje jako pětice

$$M = (Q, \Sigma, R, s, F)$$

kde Q je konečná množina stavů, Σ je vstupní abeceda, R je konečná množina přechodových pravidel, $s \in Q$ je počáteční stav a $F \subseteq Q$ je konečná množina koncových stavů.

Množina přechodových pravidel R je definována jako relace z $Q \times \Sigma^*$ do Q. Pravidla se zapisují ve tvaru $py \to q$, kde $p, q \in Q$ a $y \in \Sigma^*$. Konfigurace obecného skákajícího konečného automatu je řetězec $\kappa \in \Sigma^* Q \Sigma^*$. Pro formální definování skoků se používá takzvaná **relace skoku** na $\Sigma^* Q \Sigma^*$ značená symbolem $\curvearrowright$. Nechť $x, z, x', z' \in \Sigma^*$ a $py \to q \in R$ tak, že $xz = x'z'$, pak obecný skákající konečný automat může provést skok z $xpyz$ do $x'qz'$, který se zapisuje jako $xpyz \curvearrowright x'qz'$. Podobně jako u předchozích modelů značíme tranzitivní uzávěr relace skoku jako $\curvearrowright^+$ a tranzitivní reflexivní uzávěr relace skoku jako $\curvearrowright^*$.

Konfiguraci $\varepsilon f \varepsilon$, kde $f \in F$, označujeme za akceptující konfiguraci. Tuto konfiguraci je též možné zapsat čistě jako f. Obecný skákající konečný automat se nachází v akceptující konfiguraci, když přečetl všechny symboly na vstupní pásce a nachází se v koncovém stavu. Jazyk $L(M)$ přijímaný obecným skákajícím konečným automat M se definuje jako

$$L(M) = \{uv | u, v \in \Sigma^*, usv \curvearrowright^* \ \ f, f \in F\}$$

V počáteční konfiguraci usv pozice stavu s v řetězci uv dle definice relace skoku reprezentuje pozici čtecí hlavy na vstupní pásce s řetězcem uv. Nechť $|uv|$ je délka řetězce uv, pak se čtecí hlava na vstupní pásce s řetězcem uv může nacházet na $|uv| + 1$ různých pozicích. Z definice přijímaného jazyka $L(M)$ vyplývá, že pokud $uv \in L(M)$, pak alespoň jedna z těchto pozic je taková, že platí $usv \curvearrowright^* \ \ f$. Z toho lze usoudit, že počáteční pozice čtecí hlavy je zvolena nedeterministicky.

2.3 Problém členství

Problém členství (anglicky *Membership Problem*) zodpovídá otázku, zda řetězec $w \in \Sigma^*$ náleží do jazyka $L(M)$ na abecedě Σ. Řešením problému členství je odpověď ano či ne, která může být případně obohacená o bližší zdůvodnění. Obecně tento problém není rozhodnutelný, ale je jen **částečně rozhodnutelný**. To znamená, že pokud je odpověď kladná, tak je možné ji získat, ale pokud odpověď kladná není, tak ji nemusí být možné získat. V této podkapitole čerpám informace z [11].

Problém členství se typicky řeší simulací na Turingově stroji, jenž v podobě, které rozumí, dostane na vstup zakódovaný popis Turingova stroje M přijímajícího jazyk $L(M)$ a řetězec w. Pro účely této práce bude výpočet problému provádět podpůrné prostředí. Modelem M, který akceptuje jazyk $L(M)$ a má vstupní abecedu Σ, může být instance libovolného modelu implementovaného v mé aplikaci. Detaily o tom, jak je problém členství v aplikaci implementován, je možné najít v kapitole Implementace v podkapitole 4.5.

Kapitola 3

Návrh řešení

V této kapitole jsem se zaměřil na návrh řešení, jímž je modulární aplikace poskytující prostředí pro spouštění algoritmů a výpočet typických problémů na pokročilých a nestandardních modelech formálních jazyků.

3.1 Existující nástroje pro práci s modely formálních jazyků

Nejlepší nástroj na pro práci s modely formálních jazyků, který jsem během svého průzkumu našel, je **JFLAP**[1]. Ostatní existující nástroje, se kterými jsem se setkal, podporují menší množinu modelů, například se specializují jen na automaty, jen na gramatiky, či čistě na standardní modely bez rozumné možnosti rozšíření.

JFLAP obsahuje širokou řadu standardních modelů formálních jazyků a algoritmů k nim. Tento nástroj je šířen pod licencí *Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License*[2], a tudíž by jeho rozšíření bylo legálně možné. JFLAP používá architekturu zásuvných modulů (anglicky plugins), což by splňovalo požadavek na modulární řešení.

Implementace nástroje JFLAP bere v potaz především standardní modely formálních jazyků. Rozšíření tohoto nástroje pro práci s nestandardními modely by sice bylo možné, nejsem si však jist, zda by takovéto řešení bylo schopné podpořit dostatečně širokou škálu nestandardních modelů a zda by čas věnovaný snaze upravit tento nástroj nebyl lépe využit implementací vlastního řešení. Jedním z problému u rozšiřování nástroje JFLAP je to, že zavedená rozhraní by nemusela být dostatečná pro všechny možné pokročilé modely, a to ať už pro ty, co jsem ve výsledné aplikaci implementoval, tak i pro ty, které mohou být součástí některého z budoucích rozšíření aplikace. Dalším problémem by mohl být způsob přidávání modulů s novými modely a algoritmy, u kterého nelze vyloučit nutnost zásahu do jádra systému, a to třeba právě kvůli úpravě rozhraní. Uznávám, že rozšíření existujícího programu může ušetřit celkový čas, pokud není nutné provádět naprosto zásadní změny v návrhu programu. Na serveru GitHub[3] se však už dá najít celá řada rozšíření nástroje JFLAP, nepřijde mi tedy příliš originální zabývat se tvorbou dalšího takového rozšíření, a to je i jeden z důvodů, proč jsem se rozhodl navrhnout vlastní řešení. Dalším důvodem pro vlastní řešení je výrazně vyšší flexibilita, která mi dovoluje navrhnout systém už od základu s myšlenkou pokročilých a nestandardních modelů. Také dobrým argumentem

[1] JFLAP - dostupný na webové stránce https://www.jflap.org/

[2] Detaily licence dostupné na adrese https://creativecommons.org/licenses/by-nc-sa/2.5/

[3] GitHub - poskytovatel hostování repozitářů pro verzovací program git - dostupný na webové stránce https://github.com/

pro nové řešení je předejití nutnosti výše zmíněných rozsáhlých úprav nástroje JFLAP. Vlastní modulární návrh zaměřující se na snadnou rozšiřitelnost navíc poskytuje i možnost výměny a implementace různých uživatelských rozhraní (viz oddíl 3.3.3). Po zvážení všech argumentů pro a proti jsem se rozhodl k návrhu a implementaci vlastní modulární aplikace.

Při zhodnocení vhodnosti nástroje JFLAP jako základ mého řešení jsem vycházel z webových stránek tohoto nástroje [9] a z vlastních experimentů a zkušeností s tímto nástrojem.

3.2 Návrh vlastní aplikace

Základní myšlenka je taková, že aplikace bude sloužit jako rámcové řešení, které poskytuje uživateli modulární rozšiřitelné prostředí složené z abstraktních tříd s vhodnými rozhraními pro snadné rozšíření a specifikování jejich vlastností, předpřipravených funkcí, dalších pomocných tříd (například pro načítání instancí modelů, či vstupů algoritmů), vzorových modelů, problému členství (viz podkapitola 2.3), případně dalších problémů, které ve formálních jazycích můžeme řešit, a vzorových algoritmů implementujících řešení těchto problémů. Uživatel může využít toto prostředí pro snadné definování vlastní **komponenty** (viz podkapitola 3.3) neboli modelu formálních jazyků, algoritmu pro tyto modely, nebo celého problému v oblasti modelů formálních jazyků. Následně si uživatel může otestovat vlastnosti své nové komponenty s využitím vzorových či dalších vlastních komponent.

V případě, že by jakákoliv vzorová komponenta nebyla schopna splnit všechny vlastnosti a požadavky uživatele, si bude moci uživatel rychle implementovat svoji variantu této komponenty vhodným rozšířením anebo přepsáním některého bodu rozšíření (typicky metody) dané komponenty. Upravená komponenta bude díky rozhraním kompatibilní se všemi ostatními funkcemi tohoto podpůrného prostředí.

Program funguje tak, že si uživatel vybere požadovaný problém k vyřešení (např. problém členství), třídu modelu (či více tříd modelů, například při problému konverze gramatiky na odpovídající automat), na které se má problém řešit (např. konečný automat), soubory s popisy instancí modelů (např. konečné automaty `A` a `B` se stavy `a`, `b`, `c`... a pravidly `Aa->B`...), algoritmus, který zvládne vypočítat zvolený problém na zvoleném modelu, a soubory se vstupy pro vybraný problém. Program poté spustí vybraný problém, který načte jednotlivé instance zvoleného modelu ze souborů s jejich popisy a spustí výpočty vybraného algoritmu na načtených instancích, přičemž bude paralelně spuštěn samostatný výpočet pro každou instanci modelu s každou množinou vstupů, které tento algoritmus používá. Po ukončení průběhu všech spuštěných výpočtů problém sestaví výsledek celého řešení z dílčích výsledků výpočtů spuštěných algoritmů a program následně vypíše, uloží či jinak zobrazí tento finální výsledek.

3.3 Struktura aplikace

Základem řešení je modulární aplikace využívající existující modulární systém či rámcové řešení (viz podkapitola 3.4).

Program obsahuje tři typy modulů, a to hlavní modul, moduly obsahující komponenty a moduly s aplikačním či uživatelským rozhraním. Uživatel si může vytvořit libovolný počet nových komponentních modulů a modulů s rozhraním. Také je možné deaktivovat či z prostředí kompletně vyjmout jakýkoliv z existujících modulů, kromě hlavního modulu, jenž je nezbytný pro chod aplikace a všechny ostatní moduly na něm závisí.

3.3.1 Komponentní moduly a komponenty

Komponentní moduly jsou takové, které do systému přidávají obsah ve formě komponent, přičemž komponentou rozumíme jedno z následujících:

- model formálního jazyka
- problém neboli úlohu řešitelnou na modelech formálního jazyka
- algoritmus schopný řešit daný problém na určité množině modelů formálních jazyků

Každý komponentní modul při zavedení zaregistruje každou svoji komponentu pod unikátním názvem. Technicky je možné, aby jeden modul obsahoval více komponent, ale pro zachování maximální modularity je doporučeno dodržovat pravidlo, že jeden modul obsahuje právě jednu komponentu. Toto pravidlo je rozumné porušit v případě, že jedna komponenta závisí na přítomnosti druhé komponenty. Například nový algoritmus a nový problém, který definuje jako svůj výchozí algoritmus právě tento nový algoritmus, je rozumné dát do jednoho modulu, pokud autor nepředpokládá, že by se nový algoritmus využil v budoucnu v jiném novém problému. Pokud by se stalo, že dvě či více komponent na sobě závisí navzájem, čili tvoří takzvaný kruh závislostí, tak je nutné je umístit do jednoho modulu.

Kromě samotných komponent mohou komponentní modely obsahovat i rozhraní a abstraktní třídy. Rozhraní a abstraktní třídy se však neregistrují, jelikož není možné přímo vytvořit jejich instanci.

Model

Model formálního jazyka je v prostředí reprezentován jako jedna třída nesoucí jméno tohoto modelu. Tato třída se skládá z dalších dílčích tříd a jen společně s několika pomocnými třídami utváří model jako funkční komponentu. Toto rozčlenění dává uživateli možnost vytvořit nový model prostým upravením jedné dílčí třídy existujícího modelu. Při vytváření nového modelu by měl model navazovat na nejbližší vhodnou třídu či rozhraní v hierarchii modelů (viz obrázek 4.1).

Co se výpočtů týče, tak model je pouze statický objekt a mezi jeho zodpovědnosti tedy patří jen schopnost načíst instanci modelu, uchovat ji se všemi jejími pravidly a nezbytnými součástmi a vytvořit její počáteční konfiguraci[4]. Model ještě musí umět zjistit množinu pravidel, které lze aplikovat na danou konfiguraci, a vytvořit jednu či více nových konfigurací aplikací pravidla na danou konfiguraci. Nové konfigurace musí být stejného typu jako původní konfigurace, což lze zaručit vytvořením a upravením kopie. Díky tomu, že model vytváří počáteční konfiguraci, a protože všechny ostatní konfigurace během výpočtu vznikají jako kopie, které jsou následně upravené, získává model schopnost si určit, jakou implementaci konfigurace bude používat.

Konfigurace se výrazně podílí na funkčnosti modelu, protože mezi její nejdůležitější vlastnosti spadá schopnost rozhodnout se, zda je v kontextu daného modelu akceptující a/nebo ukončující. **Ukončující konfigurace** je taková konfigurace, která ukončuje část výpočtu kvůli tomu, že na ni algoritmy nesmí či nemohou aplikovat žádná další pravidla a vytvořit tak v kontextu daného modelu novou konfiguraci. Základním důvodem, proč není možné aplikovat na konfiguraci pravidlo je to, že množina aplikovatelných pravidel na tuto

[4]Konfigurace je pojem z názvosloví automatů. U gramatik se termínem konfigurace myslí větná forma. Počáteční konfigurace gramatiky je tedy věta, která obsahuje pouze počáteční neterminální symbol gramatiky.

konfiguraci je v kontextu daného modelu prázdná. Konkrétní model má možnost definici své ukončující konfigurace rozšířit. Definice **akceptující konfigurace** záleží na modelu, ke kterému se konfigurace vztahuje. Příklad takové definice lze nalézt v oddílu 2.1.2 v části zabývající se konečnými automaty. Jelikož gramatiky jsou generativní modely, a tudíž nemají svoji definici akceptující konfigurace, tak u gramatik definujeme akceptující konfiguraci jako větu, tedy jako každou větnou formu, která obsahuje pouze terminální symboly. Dále konfigurace musí být schopny vytvořit svoji kopii, jak už bylo zmíněno výše. Konfigurace jsou také složené z další dílčích tříd, které se liší mezi konfiguracemi pro automaty a gramatiky. Více podrobností lze najít v kapitole 4.

Načítání instancí modelu je delegováno na takzvaný `Načítač instancí` (anglicky *instance loader*). Při registraci nové třídy modelu je nutné zároveň registrovat i načítač pro tento model. Minimální požadavek na načítač instancí modelu je takový, aby syntaxe jím načítaných instancí vyhovovala správnému zápisu tohoto modelu. Principiálně funguje načítač jako syntaktický analyzátor (anglicky *parser*). Prostředí obsahuje připravené abstraktní načítače pro automaty a gramatiky, kterým pro správnou funkci stačí doplnit informace o tom, jaké konkrétní třídy mají pro instanci modelu vytvářet. Pro definování syntaxe pravidel tyto předpřipravené načítače používají takzvanou pravidlovou formu, o které se lze více dočíst v podkapitole 4.3 zabývající se implementací načítání instancí.

Další důležitou součástí modelu je třída reprezentující jeho pravidla. Jaká konkrétní třída bude použita při načítání pravidel, je definováno v příslušné metodě v načítači instancí modelu. Obecně řečeno, tato pravidla slouží modelu pro změnu konfigurace, a tím pádem dávají modelu schopnost posouvat výpočet dopředu. Jak již už bylo uvedeno výše, aplikování pravidla a vytvoření nové konfigurace patří mezi zodpovědnosti modelu, avšak třída pravidla má kromě uchovávání součástí tohoto pravidla i další svoji důležitou zodpovědnost, a to schopnost rozhodnout, zda je pravidlo aplikovatelné na danou konfiguraci.

Problém

Problémem se v tomto textu myslí formální problém či praktická úloha řešitelná na modelech formálního jazyka. Při výpočtu problému v podpůrném prostředí zvolený problém určuje, co se bude v tomto výpočtu řešit, a jaký je účel výpočtu. Stejně tak jako modely mají i problémy svoji hierarchii, kterou by měl nový problém vhodně rozšiřovat (viz obrázek 4.2). Problémy obecně mohou simulovat chod instance modelu a zjišťovat z něj různé informace, získávat informace o samotné struktuře instance modelu, optimalizovat instanci modelu, měnit strukturu instance modelu či transformovat instanci modelu do instance jiného modelu.

Hlavní modul inicializuje problém a předává mu uživatelské parametry, které obsahují všechny potřebné informace, jako například zvolené modely a vybraný algoritmus, k vykonání výpočtu problému. Spuštěný problém je pak zodpovědný za načítání instancí modelů, jež je delegováno na příslušné načítače instancí (viz pododdíl Model v oddílu 3.3.1), zpracování vstupů, připravení výpočtu algoritmů, sesbírání a interpretace dílčích výsledků a sestavení konečného celkového výsledku. Je možné říci, že problém tvoří takovou významovou obálku, která obaluje spouštění algoritmů na zvolených modelech s cílem získat výsledky v konkrétní podobě za specifickým účelem.

Po dokončení inicializace je tedy problém spuštěn, přičemž samotný problém rozhoduje o tom, jak budou výpočty probíhat a jak budou využívány načtené instance modelu a vstupní soubory. Algoritmy mají vestavěnou schopnost spouštět se paralelně, čehož by

měl problém při vytváření instancí algoritmů a jejich vykonávání využít. O algoritmech pojednává podrobněji následující pododdíl níže.

To, jakým způsobem problém využije vstupní soubory, zcela závisí na povaze problému. Problémy, které mají spíše simulační podobu (simulují chod instance modelu), jako například problém členství nebo problém zastavení, nejspíše použijí vstupní soubory jako vstupy pro načtené instance modelů. Transformační problémy, jako například různé optimalizace popisu instance, převod na deterministický model nebo převod na instanci jiného modelu, ani nemusí používat žádné jiné vstupy než právě popisy instancí.

Obdobně podoba konečného výsledku bude reflektovat účel problému. Například výsledkem problému členství je odpověď ano či ne, a v případě odpovědi ano další dodatečné informace jako posloupnost derivací či derivační strom. Výsledkem transformačního problému však může být nový upravený popis instance modelu. Problém musí být schopen vypsat konečný výsledek na standardní výstup a uložit výsledek do souboru, přičemž to, která z těchto metod bude použita, si volí uživatel při spuštění problému.

Ne všechny problémy musí být úplně obecné. Některé problémy můžou být relevantní například pouze pro gramatiky, nebo pro ještě menší množinu modelů. Stejně tak pouze některé algoritmy dává smysl použít v určitém problému. Každý problém tedy specifikuje nejobecnější podporované rozhraní či (abstraktní) třídu v hierarchii modelů a v hierarchii algoritmů. Pokud uživatel zvolí takový model či algoritmus, který není instancí odpovídajícího podporovaného rozhraní či třídy, tak se jedná o chybu uživatelského vstupu, na niž program upozorní chybovou hláškou.

Algoritmus

Algoritmus obsahuje detailní popis toho, jak se má zvolený problém provést. Obdobně jako modely a problémy mají i algoritmy svoji hierarchii, již by měl nový algoritmus vhodně rozšiřovat (viz obrázek 4.3).

Instance algoritmu je inicializována instancí modelu, na níž se bude výpočet algoritmu provádět, a nebo více instancemi ve specifických případě, například když algoritmus porovnává dvě a více instancí, časovým limitem (anglicky *timeout*) a případně dalšími vstupy algoritmu. Pokud algoritmus neprovádí žádné úpravy ve struktuře instance modelu, tak je možné stejnou instanci modelu použít pro více souběžných výpočtů algoritmu. Každý algoritmus musí mít schopnost být paralelně spuštěn ve vedlejším vlákně tak, aby hlavní vlákno vykonávající kód problému mohlo pokračovat v práci. Po dokončení výpočtu musí být algoritmus schopen vrátit výsledek v textové podobě, a zároveň by měl být výsledek dostupný v objektové podobě pro případné další zpracování pomocí k tomu určené metody (anglicky takzvaný *getter*).

Důležitá vlastnost, kterou musí každý algoritmus podporovat, je časový limit, protože v mnoha případech se může stát, že by výpočet běžel ve smyčce donekonečna. Průběh algoritmu, který skončí dosažením časového limitu, je označen za neúspěšný v případě, že nenašel žádné požadované řešení, přičemž by informace o dosažení limitu měla být obsažena ve výstupu, jelikož uživatel potřebuje vědět, že tento neúspěch není stoprocentní, protože s dostatečným množstvím času by výpočet mohl teoreticky uspět. Pokud vykonávání algoritmu dosáhne časového limitu a zároveň nalezne alespoň jedno požadované řešení, tak je označeno za úspěšné a jsou vráceny podrobnosti úspěchu. V případě, že algoritmus najde jen určitou neprázdnou vlastní podmnožinu ze všech požadovaných řešení, tak ta řešení, jež našel, označí za úspěšná, a ta, jež nenašel či nestihl najít, označí za neúspěšná s informací o dosažení časového limitu.

K uchovávání historie konfigurací v průběhu algoritmu, který má simulační podobu, je možné použít na třídu `Historie konfigurací`. Algoritmus se potom stará jen o aplikování pravidel na konfiguraci získanou z historie, generování nových konfigurací a jejich následné ukládání do historie. Historie pak řeší, jakým způsobem se konfigurace ukládají do paměti a jaké datové struktury se k tomu využívají. To umožňuje rozdělit implementaci algoritmu na čistě logickou část a část starající se o uchovávání dat. Použití či implementace vlastní historie konfigurací je zcela volitelné, avšak použití třídy konfigurace je pro algoritmy provádějící simulaci chodu modelu nezbytné, jelikož v konfiguraci jsou definované ukončující a akceptující podmínky pro daný model (viz pododdíl Model v oddílu 3.3.1).

Algoritmy jsou typicky velmi specializované. Řeší většinou jen jeden problém na jednom konkrétním modelu či jen malé množině modelů. Obdobně jako problémy definují algoritmy nejobecnější rozhraní či (abstraktní) třídu modelu, které podporují. Algoritmy nedefinují podporované problémy, protože v konečném důsledku kterýkoliv nový problém může chtít existující implementovaný algoritmus použít, a to i kdyby třeba jen jako část celého řešení.

3.3.2 Hlavní modul

Hlavní modul slouží jako centrální řadič, jenž obsluhuje žádosti o zpracování výpočtu přicházející z modulů rozhraní a spravuje informace o všech registrovaných modelech, algoritmech a problémech.

Poté, co modulární systém dokončí zavádění všech načtených modulů, začne hlavní modul čekat na požadavek o zpracování výpočtu, který bude zaslán jedním z modulů rozhraní. Po obdržení tohoto požadavku hlavní modul zahájí svoji činnost zpracováním uživatelských voleb. Uživatel zde má povinnost specifikovat právě jeden problém k vykonání, model či modely, na kterých bude problém vykonán, a alespoň jeden soubor popisují instanci modelu. Dále uživatel může specifikovat vstupní soubory pro vybraný problém, algoritmus, jenž má problém vykonat, a další parametry ovlivňující například průběh algoritmů, způsob generování výstupu nebo časový limit pro vykonávání algoritmů. V případě, že uživatel nezvolí explicitně žádný algoritmus, tak by měl problém být schopen na základě uživatelem zvoleného modelu či zvolených modelů vybrat vhodný výchozí algoritmus.

Následně je hlavním modulem spuštěn vybraný problém, který zajistí načtení všech instancí modelu a vstupů, spuštění výpočtu algoritmu pro každou instanci a každou množinu vstupů algoritmu, a sesbírání dílčích výsledků z jednotlivých vykonání algoritmů pro sestavení celkového výsledku problému. Zodpovědnost za načítání instancí modelu je delegována na takzvaný načítač instancí (anglicky *instance loader*) registrovaný pod stejným identifikátorem, jako odpovídající třída modelu. Nakonec hlavní modul předá vyřešený problém zpět použitému rozhraní, které vypíše, uloží či jinak zobrazí výsledek výpočtu získaný od řešeného problému.

Mimo svoji základní činnost obsahuje hlavní modul také mnoho rozhraní, abstraktní třídy a implementace obecných modelů, algoritmů a problémů.

3.3.3 Moduly rozhraní

Moduly rozhraní jsou takové moduly, které obsahují uživatelské či aplikační rozhraní. Není tedy řeč o rozhraní z objektově orientovaného programování. Tyto moduly by neměly registrovat žádné komponenty, aby nevznikala závislost mezi komponentou a konkrétním rozhraním.

Princip funkce modulu rozhraní je takový, že modul zpracuje příchozí požadavek do podoby, které rozumí hlavní modul, a požadavek mu předá k vykonání. Při zpracování

požadavku je možné provádět kontroly, zda je požadavek správně definován. Modul rozhraní se dále stará jen o zobrazení výsledků a chyb. Díky tomuto principu je snadné přidávat do aplikace nová naprosto rozdílná rozhraní nebo existující nepoužívaná rozhraní odebrat, a to bez nutnosti zásahu do zdrojového kódu hlavního modulu či ostatních modulů.

Aplikace obsahuje jeden vzorový modul rozhraní, a to modul implementující rozhraní příkazové řádky, jež je vhodné pro zpracování požadavků z dávkových skriptů. Dalšími rozhraními, kterými by bylo možné aplikaci v budoucnu rozšířit, by mohla být například grafické uživatelské rozhraní nebo rozhraní webové služby (anglicky *web service*).

V případě, že některý nový problém či algoritmus vyžaduje novou uživatelskou volbu, kterou existující rozhraní nepodporuje, tak je nutné upravit modul rozhraní a volbu podpořit. Určitým způsobem k obejití toho problému je použití jednoho ze vstupních souborů k předání těchto specifických parametrů. Pokud nový model vyžaduje nový parametr, tak se obecně nejedná o žádný problém, jelikož je možné přidat parametr v načítači instancí tohoto modelu. Případně lze použít existující dodatečné parametry v načítači instancí.

3.4 Modulární systémy a rámce

Řešení má být implementováno jako modulární aplikace. Tato podkapitola se zabývá výběrem modulárního systému či rámcového řešení. Před samotným výběrem je nutné definovat požadované vlastnosti pro zhodnocení možností. Prvním požadavkem je, aby nový modul bylo možné sestavit bez nutnosti opětovné kompilace všech existujících modulů. Druhou požadovanou vlastností je, aby jednotlivé moduly byly po kompilaci stejně samostatné a oddělitelné, jako byly v podobě zdrojového kódu. Tato vlastnost dává uživateli možnost používat jen hlavní modul spolu s moduly, které potřebuje, a vynechat ze spuštění aplikace pro něj nepotřebné moduly. Třetí vlastnost, na kterou se zaměřuji, je možnost podpory využití modulů implementovaných v různých programovacích jazycích.

Přijatelný systém pro řešení musí splňovat první dva požadavky. Dále je preferován systém s co největší podporou třetího požadavku. V poslední řadě je v úvahu brán počet dalších funkcí modulárního systému a jeho obecné vlastnosti.

3.4.1 Vestavěné modulární systémy programovacích jazyků

Různé programovací jazyky, jako například Python, Ruby nebo Rust, obsahují vestavěný modulární systém. Případně pro další jazyky existuje rozšíření s modulárním systémem, jako například *Java Platform Module System (JPMS)* v jazyku Java verze 9. Tyto vestavěné modulární systémy většinou obsahují jen minimální či značně omezenou množinu funkcí pro tvorbu modulárních aplikací. Pro řešení by byly použitelné, ale specificky navržené modulární systémy nabízejí více vlastností. Vestavěný modulární systém konkrétního programovacího jazyka je navíc pevně spojen s daným jazykem a nemá tedy žádnou podporu pro využití modulů napsaných v jiných programovacích jazycích.

3.4.2 Dynamický modulární systém OSGi

OSGi je souhrn standardizovaných specifikací navrhujících dynamický modulární systém primárně pro aplikace napsané v jazyku Java. Mezi standardní vlastnosti OSGi rámcového řešení se řadí možnost měnit, přidávat a odebírat moduly i za běhu systému. OSGi totiž poskytuje prostředí, jež spravuje běh modulů a je schopné o nich vypsat i řadu informací, například v jakém stavu se daný modul nachází a jaké služby tento modul poskytuje.

Existuje mnoho různých implementací OSGi, a to nejen pro Javu, ale i pro další jazyky. Tyto implementace často rozšiřují množinu standardních OSGi funkcí různými dalšími funkcemi. Při psaní tohoto oddílu jsem vycházel z oficiálních stránech OSGi [8].

Equinox OSGi

Equinox OSGi je implementace OSGi specifikace pro jazyk Java vytvořená nadací Eclipse[5]. Jádro Equinox OSGi se používá jako referenční OSGi implementace. Kromě tohoto jádra však Equinox rozšiřuje standardní specifikaci OSGi o celou řadu dalších funkcí a podpůrných nástrojů.

Mimo jiné tato implementace do určité míry podporuje i využití modulů napsaných v dalších jazycích, například `C++`. Pro zprovoznění těchto modulů je však potřebná náročná dodatečná konfigurace a řada externích nástrojů jako OSGi Native Library Support pro `C++`, a kombinace modulů v různých jazycích navíc přináší problémy s kompatibilitou rozhraní a interoperabilitou těchto modulů. Díky nutnosti dodatečné konfigurace a použití těchto nástrojů je tvorba modulu v jiném programovacím jazyku než v Javě výrazně náročnější a potencionálně ztrácí výhodu použití jiného jazyka, pokud výhodou měla být uživatelská přívětivost a rychlá implementace modulu. I přesto by to však mělo být možné, a v případě že uživateli jde o zrychlení algoritmu použitím nízkoúrovňového jazyka snad i přínosné. V těchto dvou odstavcích používám parafráze z webové stránky Equinox OSGi [2].

Dalším faktorem při zvažování vhodnosti rámcového řešení OSGi je fakt, že jím primárně podporovaným jazykem je Java. Java nabízí možnost tvorby multiplatformních aplikací. Navíc v sobě Java kombinuje dobrý kompromis mezi rychlým vykonáváním nízkoúrovňových jazyků jako je `C` či `C++` a uživatelskou přívětivostí vysokoúrovňových jazyků jako například Python. Tvorba nového modulu tedy bude pro uživatele relativně přívětivá a průběh spuštěných algoritmů bude relativně rychlý. Dále stojí za zmínku i fakt, že OSGi i přes to, že nabízí celou řadu pokročilých funkcí pro správu modulárních aplikací, je malý a kompaktní systém, který je možné provozovat i v řadě vestavěných zařízení. Po zhodnocení všech vlastností modulárního systému OSGi a implementace Equinox OSGi jsem se rozhodl použít tento modulární systém při návrhu výsledné aplikace. Možnost dynamické výměny modulů za běhu sice není v mém návrhu potřebná, ale poskytuje dobrou příležitost pro budoucí uživatele a budoucí rozšíření. Pokud by aplikace byla v budoucnosti nasazena například na server, kde by zpracovávala požadavky přicházející z rozhraní webové služby, tak by možnost rozšíření tohoto systému přidáváním modulů bez nutnosti systém odstavit byla velmi užitečná.

3.4.3 Další modulární systémy

V tomto oddílu se stručně zabývám svými zjištěními při hledání vhodných modulárních systémů. Různých systémů a rámcových řešení je k dispozici celá řada, z většiny se ale jedná o specializované systémy na určitý druh aplikace či na aplikace používající určitou technologii. Takovéto specializované systémy nejsou v souladu s návrhem mého řešení, tudíž nestojí za to se zde o nich zmiňovat. Níže popíšu další dva modulární systémy, jejichž využití by pro vypracování řešení této práce mohlo být zajímavé.

[5] Nadace Eclipse, anglicky Eclipse foundation, dostupné na: https://www.eclipse.org/

Microsoft Prism

Microsoft Prism je rámcové řešení pro tvorbu volně vázaných (anglicky *loosely coupled*) aplikací s grafickým rozhraním využívajícím technologií WPF, .NET MAUI nebo Xamarin forms. Prism podporuje správu modulů za běhu aplikace, řešení závislostí modulů včetně detekce kruhových závislostí, stahování modulů na vyžádání (anglicky *on-demand*). Prism sice podporuje téměř všechny .NET programovací jazyky, mezi které se řadí například `C#`, `C++`. `F#` a Visual Basic, a kombinace modulů napsaných v různých z těchto jazyků v jedné aplikaci je možná, ale nedoporučuje se takovouto kombinaci používat, jelikož vede k výraznému zvýšení komplexity aplikace a problémy s kompatibilitou. V tomto odstavci vycházím z dokumentace pro Prism [7].

Tento systém je dobrým řešením se širokou řadou modulárních funkcí. Oproti OSGi však postrádá obecnost, jelikož je Prism příliš specializovaný na řešení aplikací využívajících výše zmíněné technologie uživatelského rozhraní. Možnost tvorby modulů v různých jazycích je podobně omezená jako v OSGi a lze předpokládat podobné problémy, pouze se mezi nimi liší množina podporovaných jazyků.

Spring

Spring je typicky známý jako modulární rámcové řešení pro tvorbu reaktivních webových aplikací v jazyku Java. Ve skutečnosti se však jedná o velmi rozsáhlý systém, který kromě webových aplikací podporuje i tvorbu celé řady dalších typů aplikací, například cloudové aplikace, aplikace pro událostmi řízené řetězové zpracování, mikroslužby, ale i aplikace pro dávkové zpracování. Jednou z vlastností Springu je i tvorba modulárních aplikací. Princip vývoje modulárních aplikací ve Springu spočívá ve využití jedné ze základních funkcí Springu, a to vkládání závislostí (anglicky *dependency injection*), která dovoluje vytvářet volně propojený kód pomocí delegace tvorby objektů a jejich správy na toto rámcové řešení. Rozšíření Spring Modules pak přidává další funkce pro tvorbu modulárních aplikací. Spring neobsahuje dynamickou správu modulů (tedy výměnu modulů za běhu aplikace), ale rozšíření Spring Modules nabízí podporu pro integraci s OSGi.

Jelikož i Spring sám o sobě je modulární rámcové řešení, je tedy možné z něj použít pouze ty části, které jsou využity při tvorbě aplikace bez toho, aby zbytek tohoto velkého systému zatěžoval výsledné řešení. Zároveň to však ponechává možnost kdykoliv aplikaci snadno rozšířit o další funkce, které Spring nabízí, například nasazení na cloudu a zpřístupnění aplikace pomocí mikroslužeb. Použití Springu jako modulárního systému pro výslednou aplikaci by mohlo být určitě zajímavé a nejspíše i nestandardní řešení, jehož výhodou by byla snadná rozšiřitelnost v oblasti nasazeného prostředí. Nakonec jsem se však přiklonil k použití OSGi, jakožto obecnějšího, lehčího a kompaktnějšího řešení. K vypracování těchto dvou odstavců jsem čerpal informace z webové stránky Springu [10], kde lze najít i dokumentaci k tomuto rámcovému řešení.

Kapitola 4

Implementace

V této kapitole popisuji implementaci výsledné aplikace a použité technologie. Aplikace je implementována v jednotlivých modulech s využitím modulárního systému Equinox OSGi. Více o výběru systému a jeho vlastnostech se lze dočíst v kapitole o návrhu řešení v podkapitole 3.4, která se věnuje modulárním systémům. V následujících odstavcích zmiňuji názvy konkrétních metod. Ve většině případů není nutné uvádět celou signaturu dané metody či ji vícekrát opakovat, proto často používám zápis `metoda(...)` opomíjející její návratovou hodnotu a parametry.

Každý modul, nebo v terminologii OSGi též svazek (anglicky *bundle*), musí obsahovat svůj takzvaný Aktivátor. Aktivátor obsahuje metody `start(BundleContext context)` a `stop(BundleContext context)`. Metoda `start(...)` se volá vždy, když je příslušný modul načten a spuštěn systémem OSGi. Obdobně je metoda `stop(...)` volána při zastavení daného modulu. Tyto metody jsou použity například pro registraci a odregistraci komponent v komponentních modulech a pro výpis ladících informací o aktivaci či deaktivaci modulu. Parametr `BundleContext` ve výše zmíněných metodách slouží jako přístupová třída k rozhraním OSGi rámce.

4.1 Jádro podpůrného prostředí

Jádro aplikace je obsaženo v hlavním modulu. Nejdůležitějšími třídami jádra jsou `Spouštěč problémů` (anglicky *problem launcher*), `Registr komponent` (anglicky *component register*) a třída `Jádro` (anglicky *core*).

Spouštěč problémů je nejdůležitějším rozhraní aplikace, jelikož obsahuje statickou metodu `launch(IProblemArguments arguments)`, která slouží k zahájení výpočtu zvoleného problému. Tuto metodu volají moduly uživatelského či aplikačního rozhraní s příslušnými informacemi o tom, jaký výpočet se má provést. Všechny tyto informace jsou obsaženy v parametru `IProblemArguments arguments` metody `launch(...)`. Metoda následně s využitím registru komponent zjistí třídu zvoleného problému, a pokud existuje, vytvoří instanci tohoto problému. Nová instance problému je inicializována stejným parametrem `IProblemArguments arguments`, se kterým byla volána metoda `launch(...)`, tudíž instance problému má k dispozici všechny informace o požadavku o vykonání výpočtu. Následně je problém spuštěn, a pokud nedošlo k žádné chybě, tak metoda `launch(...)` vrátí vypočítanou instanci problému, jíž obdrží zpět modul rozhraní, který se postará o zobrazení či uložení výsledku získaného z tohoto vyřešeného problému patřičným způsobem. Pokud někde v průběhu vytváření instance problému či jeho výpočtu dojde k chybě, tak metoda

`launch(...)` vyhodí odpovídající výjimku, na kterou modul rozhraní musí uživatele náležitě upozornit.

Registr komponent slouží k registrování jednotlivých komponent (viz oddíl 3.3.1). Každá registrace komponenty musí obsahovat unikátní identifikátor, pod kterým je komponenta registrována, třídu reprezentující tuto komponentu a řetězec s popisem komponenty. Při registraci modelu je navíc nutné uvést i načítač instancí tohoto modelu. Každý typ komponenty má svůj registr se svými unikátními identifikátory, tudíž komponenty různých typů mohou sdílet stejný identifikátor. Registr komponent má dále metody zprostředkovávající přístup ke třídám, popisům a případně načítačům instancí registrovaných komponent pod daným identifikátorem. Kromě registrování komponent je samozřejmě možné komponenty i odregistrovat. Odregistrace komponent je nutná, jelikož OSGi podporuje dynamickou správu modulů a tudíž i vypnutí modulu. Bez odregistrace by po vypnutí komponentního modulu zůstaly registrované neexistující komponenty.

Hlavním důvodem pro existenci registru komponent je poskytnutí rozhraní pro dynamický výběr komponent použitých při výpočtu, které navíc podporuje předem neznámý výčet komponent. Všechny vzniklé instance komponent vznikají tak, že informace o provedení výpočtu poskytnuté modulem rozhraní obsahují identifikátory zvolených komponent, na základě kterých se pomocí registru komponent dynamicky vyberou třídy použité pro vytvoření instancí. Instance problému se vytváří ve výše zmíněném spouštěči problémů. Tvorbu instancí modelů a algoritmů si pak řídí sám spuštěný problém, který ale též pro identifikaci správné třídy využívá registr komponent.

Třída `Jádro` zapouzdřuje takzvanou **vykonávací službu** (angl. *executor service*), která poskytuje skupinu vláken pro prostředím poskytnutou implementaci paralelního vykonávání algoritmů. Všechny algoritmy které rozšiřují abstraktní třídu algoritmus (`AAlgorithm` viz obrázek 4.3) jsou spouštěny na vláknech této vykonávací služby. Počet dostupných vláken ve vykonávací službě je o jedna menší než počet dostupných logických procesorů stroje, na kterém je prostředí spuštěno. O jedna méně z toho důvodu, že kromě vláken, na kterých probíhají výpočty spuštěných algoritmů, vždy běží ještě jedno hlavní vlákno vykonávající zvolený problém, jenž je zodpovědný za spuštěné výpočty. Počet dostupných vláken je takto nastaven z toho důvodu, že vyšší počet vláken, než je počet dostupných logických procesorů, typicky zhoršuje výkonost paralelního výpočtu. Kromě vykonávací služby `Jádro` obsahuje ještě statickou metodu pro konfiguraci ladících výpisů napříč moduly.

Hlavní modul dále obsahuje i třídu `CoreConstants`, jež obsahuje definice všech prostředím používaných konstant. Mnoho z těchto konstant jsou řetězce obsahující regulární výrazy (anglicky *regular expressions*).

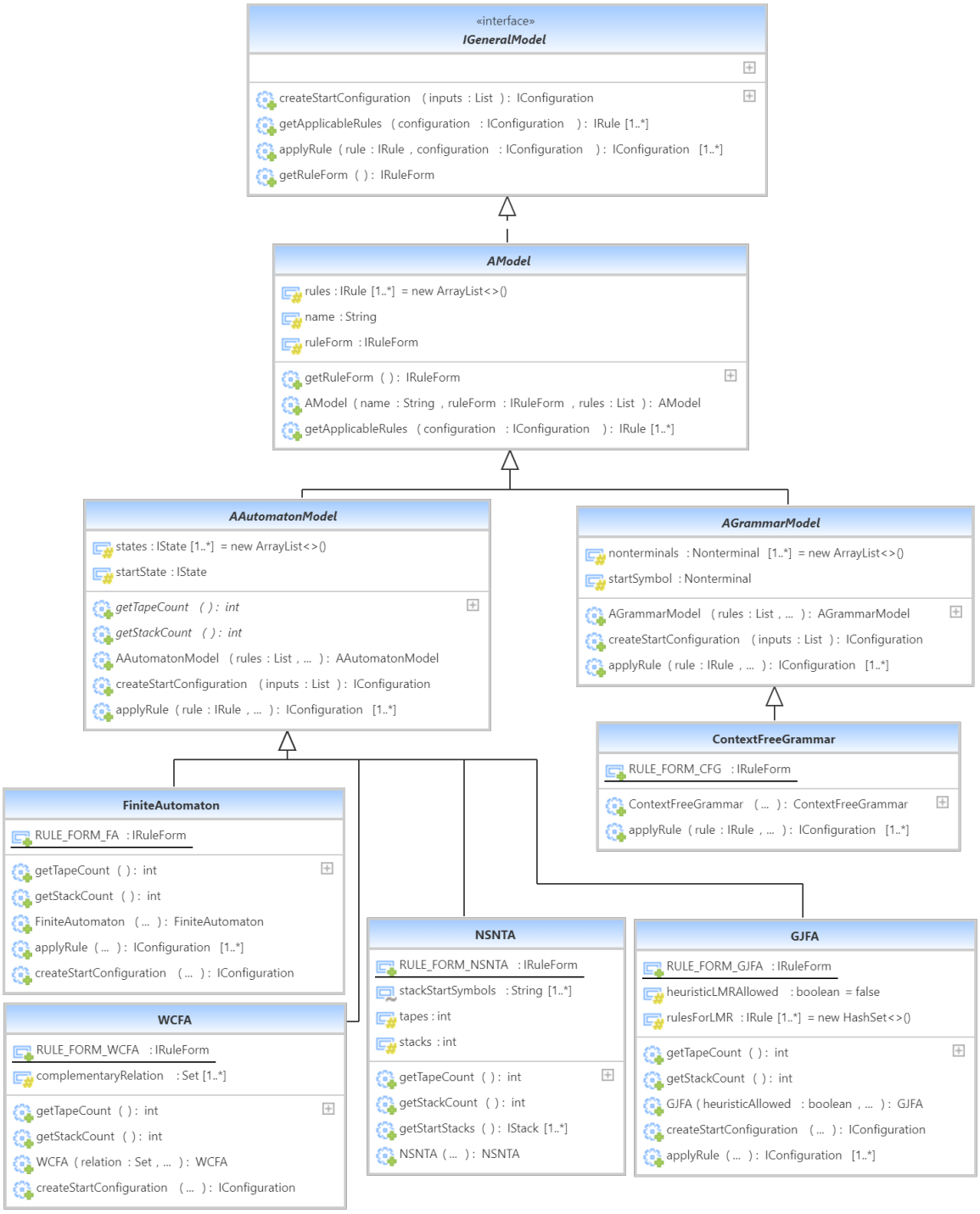
4.2 Obecná rozhraní

Podpůrné prostředí nabízí celou škálu obecných rozhraní, a to od modelů, algoritmů a problémů, přes konfigurace, pravidla a vstupní pásky až po načítače (angl. *loaders*). Diagramy tříd v této podkapitole obsahují pouze nejdůležitější metody a některé hlavičky metod byly zkráceny.

4.2.1 Rozhraní modelů

Hierarchii rozhraní modelů lze vidět na obrázku 4.1. Obecný model (`IGeneralModel`) navrhuje operace jako vytvoření počáteční konfigurace, získání aplikovatelných pravidel na danou konfiguraci, aplikování pravidla na danou konfiguraci a předání použité pravidlové

formy pro načtení instance tohoto modelu. Uvedený abstraktní model (`AModel`) obsahuje implementaci metod z rozhraní `Obecný model`, a dále má i další vlastnosti jako uchovávání pravidel instance a jejího pojmenování.



Obrázek 4.1: **Diagram tříd - typová hierarchie modelů.** Třídy `ContexFreeGrammar`, `FiniteAutomaton`, `NSNTA`, `GJFA` a `WCFA` jsou konkrétní implementované modely.

Abstraktní model automatu (`AAutomataModel`) připravuje další obecnou podporu pro automaty, a to uchování seznamu stavů, tvorbu počáteční konfigurace, definici počátečního stavu, který pak vystupuje v tvorbě počáteční konfigurace, a implementaci metody aplikující pravidlo na danou konfiguraci. Součástí obecné implementace modelu automatu jsou i obecné třídy jako `Pravidlo automatu`, `Konfigurace automatu`, `Stav`, `Jednoduchá páska` a `Jednoduchý zásobník`. V metodě vytvářející počáteční konfiguraci automatu se s použitím jedné či více instancí třídy `Jednoduchá páska` a nula či více instancí třídy `Jednoduchý zásobník` vytváří instance této obecné konfigurace automatu, která podporuje více vstupních pásek i zásobníků. Obecná konfigurace automatu pak implementuje i metody testující, zda je konfigurace akceptující a/nebo ukončující. Akceptující konfigurace je taková, ve které se automat nachází v koncovém stavu a všechny vstupní pásky jsou prázdné. Ukončující konfigurace je implementována přesně podle definice v návrhu řešení (viz pododdíl Model v oddílu 3.3.1). Obecné pravidlo automatu také podporuje více vstupních pásek a zásobníků. Pravidlo je aplikovatelné, pokud se jeho levá strana shoduje s danou konfigurací automatu. Aplikace pravidla je pak implementovaná jednoduše jako přečtení znaků z pásek a zásobníků, a nastavení konfigurace do výsledné stavu uvedeného na pravé straně pravidla. Obecný model automatu také zavádí abstraktní metody pro zjištění počtu vstupních pásek a zásobníků v instanci automatu.

Konkrétní implementace automatů pak rozšiřují či nahrazují obecnou implementaci dle své potřeby. Na obrázku 4.1 je vidět zavedená konvence, že každý konkrétní model obsahuje veřejný statický konstantní atribut uchovávající definici pravidlové formy tohoto modelu. Detaily o implementaci jednotlivých konkrétních modelů a definicích jejich pravidlových forem lze nalézt níže v kapitole 4.5.

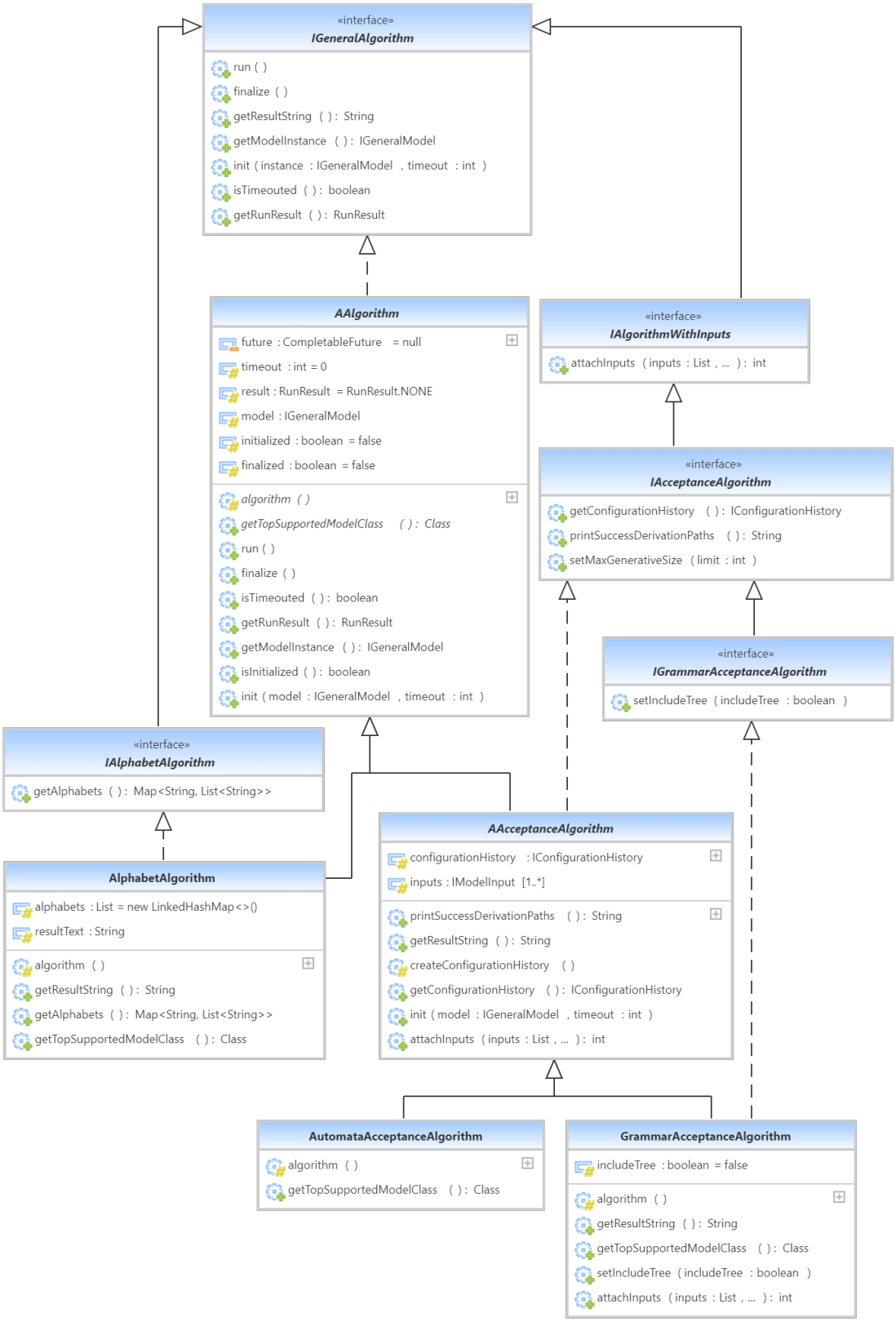
Obecný abstraktní model (`AModel`) je dále rozšířen abstraktním modelem gramatiky (`AGrammarModel`), jenž uchovává seznam neterminálů a počáteční neterminál gramatiky. Terminální symboly nejsou uchovávány samostatně, ale jsou obsaženy v jednotlivých pravidlech v seznamu pravidel. Abstraktní model gramatiky též implementuje metody pro vytvoření počáteční konfigurace a aplikování pravidla na konfiguraci. Proč je větná forma v implementaci gramatiky pojmenována jako konfigurace, je možné se dočíst v oddílu 3.3.1. Obecná implementace modelu gramatiky obsahuje i další obecné třídy, a to `Pravidlo gramatiky`, `Konfigurace gramatiky` a rozhraní `Symbol gramatiky`, jež je implementováno třídami `Neterminál` a `Terminál`. Konfigurace gramatiky odpovídá její větné formě a je implementována jako derivační graf. Derivační graf a ne strom z toho důvodu, že obecně pro všechny možné typy gramatik (například pro neomezené gramatiky), tvoří derivace acyklický derivační graf. Počáteční konfigurace je samozřejmě vytvořena z počátečního symbolu gramatiky. Aplikace pravidla je pak implementována obdobně jako u pravidla automatu. Tedy pokud levá strana odpovídá současné konfiguraci, tak se příslušný neterminál (či neterminály v případě neomezených gramatik) rozgenerují na řetězec symbolů na levé straně pravidla. Zatímco u automatů (až na výjimky) je výsledkem aplikace pravidla na konfigurace právě jedna konfigurace, tak u gramatik může být výsledkem několik konfigurací, protože pravidlo může být obecně aplikováno na více různých místech v řetězci symbolů.

4.2.2 Rozhraní problémů

Rozhraní obecný problém (`IGeneralProblem`) z hierarchie problémů vyobrazené na obrázku 4.2 obsahuje metody pro inicializaci problému, jeho spuštění a získání výsledků v textové podobě či v podobě použitých instancí spuštěných algoritmů. Abstraktní problém `AProblem` implementuje kompletní inicializaci problému, nástroje pro tvorbu výsledků

Obrázek 4.2: **Diagram tříd - typová hierarchie problémů.** Třídy `MembershipProblem` a `AlphabetProblem` jsou konkrétní implementované problémy.

požadovaných rozhraním obecného problému a metody pro načítáním instancí a vstupů. Obecný problém zavádí abstraktní metodu `fallbackAlgorithmID()` pro volbu výchozího algoritmu pro zvolený model, pokud argumenty problému neobsahují informaci o zvoleném algoritmu. Tuto metodu musí implementovat konkrétní problémy.



Obrázek 4.3: **Diagram tříd - typová hierarchie algoritmů.** Třídy `AlphabetAlgoritm`, `AutomataAcceptanceAlgoritm` a `GrammarAcceptanceAlgoritm` reprezentují konkrétní implementované algoritmy.

4.2.3 Rozhraní algoritmů

Rozhraní obecný algoritmus (`IGeneralAlgorithm`) obsahuje metody pro inicializaci algoritmu, spuštění algoritmu, dokončení algoritmu, otestování, zda vypršel časový limit pro běh algoritmu, získání výsledku a vrácení instance modelu, na které se algoritmus prováděl.

Inicializaci, spuštění, dokončení algoritmu a získání jeho výsledku je implementováno v abstraktní třídě algoritmus (`AAlgorithm`), kterou by měl každý implementovaný algoritmus v podpůrném prostředí rozšiřovat. Poté, co je algoritmus inicializován, je možné ho spustit metodou `run()`. Tato metoda vytvoří nový objekt třídy `CompletableFuture` a použije vykonávací službu z `Jádra` (viz podkapitola 4.1) pro spuštění implementace konkrétního algoritmu v abstraktní metodě `algorithm()`. Konkrétní třída algoritmu tedy musí implementovat tělo algoritmu v metodě `algorithm()`. Jelikož řízení vláken v jazyku Java je kooperativní, tak každý konkrétní algoritmus musí ve svém těle opakovaně kontrolovat metodu `isTimeouted()`, a pokud metoda vrátí kladnou odpověď, tak musí algoritmus nastavit výsledek běhu na *čas vypršel* a okamžitě ukončit svoji činnost. Další možné typy výsledku průběhu algoritmu jsou *úspěch*, *nastala chyba* a *výpočet byl dokončen*. Výsledek *výpočet byl dokončen* znamená, že algoritmus skončil bez nalezení úspěšného výsledku. Před získáním výsledku algoritmu je nutné zavolat metodu `finalize()`, která dokončuje vykonávání algoritmu. Tato metoda je blokující, což znamená, že pokud výpočet algoritmu stále probíhá, tak vlákno volající tuto metodu bude čekat, dokud se výpočet algoritmu nedokončí. Pokud byl výpočet již dokončen, tak volající vlákno nemusí na nic čekat.

Na obrázku 4.3 lze vidět, jak je rozhraní obecný algoritmus rozšířené řadou dalších rozhraní, které buď obsahují volitelné funkce algoritmu nebo slouží pro specifikaci podporovaného algoritmu v jednotlivých problémech. Například rozhraní `IAlphabetAlgorithm` slouží pro určení algoritmu schopného řešit problém abecedy (viz oddíl 4.5.7). Problematika akceptačních algoritmů je řešena v oddílu 4.5.1.

4.3 Načítání instancí modelů

Jak už bylo řečeno v kapitole Návrh řešení v oddílu 3.3.1 zabývajícím se komponenty, tak načítání instancí modelů provádí takzvané načítače instancí. Načítání instancí se může zdát jako jednoduchý problém, který lze vyřešit obyčejným syntaktickým analyzátorem. Ve skutečnosti se však jedná o jednu z nejvíce propracovaných částí aplikace, která uživateli poskytuje velké usnadnění při implementaci nového modelu. Celá infrastruktura načítání se navíc stará i o definování a kontrolu syntaxe zápisu instancí modelu.

Prostředí implementuje obecný abstraktní načítač instancí (`AInstanceLoader`). Proces načítání instancí se spouštím zavoláním metody `List<IGeneralModel> load(List<Path> paths)` tohoto abstraktního načítače. Tato metoda má jako parametr seznam cest k souborům s popisy instancí modelu, a vrací seznam načtených instancí typu obecný model (`IGeneralModel` viz obrázek 4.1). Metoda `load(...)` určuje strukturu procesu načítání a skládá se z řady dílčích metod, z nichž většina je definována jako abstraktní.

Abstraktní načítač instancí modelů je rozšířen abstraktním načítačem instancí automatů (`AAutomataLoader`) a načítačem instancí gramatik (`AGrammarLoader`), které dále specializují proces načítání a implementují většinu abstraktních metod z obecného načítače. Dědičnost načítačů je znázorněna na obrázku 4.4. Jediné metody, jež zůstávají abstraktní a neimplementované, jsou ty metody, jejichž jediná zodpovědnost je vytvořit konkrétní objekt použitý ve výsledné instanci modelu či vytvořit samotnou konkrétní instanci modelu. Metody vytvářející konkrétní objekty, jako jsou například pravidla, instance, stavy,

Obrázek 4.4: **Diagram tříd - typová hierarchie načítačů instancí.** Načítače `FALoader`, `NSNTALoader` a `CFGLoader` zde slouží jako příklady konkrétních načítačů. Diagram obsahuje pouze nejdůležitější metody. Některé hlavičky metod byly zkráceny.

terminály a neterminály, jsou pak implementovány konkrétními načítači instancí konkrétních modelů. Konkrétní načítače instancí musí ještě implementovat metodu `IRuleForm getProvidedRuleForm()`, která načítači poskytuje informace o podobě zápisu pravidel instancí v takzvané **pravidlové formě** (viz následující oddíl 4.3.1).

Při načítání se po řádcích prochází každý soubor, jenž obsahuje vždy právě jeden popis instance modelu. Popis instance se skládá ze dvou částí. První část obsahuje parametry instance ve tvaru `jméno: hodnota`. Těmito parametry mohou být například počáteční stav, koncové stavy, počáteční symboly na zásobnících, počáteční symbol gramatiky a výčet neterminálů. Druhá část je složena z výčtu pravidel, přičemž každý řádek je buď prázdný,

nebo obsahuje korektní zápis pravidla. Správnost zápisu pravidla se kontroluje pomocí regulárního výrazu pro pravidlo získaného z pravidlové formy. Řetězec obsahující pravidlo se v metodě `parseRule(...)` dále rozdělí na jednotlivé části odpovídající regulárním výrazům z pravidlových částí. Z těchto částí se v metodě `parseElement(...)` vytváří zpracované části typu `IParsedRuleElement`. Zpracované části jsou složeny z dat, což může být znak, řetězec, stav nebo symbol gramatiky, a z pravidlové části použité pro jejich načtení. Ze zpracovaných částí se pak metodou `createRule(...)` sestaví načtené pravidlo. Po načtení všech pravidel a dosažení konce souboru je volána metoda `instanceCreation(...)`, jež deleguje tvorbu instance modelu na metodu `createInstance(...)` implementovanou konkrétními načítači. Ta ze všech načtených pravidel a parametrů vytvoří instanci odpovídajícího modelu. Nakonec metoda `load(...)` vrátí seznam všech načtených instancí.

4.3.1 Pravidlová forma a pravidlové části

Pravidlová forma je třída tvořená seznamem pravidlových částí, a jak už její název napovídá, tak určuje syntaktický zápis pravidla. Každý model musí definovat pravidlovou formu svých pravidel a zprostředkovat ji svému načítači instancí. Pravidlová formu spolu s načítačem instancí tvoří dodatečnou úroveň abstrakce mezi zápisem instance modelu a načtenou instancí modelu. Kromě toho, zda je načítaná část pravidla správná, poskytuje pravidlová forma načítači i informaci, o jakou část se jedná. To dává načítači možnost pracovat s částmi pravidel jako s pojmenovanými významovými celky místo jen s textovými řetězci bez dalšího významu.

Pravidlová část je elementární syntaktická část zápisu pravidla. Každá pravidlová část má jméno, typ určující její význam, dva regulární výrazy a řadu dalších vlastností. Dva regulární výrazy z toho důvodu, že jeden neobsahuje zachytávající skupinu (anglicky *capturing group*) a slouží pro pravidlovou formu k vybudování regulárního výrazu pro celé pravidlo. Druhý regulární výraz obsahuje zachytávající skupinu, která by měla zachytit pouze řetězce s daty nesoucími význam. Tento druhý regulární výraz slouží ke zpracování jednotlivých částí v rámci jednoho pravidla. Oba regulární výrazy musí stejně vyhodnotit stejný řetězec, liší se jen výskytem zachytávající skupiny. Například pravidlová část *čtení jednoho znaku z pásky* se zápisem jednoho znaku v uvozovkách by mohla mít nezachytávající regulární výraz `"[^\"]"` a zachytávající regulární výraz `"([^\"])"`, kde zachytávající skupina zachycuje právě ten jeden znak bez uvozovek.

Pravidlové části se dělí na jednoduché a složené. **Jednoduchá pravidlová část** reprezentuje vždy jen jeden segment zápisu pravidla, což může být například stav, neterminál nebo i řetězec znaků. Význam jednoduché pravidlové části je určen jejím typem. Podporované typy jednoduchých pravidlových částí jsou:

- čtení z pásky
- zápis na pásku
- čtení ze zásobníku
- zápis na zásobník
- stav automatu
- řetězec terminálních symbolů gramatiky
- neterminální symbol gramatiky

- oddělovač levé a pravé strany pravidla (zapisovaný symbolem `->`)
- jiný typ - pro případ, že by model potřeboval speciální vlastní typ pravidlové části

Některé jednoduché pravidlové části mohou být kvantifikované, což znamená, že lze specifikovat minimum a maximum jimi požadovaných znaků. Například pravidlová část typu *čtení z pásky* může mít minimum a maximum rovné jedné, přičemž se pak jedná o pravidlovou část *čtení právě jednoho znaku z pásky*. Nastavením minima na nulu lze povolit takzvaný **epsilon přechod**, tedy přechod z jednoho stavu automatu do druhého stavu bez čtení symbolu z pásky. Nastavením maxima na neomezený počet vznikne pravidlová část *čtení řetězce znaků z pásky*.

Složené pravidlové části jsou takové části, které obsahují jednu nebo více jednoduchých pravidlových částí a určitým způsobem rozšiřují jejich možnost zápisu. Typ složené části je vždy *složená část*. Podpůrné prostředí obsahuje implementaci dvou různých složených částí, a to *opakování jedné části* (ve zdrojovém kódu pod jménem `MultiPart`) a *opakování sjednocení částí* (`UnionMultiPart`).

Složená část *opakování jedné části* umožňuje definovat určitý počet dovolených opakování jedné části v zápisu pravidla. Kupříkladu standardní konečný automat by mohl mít pravidlovou formu definovanou následovně (jednotlivé pravidlové části kromě oddělovače uvedeny v hranatých závorkách)

[*stav*] [*čtení z pásky*(1 znak)] `->` [*stav*]

přičemž zápis takového pravidla by mohl být například

```
s1 "a" -> s2
```

Tento zápis znamená, že automat nacházející se ve stavu `s1` přečte z pásky znak `a` a přesune se do stavu `s2`. Pravidlová forma rozšířené varianty konečného automatu, která by dovolovala čtení z jedné až n pásek, by pak mohla být definována jako:

[*stav*] [*opakování*([*čtení z pásky*(1 znak)], minimum 1, maximum n)] `->` [*stav*]

Příklad zápisu pravidla odpovídajícího této pravidlové formě by pak mohl vypadat takto

```
s1 "a" "b" "c" -> s2
```

pokud n je alespoň 3. Význam takto zapsaného pravidla by byl takový, že automat nacházející se ve stavu `s1` přečte z první pásky znak `a`, z druhé pásky znak `b`, z třetí pásky znak `c` a přesune se do stavu `s2`. Rozšířená varianta konečného automatu by šla rozšířit ještě více tak, že automatu bude umožněno číst nula až m znaků současně. Pravidlová forma pro takto rozšířenou variantu konečného automatu by šla definovat jedním ze dvou následujících způsobů:

a) [*stav*] [*opakování*([*čtení z pásky*(0 až m znaků)], minimum 1, maximum n)] `->` [*stav*]
b) [*stav*] [*opakování*([*čtení z pásky*(0 až m znaků)], minimum 0, maximum n)] `->` [*stav*]

Tyto dvě pravidlové formy se liší jen minimem části *opakování čtení z pásky*. Význam obou forem je stejný, ale rozsah jejich možného zápisu se liší. Dva možné zápisy epsilon přechodu:

1) `s1 "" -> s2`
2) `s1 -> s2`

První pravidlo odpovídá oběma pravidlovým formám. Druhé pravidlo však odpovídá pouze pravidlové formě b, která díky minimu části *opakování čtení z pásky* rovnému nule dovoluje celou pravidlovou část *čtení z pásky* vynechat. To, zda autor nového modelu takovéto definice pravidel povolí, či nikoliv, je zcela na něm, jelikož lze argumentovat, že obě varianty jsou správné.

Složená část *opakování sjednocení částí* je principiálně velmi podobná části *opakování jedné části* jen s tím rozdílem, že místo jedné jednoduché části se zde může opakovat hned několik jednoduchých částí v libovolném pořadí. Příkladem použití této části je například definice pravidlové formy pro bezkontextovou gramatiku, která by mohla vypadat následovně:

[*neterminál*] `->` [*opakování sjednocení*([*neterminál*], [*terminál*], minimum 0, maximum ∞)]

Jednoduché pravidlové části mohou být takzvaně injektovatelné. **Injektovatelné pravidlové části** podporují vložení seznamu řetězců, jenž ovlivní jejich zápis a tudíž jejich regulární výrazy. Injektovatelnou pravidlovou částí je například *neterminální pravidlová část*, která slouží k načtení neterminálního symbolu v gramatice. Při načítání instance gramatiky postupuje načítač tak, že v parametrech instance přečte výčet neterminálů této gramatiky. Výčet vloží do kopie jemu dodané pravidlové formy, která vložení propaguje na všechny své odpovídající dílčí pravidlové části. To zajistí, že při procházení pravidla neterminální pravidlová část detekuje vždy jen definované neterminální symboly. Tento princip vkládání významu do pravidlových částí umožňuje podporu předem definovaného výčtu prvků dané pravidlové části, zároveň ale způsobuje, že pravidlová forma obsahující injektovatelné části je proměnná (anglicky *mutable*). Pravidlová forma, která neobsahuje injektovatelné pravidlové části, je neměnná (anglicky *immutable*).

Uživatel si pro svůj nový model může definovat libovolnou pravidlovou formu i nové pravidlové části. Při použití více složených částí s variabilním počtem opakování v pravidlové formě vedle sebe je nutné dbát na to, aby při zápisu pravidla bylo vždy jednoznačné, která část řetězce spadá pod kterou pravidlovou část. Pokud by uživatel například umístil vedle sebe *čtení z nula až n pásek* a *čtení z nula až m zásobníků*, přičemž obě tyto části by měli zápis ve tvaru `"řetězec"` bez žádného rozlišujícího prvku a bez injektování páskových a zásobníkových abeced, tak u zápisu `"a""b"` není jasné, jestli se jedná o dvě čtení z pásek, o jedno čtení z pásky a jedno čtení ze zásobníku, či o dvě čtení ze zásobníku. Jakým způsobem by načítač zpracoval zápis takovéhoto pravidla není definované, i když z principu funkce regulárních výrazů by se dalo předpokládat, že by se daný zápis zpracoval jako dvě čtení z pásek.

Novou pravidlovou část má smysl definovat v případě, že pravidla modelu obsahují konstrukci, která nejde jednoduše vyjádřit s použitím existujících pravidlových částí. Dalším důvodem je situace, kdy daná pravidlová část sice existuje, ale uživatel by rád použil odlišný zápis. Například pokud bychom měli automat s velkým počtem pásek, tak pravidlo čtoucí z třinácté pásky by obsahovalo dvanáct prázdných čtení, která by uživatel musel do zápisu pravidla všechny zapsat. Místo toho by uživatel mohl chtít vytvořit alternativní pravidlovou část pro čtení z pásky, a to takovou, kde by čtený řetězec obsahoval číselnou předponu, vyjádřující pořadí pásky. Příkladem by mohlo být

```
s1 13"a" -> s2
```

což vyjadřuje, že automat nacházející se ve stavu `s1` přečte z třinácté pásky znak `a` a přesune se do stavu `s2`. Při použití uživatelem definovaných pravidlových částí je nutné příslušně

upravit načítač instancí modelu. Minimálně je nutné upravit metodu `parseElement(...)`, která vytváří zpracovanou část pravidla, a metodu `createRule(...)`, která ze zpracovaných částí pravidla sestavuje nové pravidlo.

4.3.2 Variantní zápis pravidel

Načítání instancí podporuje variantní zápis pravidel. Tím se myslí, že jednotlivé pravidlové části na levé straně pravidla či celá pravá strana pravidla může obsahovat více než jednu variantu. Varianty lze nejlépe vysvětlit na příkladu. Nechť existuje konečný automat s těmito pravidly:

1) `s1 "a" -> s2`
2) `s1 "b" -> s2`
3) `s2 "a" -> s3`
4) `s2 "a" -> s4`

První a druhé pravidlo se liší jen v části *čtení znaku z pásky*, tudíž lze zapsat na jeden řádek zjednodušeně pomocí variantního zápisu následovně:

```
s1 "a","b" -> s2
```

Varianta existuje pouze v zápisu. Při načítání se z ní vytvoří dvě pravidla stejně, jako v nezjednodušeném zápisu. Obdobně lze vidět, že třetí a čtvrté pravidlo se liší jen ve výsledném stavu. Tuto skutečnost lze také zapsat variantním zápisem jako:

```
s2 "a" -> s3, s4
```

Rozdíl mezi použitím varianty na levé straně pravidla a pravé straně pravidla je v tom, že varianta na pravé straně pravidla se vždy vztahuje k celé pravé straně. Nechť existuje zásobníkový automat s pravidly:

1) `"1" s1 "a" -> "2" s2`
2) `"1" s1 "a" -> "3" s3`

Zápis prvního pravidla znamená, že automat nacházející se ve stavu `s1` přečte z pásky znak `a` a z vrcholu zásobníku znak `1`, přesune se do stavu `s2` a na vrchol zásobníku vloží znak `2`. Jak již bylo řečeno, variantní zápis se vždy vztahuje na celou pravou stranu, tudíž lze tyto dvě pravidla zapsat zjednodušeně jako:

```
"1" s1 "a" -> "2" s2, "3" s3
```

Nechť existuje jiný zásobníkový automat s pravidly:

1) `"1" s1 "a" -> "2" s2`
2) `"2" s1 "b" -> "2" s2`

Na těchto pravidlech si lze všimnout, že se liší na levé straně pravidla jak ve čtení ze zásobníku, tak ve čtení z pásky. Někoho by mohlo napadnout použít zjednodušený zápis a zapsat tato pravidla následovně:

```
"1","2" s1 "a","b" -> "2" s2
```

To by se však dopustil chyby, jelikož při použití variabilního zápisu na více různých místech se vytvoří pravidla z kartézského součinu všech možností. Abstraktní metoda vypočítávající n-ární kartézský součin je definovaná v obecném abstraktním načítači instancí (viz obrázek 4.4). Výsledkem uvedeného zápisu by tedy byla následující pravidla:

```
"1" s1 "a" -> "2" s2
"1" s1 "b" -> "2" s2
"2" s1 "a" -> "2" s2
"2" s1 "b" -> "2" s2
```

Variantní zápis podporuje i složená pravidlová část *opakování jedné části*, přičemž varianta se definuje vždy na jednom z opakovaných prvků. Například u pravidlové formy

[*stav*] [*opakování*([*čtení z pásky*(1 znak)], minimum 1, maximum *n*)] `->` [*stav*]

kde n je 3 lze zapsat

```
s1 "a" "b","c" "d" -> s2
```

což znamená, že automat nacházející se ve stavu `s1` přečte z první pásky znak `a`, z druhé pásky znak `b` nebo `c`, ze třetí pásky znak `d` a přesune se do stavu `s2`. Použití variantního zápisu v pravidlové části *opakování sjednocení částí* se nedoporučuje, jelikož není explicitně podporováno.

4.4 Modul rozhraní příkazové řádky

Rozhraní příkazové řádky využívá OSGi konzoli a definuje v ní nový příkaz pro specifikaci požadavků o provedení výpočtu problému.

Příkaz nese jméno `flf` jakožto zkratku názvu programu *Formal Language Framework*. Pomocí příkazu `help flf` nebo `flf -h` lze vyvolat nápovědu o použití příkazu včetně všech možný atributů. Modul rozhraní použití `Registr komponent` k dynamickému získání seznamů podporovaných modelů, problémů a algoritmů.

Činnost modulu rozhraní příkazové řádky funguje tak, že poté, co uživatel zadá příkaz, knihovna `Picocli`[1] zkontroluje správnost argumentů příkazu a pokud je vše v pořádku, tak zavolá statickou metodu `launch(...)` třídy `Spouštěč problému` (viz podkapitola 4.1), která spustí výpočet problému. Po dokončení výpočtu modul rozhraní vypíše výsledek či ho zapíše do souboru.

4.5 Komponentní moduly

V této kapitole popisuji všechny konkrétní komponenty obsažené buď v hlavním modulu nebo v jednotlivých komponentních modulech. U modulů obsahující model je vždy uvedena

[1] Picocli - dostupné na https://picocli.info/

definice pravidlové formy modelu, příklad zápisu pravidla, příklad popisu instance a příklad výstupu problému členství spuštěném na tomto modelu. U definic pravidlových forem obsahujících složené pravidlové části zapisuji minimum a maximum opakování ve zkrácené podobě po řadě jako min a max.

4.5.1 Komponenty obsažené v hlavním modulu

Přímo do hlavního modulu jsem se rozhodl umístit nejdůležitější a nejspíše nejpoužívanější komponenty podpůrného prostředí. Jedná se o problém členství, který je vždy zvolen jako výchozí problém, když uživatel žádný problém explicitně nespecifikuje. Dále jsou zde obsaženy dva obecné algoritmy schopné řešit problém členství na automatech a gramatikách.

Problém členství

Problém členství je implementován v třídě `MembershipProblem` z obrázku 4.2. Na vstupu vyžaduje problém členství jednu třídu modelu, jeden či více souborů s popisem instancí zvoleného modelu a jeden či více vstupních souborů.

Výpočet problému funguje tak, že problém s využitím pomocných metod pro načítání z abstraktní třídy `AProblem` načte instance modelu a vstupy. Dále se pro každou kombinaci instance modelu a vstupu vytvoří instance zvoleného **akceptačního algoritmu**. V rámci této práce nazývám každý algoritmus, který je schopen řešit problém členství, jako akceptační algoritmus. Po vytvoření každé instance akceptačního algoritmu je algoritmus inicializován a rovnou spuštěn. Spuštěné algoritmy se vykonávají paralelně, zatímco problém připravuje další instance algoritmů pro spuštění.

Poté, co problém spustí všechny vytvořené instance algoritmů, začne problém okamžitě sbírat výsledky algoritmů. Výsledky algoritmů je možné číst až po zavolání blokující metody `finalize()` (viz oddíl 4.2.3). Výsledky jsou setříděny tak, aby byly všechny výsledky pro jednu instanci modelu vždy u sebe. Tímto krokem zároveň končí výpočet problému. Příklady výstupů problému členství je možné vidět na obrázcích 4.6, 4.8, 4.11, 4.13 a 4.15.

Obecné akceptační algoritmy pro automaty a gramatiky

Podpůrné prostředí obsahuje dva obecné akceptační algoritmy, jejichž třídy jsou k vidění na obrázku 4.3, a to jeden pro automaty (`AutomataAcceptanceAlgorithm`) a druhý pro gramatiky (`GrammarAcceptanceAlgorithm`). Algoritmus schopný řešit problém členství pro gramatiky označuji jako akceptační i přesto, že gramatiky jsou generativní modely a algoritmy, které simulují gramatiky, by se měly nazývat generativní. Důvodem pro použití mnou zvoleného názvosloví je fakt, že i když se principiálně jedná o generativní algoritmus, tak obecný akceptační algoritmus pro gramatiky má na vstupu řetězce, o kterých je nutné rozhodnout, zda patří do jazyku gramatiky či nikoliv.

Algoritmy fungují tak, že používají obecná rozhraní modelů, pravidel a konfigurací k dosažení akceptující konfigurace. Důležité je brát v potaz, že tyto algoritmy nejsou implementovány s důrazem na efektivitu výpočtu, ale tak, aby byly schopné podpořit všechny implementované automaty a gramatiky, jejichž rozhraní odpovídá obecnému rozhraní modelu. Pokud uživatel chce pouze otestovat jím navržený model formálního jazyka, tak právě **tyto dva algoritmy dovolují uživateli se soustředit pouze na implementaci modelu a jeho vlastností** bez nutnosti se zabývat implementací specifického akceptačního či generativního algoritmu pro jeho model.

Obecný akceptační algoritmus pro automaty má na vstupu právě jednu instanci modelu a právě jeden vstup. Vstupem se myslím dostatečný počet řetězců pro všechny vstupní modely. Tedy například pro Watson-Crickův konečný automat (viz oddíl 2.2.1) je nutné mít na vstupu dva řetězce. Algoritmus nechá model vytvořit ze vstupu počáteční konfiguraci a umístí ji do fronty. Jelikož algortimus používá frontu, tak implementuje BFS[2]. Fronta konfigurací je uchovávána ve třídě `Historie konfigurací BFS`. Následně se v cyklu vezme konfigurace z přední části fronty, a pokud to není ukončující konfigurace, tak se na ni aplikují všechna možná aplikovatelná pravidla. Výstupem aplikací pravidel jsou nové konfigurace, které se zařadí do fronty. Algoritmus končí úspěšně, pokud našel akceptující konfiguraci pro daný vstupní řetězec, nebo neúspěšně, pokud vypršel čas nebo se vyprázdnila fronta. Fronta se vyprázdní, pokud už nebylo možné vytvořit aplikací pravidla žádnou novou konfiguraci.

Třída `Historie konfigurací BFS` obsahuje kromě uchovávání fronty konfigurací několik dalších funkcí. Historie si pamatuje množinu všech přidaných (neboli viděných) konfigurací. To umožňuje při přidávání konfigurace již viděnou konfiguraci odmítnout. Dále historie při přidávání nové konfigurace kontroluje, zda je konfigurace akceptující, a pokud ano, tak si ji uloží zvlášť do seznamu akceptujících konfigurací. Historie konfigurací také umí vytvořit textovou reprezentaci posloupnosti konfigurací od počáteční konfigurace do libovolné konfigurace v historii.

Obecný akceptační algoritmus pro gramatiky funguje podobně, jako ten pro automaty. Rozdílem je, že na vstupu má algoritmus pro gramatiky všechny řetězce, o kterých problém členství chce rozhodnout, zda spadají do jazyku gramatiky. Všechny řetězce má na vstupu z toho důvodu, že počáteční konfigurace gramatiky nezáleží na vstupu, ale pouze na počátečním symbolu. Tedy není nutné spouštět pro každý vstupní řetězec paralelně nový výpočet stejného generativního algoritmu, jelikož všechny výpočty generativního algoritmu na jedné instanci gramatiky budou probíhat stejně. Tento rozdíl se dále promítá i do ukončující podmínky pro algoritmus. Obecný akceptační algoritmus pro gramatiky končí, pokud nalezl akceptující konfiguraci pro každý ze vstupů, vyprázdnila se fronta konfigurací nebo vypršel čas. Problémem u generativních modelů, jako jsou gramatiky, je to, že u drtivé většiny všech gramatik v nedeterministickém generování konfigurací nikdy nedojde k tomu, že by už nebylo možné vygenerovat novou konfiguraci, a algoritmus by tedy ve většině případů končil dosažením časového limitu. Z tohoto důvodu je do algoritmu zavedena **heuristika omezení délky generativních prvků**. Uživatel může v každém požadavku o výpočet problému členství specifikovat maximální počet znaků v generativních prvcích. U gramatik se generativním prvkem myslí celá větná forma, omezen je tedy maximální počet symbolů ve větné formě. U automatů je každý zásobník samostatný generativní prvek, omezeny jsou tedy délky všech zásobníků automatu. Tato heuristika je implementována ve třídě `Historie konfigurací BFS`.

Nakonec jsou vytvořeny výsledky obecného akceptačního algoritmu pro gramatiky, a to tak, že pro každý jednotlivý vstup v tomto algoritmu je vytvořen vlastní výsledek. Pokud uživatel specifikuje parametrem `-tree`, že požaduje vygenerovat i derivační strom (obecně derivační graf viz oddíl 4.2.1), tak algoritmus ke každému úspěšnému výsledku přidá odpovídající derivační strom. Příklad derivačního stromu je možné vidět na obrázku 4.9.

4.5.2 Modul konečného automatu

V tomto modulu implementuji model konečného automatu (definice viz oddíl 2.1.2). Konečný automat zde slouží jako příklad implementace standardního automatu. Pro demon-

[2]BFS = *Breadth-first search* neboli česky „prohledávání do šířky“

straci možnosti využít vlastní implementaci konfigurace a pravidla obsahuje model konečného automatu třídy `Pravidlo konečného automatu` a `Konfigurace konečného automatu`. Proto je možné na obrázku 4.1 vidět, že třída `FiniteAutomaton` přepisuje metody (anglicky *method override*) `appyRule(...)` a `createStartConfiguration(...)`.

Pravidlová forma (viz oddíl 4.3.1) konečného automatu je definována jako

[*stav*] [*čtení z pásky*(1 znak)] `->` [*stav*]

a zápis pravidla konečného automatu vypadá následovně

```
s1 "a" -> s2
```

Příklad celého popisu instance konečného automatu lze vidět na obrázku 4.5. Popis instance konečného automatu obsahuje pouze parametry `start`, což označuje počáteční stav, a `final` značící množinu koncových stavů. Není nutné definovat abecedu ani seznam všech stavů, jelikož tyto množiny lze sestavit z definovaných pravidel. Výsledek problému členství spuštěného s konečným automatem z obrázku 4.5 je k vidění na obrázku 4.6.

```
start: nic
final: abac

nic "a" -> a, nic
nic "b","c" -> nic
a "b" -> ab
ab "a" -> aba
aba "c" -> abac
abac "a","b","c" -> abac
```

Obrázek 4.5: **Příklad popisu instance konečného automatu.** Tento automat přijímá jakýkoliv řetězec w na abecedě $\Sigma = a, b, c$, jehož podřetězcem je řetězec *abac*. Popis instance se nachází v souboru „`./inputs/fa_contains_abac_nondeterministic.txt`“.

```
Results:
|-----------------------------------------------------------------|
Instance: fa_contains_abac, Algorithm: acceptautomata

"abac" -> True: nic"abac" |- a"bac" |- ab"ac" |- aba"c" |- abac""
"abc" -> False.
"aabacb" -> True: nic"aabacb" |- a"abacb" |- a"bacb" |- ab"acb" |-
    aba"cb" |- abac"b" |- abac""
"ababbacab" -> False.
"d" -> False.
"" -> False.
"abbac" -> False.
|-----------------------------------------------------------------|
```

Obrázek 4.6: **Příklad výsledku problému členství spuštěného s konečným automatem.** Použit byl vstupní soubor „`./inputs/input_abac.txt`“. Řádek výstupu byl ručně zalomen.

4.5.3 Modul bezkontextové gramatiky

Bezkontextová gramatika (viz oddíl 2.1.3) je implementovaná jako příklad standardní gramatiky. Tento modul obsahuje třídu `Pravidlo bezkontextové gramatiky`, kterou model bezkontextové gramatiky používá místo obecné třídy `Pravidlo gramatiky`. Pro použití vlastní třídy pravidla je nutné přepsat metodu (anglicky *method override*) `applyRule(...)`, což je možné vidět na obrázku 4.1 u třídy `ContextFreeGrammar`. V metodě `applyRule(...)` je implementované využití nejlevější derivace. Nejlevější derivace znamená, že pokud větná forma obsahuje více neterminálů, na které je možné aplikovat jedno z pravidel, tak bude pravidlo aplikované vždy na nejlevější neterminál. Tímto způsobem nejlevější derivace výrazně snižuje stavový prostor výpočtu, přičemž nesnižuje vyjadřovací sílu bezkontextových gramatik. Třída `Pravidlo bezkontextové gramatiky` obsahuje dodatečnou optimalizaci k nejlevější derivaci. Tato optimalizace spočívá v přepsání metody `isApplicable(IConfiguration configuration)`, což umožňuje pravidlo označit jako aplikovatelné pouze v případě, že ho lze aplikovat na nejlevější neterminál v řetězci.

Pravidlová forma (viz oddíl 4.3.1) bezkontextové gramatiky je definována jako

$$[\textit{neterminál}] \texttt{ -> } [\textit{opakování sjednocení}([\textit{neterminál}], [\textit{řetězec terminálů}(1 - \infty \text{ symbolů})], \text{minimum } 0, \text{maximum } \infty)]$$

a zápis pravidla bezkontextové gramatiky vypadá následovně

```
A -> abcA
```

Příklad celého popisu instance bezkontextové gramatiky lze vidět na obrázku 4.7. Popis instance bezkontextové gramatiky obsahuje pouze parametry `start`, což označuje počáteční symbol, a `nonterminals` značící množinu neterminálů. Terminály není nutné definovat, protože množinu terminálů lze odvodit ze seznamu pravidel. Výčet neterminálů je obsažen proto, aby jej bylo možné vložit do injektovatelných pravidlových částí *neterminál* a *řetězec terminálů*. Díky tomu, že jsou tyto části navržené jako injektovatelné, je možné zapisovat řetězec $w \in (N \cup \Sigma)^*$ bez nutnosti použití uvozovek. Po vložení výčtu neterminálů do pravidlových částí lze snadno v řetězci w rozlišit mezi neterminály a terminály. Například když je známo, že $S, Net \in N$, pak lze řetězec $w = abcSxyzNeta$ jednoznačně interpretovat tak, že obsahuje terminály $a, b, c, x, y, z \in \Sigma$. Nevýhodou tohoto zápisu je to, že nemůže obsahovat prázdné znaky jako například mezery a terminály musí být vždy reprezentované jako právě jeden znak.

```
start: S
nonterminals: S, A, B

S -> ASB
S ->
A -> a
B -> b
```

Obrázek 4.7: **Příklad popisu instance bezkontextové gramatiky.** Tato gramatika přijímá jazyk $L = \{a^n b^n | n \geq 0\}$ a její popis instance je možné dohledat v souboru „`./inputs/test/cfg_anbn.txt`“.

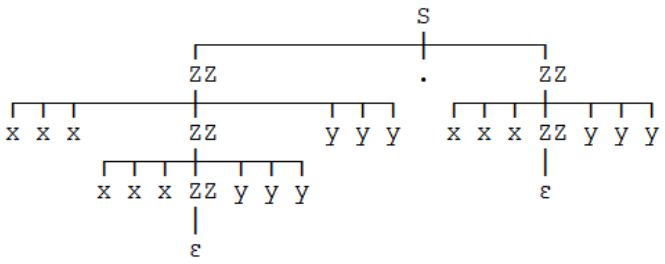
Výsledek problému členství spuštěného s bezkontextovou gramatikou z obrázku 4.7 je k vidění na obrázku 4.8. Tento výpočet byl spuštěn s parametrem `-tree`, který pro výsledky

gramatik vygeneruje krásný derivační strom. Další příklad složitějšího derivačního stromu lze vidět na obrázku 4.9.

```
Results:
|------------------------------------------------------------------|
Instance: cfg_anbn, Algorithm: acceptgrammar

"" -> True: S => ε
Derivation tree:
S
|
ε
"ab" -> True: S => ASB => aSB => aB => ab
Derivation tree:
  S
┌─┼─┐
A S B
| | |
a ε b
"aaabbb" -> True: S => ASB => aSB => aASBB => aaSBB => aaASBBB =>
    aaaSBBB => aaaBBB => aaabBB => aaabbB => aaabbb
Derivation tree:
      S
┌─────┼─────┐
A     S     B
|  ┌──┼──┐  |
a A  S  B  b
   |  ┌┼┐  |
   a A S B b
     | | |
     a ε b
"aab" -> False.
"bba" -> False.
|------------------------------------------------------------------|
```

Obrázek 4.8: **Příklad výsledku problému členství spuštěného s bezkontextovou gramatikou.** Použit byl vstupní soubor „`./inputs/input_anbn.txt`“. Řádek výstupu byl ručně zalomen.



Obrázek 4.9: **Příklad vygenerovaného derivačního stromu.** Derivační strom řetězce `xxxxxxyyyyyy.xxxyyy` vygenerovaného bezkontextovou gramatikou ze souboru „`./inputs/cfg_3x3y.txt`“. Tato gramatika má tři terminály $\Sigma = \{x, y, .\}$, neterminály $N = \{S, ZZ\}$ a pravidla $P = \{S \to ZZ.ZZ, ZZ \to \varepsilon, ZZ \to xxxZZyyy\}$. Generuje jazyk $L = \{x^{3n}y^{3n}.x^{3m}y^{3m} | n \geq 0, m \geq 0\}$. Pro spuštění výpočtu byl použit vstupní soubor „`./inputs/input_3x3y.txt`“. Řádek výstupu byl ručně zalomen.

4.5.4 Modul n-zásobníkového m-páskového automatu

n-zásobníkový m-páskový automat je pokročilý model, který jsem navrhnul. Principiálně funguje stejně jako zásobníkový automat (viz oddíl 2.1.4) pouze s několika rozdíly:

- automat obsahuje 0 až ∞ vstupních pásek
- automat obsahuje 0 až ∞ zásobníků
- v pravidle může číst z libovolných pásek 0 až ∞ znaků, epsilon přechody jsou tedy povoleny
- v pravidle může číst z libovolných zásobníků 0 až ∞ znaků

Tento automat přijímá řetězce na vstupních páskách vyprázdněním všech pásek a přesunem do koncového stavu. Vyprázdnění zásobníků není požadováno. Na abecedy vstupních pásek a zásobníků se dá pohlížet jako na dvě množiny Σ a Γ, přičemž abeceda každé konkrétní pásky je podmnožinou abecedy Σ a abeceda každého zásobníku je podmnožinou abecedy Γ. Jelikož model může obsahovat dva či více zásobníků a je známo, že zásobníkový automat s dvěma zásobníky je turingovsky úplný, tak i n-zásobníkový m-páskový automat je turingovsky úplný.

Model n-zásobníkového m-páskového automatu jsem implementoval především pro demonstraci a otestování plného rozsahu možností načítačů instancí. Model používá výchozí obecné třídy pro reprezentaci pravidel, konfigurací, stavů, vstupních pásek a zásobníků.

Pravidlová forma (viz oddíl 4.3.1) n-zásobníkového m-páskového automatu je definována jako

$$[\textit{opakování}([\textit{čtení ze zásobníku}(0-\infty \text{ znaků})],\ \min 0, \max \infty)]\ [\textit{stav}]$$
$$[\textit{opakování}([\textit{čtení z pásky}(0-\infty \text{ znaků})],\ \min 0, \max \infty)]\ \texttt{->}$$
$$[\textit{opakování}([\textit{zápis na zásobník}(0-\infty \text{ znaků})],\ \min 0, \max \infty)]\ [\textit{stav}]$$

a zápis pravidla n-zásobníkového m-páskového automatu vypadá následovně

```
"" "ABC" s1 "abc" "" "x2" -> "X" "111" s2
```

Příklad celého popisu instance n-zásobníkového m-páskového automatu lze vidět na obrázku 4.10. Popis instance n-zásobníkového m-páskového automatu na rozdíl od popisu instance konečného automatu obsahuje navíc parametr `stack start symbols`, který po řadě udává počáteční symboly na zásobnících. Výsledek problému členství spuštěného s n-zásobníkovým m-páskovým automatem z obrázku 4.10 je k vidění na obrázku 4.11.

```
start: s1
final: f
stack start symbols: "X", "X"

      s1 "a" -> "B" s1
"X""X" s1 ""  -> f
"B"    s1 "b" -> """C" s2
"B"    s2 "b" -> """C" s2
"" "C" s2 "c" -> s3
"" "C" s3 "c" -> s3
"X""X" s3 ""  -> f
```

Obrázek 4.10: **Příklad popisu instance n-zásobníkového m-páskového automatu.** Tento automat přijímá jazyk $L = \{a^n b^n c^n | n \geq 0\}$. Popis instance se nachází v souboru „`./inputs/nsnta_anbncn.txt`“.

```
Results:
|-------------------------------------------------------------------------------|
Instance: nsnta_anbncn, Algorithm: acceptautomata

"abc" -> True: "X""X"s1"abc" |- "BX""X"s1"bc" |- "X""CX"s2"c" |- "X""X"s3"" |- """"f""
"aabbcc" -> True: "X""X"s1"aabbcc" |- "BX""X"s1"abbcc" |- "BBX""X"s1"bbcc" |-
    "BX""CX"s2"bcc" |- "X""CCX"s2"cc" |- "X""CX"s3"c" |- "X""X"s3"" |- """"f""
"ab" -> False.
"aaabbb" -> False.
"" -> True: "X""X"s1"" |- """"f""
"aabc" -> False.
"abbc" -> False.
"abcc" -> False.
"cbba" -> False.
|-------------------------------------------------------------------------------|
```

Obrázek 4.11: **Příklad výsledku problému členství spuštěného s n-zásobníkovým m-páskovým automatem.** Použit byl vstupní soubor „`./inputs/input_anbncn.txt`“. Řádek výstupu byl ručně zalomen.

4.5.5 Modul Watson-Crikova konečného automatu

V tomto modulu je implementován Watson-Crickův konečný automat (viz oddíl 2.2.1). Model obsahuje vlastní třídu `Konfigurace Watson-Crickova konečného automatu`. Tato implementace přidává dodatečnou podmínku z definice akceptující konfigurace, a to konkrétně že konfigurace může být akceptující pouze tehdy, pokud řetězce na vstupních páskách měly před zahájením výpočtu stejnou délku. Dále všechny konfigurace, které tuto podmínku nesplňují, jsou automaticky označeny za ukončující, což šetří čas výpočtu. Model Watson-Crickova automatu tedy okamžitě odmítne všechny vstupy tvořené řetězci odlišné délky.

Pravidlová forma (viz oddíl 4.3.1) Watson-Crikova konečného automatu je definována jako

$$[stav]\ [opakování([čtení\ z\ pásky(0-\infty\ \text{znaků})],\ \text{min 2, max 2})]\ \texttt{->}\ [stav]$$

což by bylo ekvivalentní definici

$$[stav]\ [čtení\ z\ pásky(0-\infty\ \text{znaků})]\ [čtení\ z\ pásky(0-\infty\ \text{znaků})]\ \texttt{->}\ [stav]$$

a zápis pravidla Watson-Crikova konečného automatu vypadá následovně

```
s1 "a" "bbb" -> s2
```

Příklad celého popisu instance Watson-Crikova konečného automatu lze vidět na obrázku 4.12. Popis instance konečného automatu obsahuje parametr `relation`, který značí komplementární relaci. Aby bylo zajištěno, že komplementární relace je vždy symetrická, tak je implementována jako množina dvouprvkových množin místo množiny dvojic. Uživateli tedy stačí definovat vztah v jednom směru. Symetrický zápis je také povolen. Na obrázku 4.12 je pro dvojici (A, T) použit symetrický zápis a pro dvojici (C, G) jednostranný zápis. Zápis komplementární relace ρ z obrázku 4.12 je tedy interpretován jako $\rho = \{(A,T),(T,A),(C,G),(G,C)\}$. Výsledek problému členství spuštěného s Watson-Crikovým konečným automatem z obrázku 4.12 je k vidění na obrázku 4.13.

Při implementaci Watson-Crickova konečného automatu byla největší překážkou kontrola korektnosti popisu instance. Pravidla musí odpovídat definované komplementární množině, a to se prokázalo jako složitý úkol. Pro kontrolu správnosti jsem popsal vztah v následujícím odstavci a na jeho základě implementoval algoritmus, který kontroluje, zda každé

```
start: s
final: s
relation: (A, T), (T, A), (C, G)

s"A""T"->s
s"T""A"->s
s"C""G"->s
s"G""C"->s
```

Obrázek 4.12: **Příklad popisu instance Watson-Crikova konečného automatu.** Tento automat přijímá jazyk $L = \Sigma^*$ na abecedě $\Sigma = A, T, C, G$. Přijme ale pouze takové vstupy, kde je řetězec $w1$ na první pásce komplementární s řetězcem $w2$ na druhé pásce. Popis instance se nachází v souboru „`./inputs/wcfa_DNA_simple.txt`“.

```
Results:
|-------------------------------------------------------------------------------|
Instance: wcfa_DNA_simple, Algorithm: acceptautomata

"ATCG" "TAGC" -> True: s"ATCG""TAGC" |- s"TCG""AGC" |- s"CG""GC" |- s"G""C" |- s""""
"A" "T" -> True: s"A""T" |- s""""
"A" "C" -> False.
"ATC" "TCG" -> False.
"AAAAAAAAAA" "TT" -> False.
"GTAGCC" "CATCGG" -> True: s"GTAGCC""CATCGG" |- s"TAGCC""ATCGG" |- s"AGCC""TCGG" |-
    s"GCC""CGG" |- s"CC""GG" |- s"C""G" |- s""""
|-------------------------------------------------------------------------------|
```

Obrázek 4.13: **Příklad výsledku problému členství spuštěného s Watson-Crikovým konečným automatem.** Použit byl vstupní soubor „`./inputs/input_DNA.txt`“. Řádek výstupu byl ručně zalomen.

jedno pravidlo je zapsané správně. Kontrola správnosti celé množiny pravidel je však ponechána na uživateli, jelikož se jedná o algoritmicky velmi složitý problém.

O řetězcích $w_1, w_2 \in \Sigma^*$ z každého pravidla $q(w_1, w_2) \to t \in R$, kde $q, t \in Q$, vyjádřených jako

$$w_1 = a_1, a_2, ..., a_n \text{ a } w_2 = b_1, b_2, ..., b_m \text{ kde}$$
$$|w_1| = n, |w_2| = m; a_i, b_j \in \Sigma \text{ pro } 1 \leq i \leq n, 1 \leq j \leq m$$

musí platit, že

$$\exists k \in < min(0, n-m),\ max(0, n-m) > :$$
$$(a_{i+max(0,k)}, b_{i+|min(k,0)|}) \in \rho \text{ pro } 1 \leq i \leq min(n, m)$$

Jinak řečeno, existuje posun k z intervalu nula až rozdíl délky řetězců w_1 a w_2 takový, že když se o tento posun posunou indexy znaků delšího řetězce a vytvoří se tak podřetězec stejné délky jako kratší z řetězců w_1 a w_2, tak mezi tímto podřetězcem a kratším z řetězců musí platit, že znaky na odpovídajících indexech jsou komplementární dle relace ρ. Pokud jsou řetězce w_1 a w_2 stejně dlouhé, tak $k = 0$ a porovnání párů znaků se provede tedy přímo mezi řetězci w_1 a w_2.

Výše zmíněný vztah zaručuje, že každé jedno pravidlo koresponduje s komplementární relací ρ. Tato skutečnost ovšem nic neříká o tom, zda celá množina pravidel odpovídá komplementární relaci ρ. Na zachování vztahu mezi množinou pravidel a relací ρ musí uživatel dbát při definování instancí Watson-Crickova konečného automatu.

4.5.6 Modul obecného skákajícího konečného automatu

Model obecného skákajícího konečného automatu (viz oddíl 2.2.2) je implementován třídou `GJFA`, kterou je možné vidět na obrázku 4.1. Pravidlová forma (viz oddíl 4.3.1) obecného skákajícího konečného automatu je definována jako

$$[stav]\ [čtení\ z\ pásky(0-\infty\ \text{znaků})]\ \texttt{->}\ [stav]$$

a zápis pravidla konečného automatu vypadá následovně

```
s1 "abc" -> s2
```

Příklad celého popisu instance obecného skákajícího konečného automatu lze vidět na obrázku 4.14. Parametr `leftmost-heuristic: true` povoluje pro tuto instanci využití heuristiky nejlevějšího čtení popsané níže.

```
start: s
final: f
leftmost-heuristic: true

s "a" -> s
s "" -> f
f "abc" -> f
```

Obrázek 4.14: **Příklad popisu instance obecného skákajícího konečného automatu.** Tento automat přijímá takový řetězec w na abecedě $\Sigma = \{a, b, c\}$, který odpovídá rozšířenému rekurzivnímu regulárnímu výrazu $(a^*(aba^*(?1)^*c)^*a^*)^*\$$, kde $(?1)$ symbolizuje opakující se vnořený výskyt první zachytávající skupiny (vnější závorky) a $ symbolizuje konec řetězce. Tento regulární výraz je možné vyzkoušet v nástroji regex101 dostupném na stránce https://regex101.com/. Popis instance se nachází v souboru „`./inputs/gjfa_abc.txt`“.

Výsledek problému členství spuštěného s obecným skákajícím konečným automatem z obrázku 4.14 je k vidění na obrázku 4.15.

Při implementaci obecného skákajícího konečného automatu jsem se setkal s velkou překážkou v podobě silného nedeterminismu tohoto modelu. Páska, která je součástí konfigurace automatu, je v obecné implementaci realizována jako vstupní řetězec a číselný kurzor, který ukazuje na pozici v tomto řetězci. Při čtení se pak jen kurzor posouvá o jednu pozici doprava. Aktuální obsah pásky je vyjádřen jako podřetězec celého řetězce pásky od pozice kurzoru. Pokud by konfigurace obecného skákajícího konečného automatu byla reprezentována stejně, tak by kvůli možnosti nedeterministického skoku bylo při každé aplikaci pravidla vygenerováno x konfigurací, kde x je aktuální délka řetězce na pásce. To by pro vstup o délce n znamenalo $n!$ (faktoriál) vygenerovaných konfigurací, pokud bychom brali v úvahu, že každé pravidlo čte právě jeden znak. Dalším problémem při použití této reprezentace pásky je fakt, že obecný skákající konečný automat může přečíst každý symbol na vstupní pásce pouze jednou, takže by byla potřeba dodatečná informace o přečtených symbolech.

Tento problém jsem vyřešil vytvořením nové třídy `Skákající páska`. Tato páska si při vytvoření uloží vstupní řetězec do dvou proměnných a vůbec neuchovává pozici čtecí hlavy. V jedné proměnné se uchovává vstupní řetězec v nezměněné podobě, a v druhé je vždy aktuální řetězec. Čtení z pásky je realizováno tak, že se najdou všechny pozice na pásce, kde je možné přečíst řetězec zapsaný na levé straně pravidla. Pro každou tuto pozici se

```
Results:
|-------------------------------------------------------------|
Instance: gjfa_abc, Algorithm: acceptautomata

"abc" -> True: s"abc" |- f"abc" |- f""
"abca" -> True: s"abca" |- s"abc" |- f"abc" |- f""
"aaaa" -> True: s"aaaa" |- s"aaa" |- s"aa" |- s"a" |- s"" |- f""
"aaabcaa" -> True: s"aaabcaa" |- s"aabcaa" |- s"abcaa" |-
    s"abca" |- s"abc" |- f"abc" |- f""
"aabbaacca" -> False.
"aababcaca" -> True: s"aababcaca" |- s"ababcaca" |- s"ababcca" |-
    s"ababcc" |- f"ababcc" |- f"abc" |- f""
"" -> True: s"" |- f""
"aaabc" -> True: s"aaabc" |- s"aabc" |- s"abc" |- f"abc" |- f""
"abcbc" -> False.
"abac" -> True: s"abac" |- s"abc" |- f"abc" |- f""
|-------------------------------------------------------------|
```

Obrázek 4.15: **Příklad výsledku problému členství spuštěného s obecným skákajícím konečným automatem.** Použit byl vstupní soubor „`./inputs/input_abc.txt`“. Dva řádky výstupu byly ručně zalomeny.

vytvoří nová konfigurace s kopií aktuální pásky a řetězec se na dané pozici přečte. Čtení řetězce z dané pozice znamená, že se tento řetězec vymaže z dané pozice v aktuálním řetězci na pásce. Díky této implementaci je možné snížit počet vygenerovaných konfigurací v každém kroku výpočtu z délky aktuálního řetězce pouze na počet výskytů čteného řetězce v aktuálním řetězci na pásce.

Ve většině případů by implementace popsaná výše byla dostatečným zrychlením výpočtu. Nechť existuje obecný skákající konečný automat M s abecedou $\Sigma = \{a\}$ a s jediným pravidlem $\mathtt{s}a \to \mathtt{s}$, přičemž $\mathtt{s} \in F$ je počáteční i koncový stav. Je zřejmé že automat M přijímá jazyk $L(M) = \{a^n | n \geq 0\}$. Pro jakýkoliv řetězec $w \in \Sigma^*$ pak platí, že počet vygenerovaných konfigurací při aplikaci pravidla $\mathtt{s}a \to \mathtt{s}$ na řetězec w je rovný $|w|$, jelikož existuje právě tolik možných pozic, v nichž lze číst znak a v řetězci w. Tím celkový počet vygenerovaných konfigurací pro přijetí řetězce w je $|w|!$, zatímco standardní konečný automat ekvivalentní automatu M, který by řetězec w četl zleva znak po znaku, by zvládl přijmout řetězec w pouze s $|w|$ vygenerovanými konfiguracemi. Tento fakt mě přiměl k zamyšlení, zda by bylo možné vytvořit heuristiku využívající **nejlevější čtení**, tedy situaci, kdy by obecný skákající konečný automat četl při aplikaci pravidla pouze nejlevější výskyt čteného řetězce v řetězci na pásce.

Obecně je pravděpodobné, že použití nejlevějšího čtení snižuje vyjadřovací sílu obecného skákajícího konečného automatu. Toto vyvozuji z faktu, že aplikací nejlevějšího čtení by automat z obrázku 4.14 ztratil schopnost přečíst řetězec *abca* a všechny další řetězce, ve kterých se vyskytuje znak a po začátku podřetězce *abc*, protože by nikdy nemohl přečíst druhý znak a na konci řetězce před přečtením podřetězce *abc*. Problém je tedy v aplikaci nejlevějšího čtení na pravidlo $\mathtt{s}a \to \mathtt{s}$, které čte řetězec a, jenž je vlastním podřetězcem řetězce čteného v jiném pravidle (zde v pravidle $\mathtt{f}abc \to \mathtt{f}$).

Díky tomuto zjištění mohu implementovat heuristiku nejlevějšího čtení, která povoluje použít nejlevější čtení pouze pro pravidla, jejichž čtený řetězec není vlastním podřetězcem (viz podkapitola 2.1) řetězce čteného v jiném pravidle. Důležité je dodat, že u pravidel čtoucích ε se neprovádí žádné čtení, jelikož je vždy možné přejít do výsledného stavu. Díky

této heuristice je možné akceptovat každý řetězec z jazyka $L(M) = \{a^n | n \geq 0\}$ se stejným počtem vygenerovaných konfigurací jako u standardního konečného automatu. Pro povolení této heuristiky je nutné přidat do popisu instance parametr `leftmost-heuristic: true`.

4.5.7 Modul problému abecedy

Problém abecedy odpovídá na otázku, jak vypadají abecedy použité v určité instanci modelu. Na vstupu má právě jednu třídu modelu a jeden či více souborů s popisy instancí zvoleného modelu. Problém abecedy pak pro každou z těchto instancí modelu sestaví použité abecedy. Tento problém jsem navrhnul z toho důvodu, že jak je vidět na obrázcích 4.5, 4.7, 4.10, 4.12 a 4.14, tak popisy instancí neobsahují definice vstupních abeced. Problém abecedy tedy může sloužit k detekci chyb zápisu pravidel instance. Například může problém abecedy pomoci uživateli ověřit, zda neudělal překlep v některém z pravidel v popisu instance, která se chová jinak, než by očekával.

Všechny algoritmy určené k řešení problému abecedy musí implementovat rozhraní `IAlphabetAlgorithm` (viz obrázek 4.3). Podpůrné rozhraní obsahuje jeden algoritmus implementující toto rozhraní, a to obecný algoritmus abecedy (`AlphabetAlgorithm`). Tento algoritmus funguje pro všechny automaty a abecedy. Pro automaty vrací seznam stavů a samostatné abecedy každé vstupní pásky a každého zásobníku. Pro gramatiky vrací abecedy terminálních a neterminálních symbolů. Příklad výstupu problému abecedy spuštěného na instanci n-zásobníkového m-páskového automatu lze vidět na obrázku 4.16

```
Results:
|----------------------------------------|
Instance: nsnta_anbncn, Algorithm: alphabet
States: f, s1, s2, s3
Tape 1: a, b, c
Stack 1: B, X
Stack 2: C, X
|----------------------------------------|
```

Obrázek 4.16: **Příklad výsledku problému abecedy spuštěného s n-zásobníkovým m-páskovým automatem.** Pro vygenerování výstupu byla použita instance automatu z obrázku 4.10, jejíž popis je možné najít v souboru „`./inputs/nsnta_anbncn.txt`“.

Kapitola 5

Zhodnocení práce

V této kapitole bych měl provést zhodnocení modulárnosti systému. Jeden z požadavků, které by mělo řešení splňovat je, že by mělo být možné zkompilovat nový modul bez nutnosti úprav či opětovné kompilace ostatních modulů. Toto kritérium je splněno a testoval jsem jej na kompilaci modulu Watson-Crickova konečného automatu.

Dále v rámci zhodnocení modulárnosti systému by stálo vyhodnotit to, jak náročné je vlastně implementovat nový model do stávající aplikace. Obtížnost přidání nového modelu jsem měřil jako čas, který mi zabere implementovat nový model. Měřil jsem čas od vytvoření nové kopie vzoru nového modulu až do prvního úspěšného spuštění a získání korektních výsledků, a to včetně času potřebného k navrhnutí a sepsání příkladu jednoduchého popisu instance. Čas implementace jsem měřil u dvou modulů, které jsem implementoval jako poslední, a to u modulu Watson-Crickova konečného automatu a u modulu obecného skákajícího konečného automatu.

Implementace modulu Watson-Crickova konečného automatu mi zabrala hodinu a 45 minut. Základní implementaci lze provést v časovém rozmezí dvaceti až čtyřiceti minut. Základní implementací se myslí podědění z nadřazené třídy obecného modelu automatu, přepsání abstraktních metod z nadřazené třídy a definování pravidlové formy. Zbytek času mi zabrala implementace vlastní konfigurace pro Watson-Crickův automat, úprava metody `applyRule(...)` aplikující pravidlo na konfiguraci, načítání parametru komplementární relace a implementace kontroly správnosti zápisu jednotlivých pravidel (viz oddíl 4.5.5).

Modul obecného skákajícího konečného automatu jsem zvládl implementovat za hodinu a 24 minut. Velkou část tohoto času jsem strávil laděním chyby, kvůli které aplikace vracela špatné výsledky. Při řešení této chyby mě napadlo využít heuristiku nejlevějšího čtení, kterou jsem popsal v oddílu 4.5.6. Výše zmíněný čas nezahrnuje dobu použitou k implementaci této heuristiky, jelikož první úspěšné výsledky jsem získal i bez této heuristiky. Návrh a sepsání popisu instance a vstupů pro ni mi z času potřebného na implementaci zabralo sedm minut.

Z těchto měření se dá doba potřebná k implementaci jednoduchého modelu bez složitých vlastností odhadovat na dvacet až čtyřicet minut, a čas potřebný k implementaci pokročilého modulu na zhruba hodinu a půl. Tyto hodnoty ukazují, že přidání nového modulu není nijak zvláštně obtížené, dokonce bych řekl, že se znalostmi tohoto podpůrného prostředí i v celku jednoduché. Samozřejmě se nejedná o velmi exaktní měření. Aby bylo možné dobu potřebnou k implementaci nového modulu stanovit lépe, tak by bylo potřeba, aby skupina různých vývojářů implementovala desítky či stovky modulů. Takovéto měření by však zcela přesahovalo rozsah této práce.

Efektivitu či rychlost implementovaných obecných akceptačních algoritmů nemá smysl měřit, jelikož, jak je uvedeno v kapitole Implementace v oddílu 4.5.1, tyto algoritmy byly navrženy a implementovány s důrazem na to, aby je bylo možné obecně spustit nad každým podporovaným automatem a gramatikou. Cenou za tuto obecnost byla právě efektivita těchto algoritmů.

Kapitola 6

Závěr

Cílem práce bylo navrhnout a implementovat podpůrné prostředí pro pokročilé modely formálních jazyků. Provedl jsem průzkumu existujících systémů a zhodnotil jsem jejich případné rozšíření. Jelikož jsem si zvolil cestu vlastního řešení, tak bylo navíc nutné vybrat modulární systém pro výslednou aplikaci.

Dále bylo požadováno, aby výsledná modulární aplikace implementovala alespoň dva různé modely formálních jazyků a tři algoritmy nad těmito modely. Ve své práci popisuji pět implementovaných modelů, tři algoritmy a dva problémy.

Na závěr bych chtěl shrnout několik možných rozšíření výsledné aplikace. Očividnou možností je přidání celé řady dalších pokročilých modelů formálních jazyků a algoritmů nad těmito modely. Kromě modelů a algoritmů je možné navrhnout i mnoho transformačních a optimalizačního problémů, jako například transformace instance automatu na instance odpovídající gramatiky, či odstranění zbytečných pravidel v gramatice. Dalším možným rozšířením by bylo přidání nových uživatelských či aplikačních rozhraní. Příkladem by mohlo být grafické uživatelské rozhraní pro zobrazení instancí modelů či rozhraní webové služby pro možnost nasazení aplikace na server. Myslím si, že by bylo možné vypsat a vypracovat několik bakalářských a diplomových prací, které by moje podpůrné prostředí rozšiřovaly zmíněnými způsoby.

Těmito fakty bych tedy chtěl konstatovat, že zadání práce bylo splněno v kompletním, či možná i v lehce širším rozsahu. Skutečnost, že se mi povedlo navrhnout rozhraní modelů, algoritmů a dalších částí aplikace tak, že dva algoritmy jsou schopné řešit problém členství na všech podporovaných modelech, mi přijde jako velký úspěch.

Literatura

[1] Czeizler, E., Czeizler, E., Kari, L. a Salomaa, K. Watson-crick automata: Determinism and state complexity. In: *Descriptional Complexity of Formal Systems - 10th International Workshop, DCFS 2008*. Leden 2008, s. 121–133. ISBN 978-0-919013-56-8. Dostupné z: https://www.researchgate.net/publication/221467475_Watson-crick_automata_Determinism_and_state_complexity.

[2] Eclipse Foundation. *Equinox OSGi* [online]. (n.d.) [cit. 2023-04-25]. Dostupné z: https://www.eclipse.org/equinox/.

[3] Hammer, J. *Watson-Crick Models for Formal Language Processing*. Brno, CZ, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Dostupné z: https://www.fit.vut.cz/study/thesis/20427/.

[4] Kozen, D. C. *Automata and Computability*. Springer Verlag, 1997. ISBN 0-387-94907-0. Dostupné z: https://www.fit.vut.cz/research/publication/6177.

[5] Meduna, A. *Automata and Languages: Theory and Applications [Springer, 2000]*. Springer Verlag, 2005. 892 s. ISBN 978-1-85233-074-3. Dostupné z: https://www.fit.vut.cz/research/publication/6177 a https://vdoc.pub/documents/automata-and-languages-theory-and-applications-29ff5piv9bl0.

[6] Meduna, A. a Zemek, P. Chapter 17 Jumping Finite Automata. In: *Regulated Grammars and Automata*. Springer New York, Leden 2014, s. 567–585. DOI: 10.1007/978-1-4939-0369-6_17. ISBN 978-1-4939-0368-9. Dostupné z: https://www.researchgate.net/publication/300813104_Chapter_17_Jumping_Finite_Automata.

[7] Microsoft. *Prism Library Documentation* [online]. 2015-2022 [cit. 2023-04-26]. Dostupné z: https://prismlibrary.com/docs/.

[8] OSGi Working Group. *OSGi Working Group: The Dynamic Module System for Java* [online]. (n.d.) [cit. 2023-04-25]. Dostupné z: https://www.osgi.org/.

[9] Rodger, S. H. et al. *JFLAP* [online]. 2005-2023 [cit. 2023-04-25]. Dostupné z: https://www.jflap.org/.

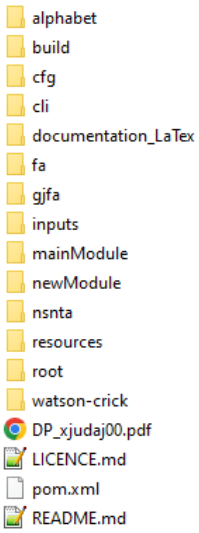
[10] VMware, Inc.. *The Spring Framework* [online]. 2023 [cit. 2023-04-26]. Dostupné z: https://spring.io/.

[11] Češka, M. *Teoretická informatika, učební text FIT VUT v Brně* [online]. 2002 [cit. 2022-01-15]. Dostupné z: http://www.fit.vutbr.cz/study/courses/TIN/public/Texty/TIN-studijni-text.pdf.

Příloha A

Obsah paměťového média

Struktura paměťového média je viditelná na obrázku A.1.



Obrázek A.1: **Struktura paměťového média.**

Popis jednotlivých složek a souborů:

- alphabet — Složka obsahující zdrojové soubory modulu problému abecedy.
- build — Složka build obsahuje všechny použité knihovny, které jsou nutné pro manuální spuštění. V této složce se dále nachází podsložka configuration, která obsahuje konfigurační soubor pro spuštění OSGi systému config.ini.
- cfg — Složka obsahující zdrojové soubory modulu bezkontextové gramatiky.
- cli — Složka obsahující zdrojové soubory modulu rozhraní příkazové řádky.
- documentation_LaTex — Složka obsahuje zdrojové soubory tohoto textu včetně všech použitých obrázků. PDF verzi textu lze z tohoto obsahu vygenerovat.
- fa — Složka obsahující zdrojové soubory modulu konečného automatu.
- gjfa — Složka obsahující zdrojové soubory modulu obecného skákajícího konečného automatu.

- inputs — Složka inputs obsahuje příklady popisu instancí modelů, příklady vstupních souborů pro tyto instance modelů a podsložku test, ve které se nachází popisy instancí modelů sloužící jako data pro testy.
- mainModule — Složka obsahující zdrojové soubory hlavního modulu aplikace.
- newModule — Tato složka slouží jako vzor nového modulu.
- nsnta — Složka obsahující zdrojové soubory modulu n-zásobníkového m-páskového automatu.
- resources — V této složce se nachází konfigurační soubor „logging.properties“, který slouží k nastavení ladících výstupů programu.
- root — Složka slouží jako zastřešení celé aplikace ve vývojovém prostředí Eclipse[1] a obsahuje spouštěcí konfiguraci pro sestavení všech modulů pomocí tohoto vývojového prostředí.
- watson-crick — Složka obsahující zdrojové soubory modulu Watson-Crickova konečného automatu.
- DP_xjudaj00.pdf — Tento soubor je PDF verze textu práce.
- LICENCE.md — Soubor LICENCE.md obsahuje licenci pro použití aplikace.
- pom.xml — Soubor pom.xml je konfigurační soubor pro nástroj Maven, který slouží k sestavení celé aplikace. Každý jednotlivý modul obsahuje také svůj vlastní konfigurační soubor pom.xml. Kořenový soubor pom.xml se na tyto dílčí konfigurační soubory odkazuje.
- README.md — Soubor README.md obsahuje informace o aplikaci, příklady použití aplikace a návod k jejímu sestavení a spuštění.

Každá složka modulu aplikace v sobě obsahuje podsložku target, ve které se nachází zkompilovaný binární soubor modulu s příponou `.jar`. Soubor config.ini obsahuje odkazy na tyto soubory, což umožňuje celou aplikaci spustit přímo ze složky build dle instrukcí v souboru README.md.

[1] Vývojové prostředí Eclipse dostupné z https://www.eclipse.org/downloads/

Příloha B

Možnosti a argumenty příkazové řádky

Rozhraní příkazové řádky poskytuje možnost ovládat program pomocí příkazů zadávaných do konzole systému OSGi. Rozhraní definuje jeden příkaz `flf`. Zadávání argumentů a možností příkazu pak vypadá stejně, jakoby uživatel v OSGi konzoli spouštěl program se jménem `flf`.

V této příloze je uveden kompletní podrobný seznam všech dostupných možností a argumentů příkazu `flf`. Rozdíl mezi možnostmi a argumenty je ten, že možnosti jsou volitelné a argumenty jsou povinné. V seznamu jsou nejdříve popsány argumenty, a po nich abecedně všechny možnosti. Pokud má nějaký parametr hodnotu, tak mezi parametr a jeho hodnotu lze zapsat mezeru nebo znak `=`. Nápovědu k použití programu obsahující tento seznam možností lze zobrazit i přímo v programu zadáním příkazu `help flf` nebo `flf -h`. Na obrázku B.1 je možné vidět hlavičku nápovědy programu se seznamem všech možností. Modul rozhraní příkazové řádky do nápovědy dynamicky přidává i seznam všech načtených modelů, problémů a algoritmů. Tento seznam lze vidět na obrázku B.2.

```
Usage: flf [-h] [--all] [--multiline] [--tree] [-a=ALGORITHM] [--div=DIVIDER] [--maxGS=SIZE] [-o=OUTPUT_FILE] [-p=PROBLEM]
           [-t=TIMEOUT] [--threads=THREADS] [-i=INPUT_FILE]... [-m=INSTANCE_DESC]... MODEL...
```

Obrázek B.1: **Hlavička nápovědy programu.**

Seznam možností a argumentů programu:

- Argument: `MODEL...` — Pomocí argumentů programu uživatel volí jednu či více tříd modelu.

  Třída modelu se specifikuje pomocí identifikátoru modelu z obrázku B.2. Počet zvolených tříd by měl odpovídat specifikaci zvoleného problému.

  Příklad: `flf cfg` = zvolení bezkontextové gramatiky.

- Možnost: `-a, --algorithm=ALGORITHM` — Volba algoritmu.

  Algoritmus se specifikuje pomocí identifikátoru algoritmu z obrázku B.2. Výchozí algoritmus je určen zvoleným problémem s ohledem na zvolenou třídu modelu.

  Příklad: `-a acceptautomata` = zvolení obecného akceptačního algoritmu pro automaty.

- Možnost: `-all` — Pokud bylo pro jeden vstup nalezeno více možných řešení, použitím této možnosti budou vypsána všechna nalezená řešení.

V opačném případě program vypisuje pouze jedno řešení na jeden vstup. Podpora této možnosti závisí na zvoleném problému.

- Možnost: `--div=DIVIDER` — Volba oddělovače vstupů ve vstupních souborech.

  Pokud zvolený model přijímá vstupy (jakýkoliv vícepáskový automat), tak se jednotlivé vstupy ve vstupních souborech oddělují řádkem obsahujícím pouze oddělovač. Oddělovač může být libovolný řetězce. Každý řádek jednoho vstupu pak reprezentuje jeden vstupní řetězec pro jednu vstupní pásku. Výchozí oddělovač je `#`. Podpora této možnosti závisí na zvoleném problému.

  Příklad: `--div=<--->` = zvolení řetězce `<-->` jakožto oddělovače vstupů.

- Možnost: `-h, --help` — Zobrazí nápovědu o použití programu.

  Příklad: `flf -h`

- Možnost: `-i, --input=INPUT_FILE` — Volba vstupního souboru pro zvolený problém. Možnost lze uvést vícekrát.

  Každý problém sám definuje, jakým způsobem využívá vstupní soubory. Problém členství používá tyto soubory jako řetězce, o kterých je nutné rozhodnout, zda patří do jazyků zvolených instancí modelů. Neslouží pro výběr souborů s popisy instancí modelů.

  Příklad: `-i input.txt` = volba vstupního souboru `input.txt`.

- Možnost: `-m, -d=INSTANCE_DESC` — Volba souboru s popisem instance zvolené třídy modelu. Možnost lze uvést vícekrát.

  Počet podporovaných a vyžadovaných souborů s popisem instance se může lišit u každého problému. Většinou problémy vyžadují alespoň jeden soubor s popisem instance.

  Příklad: `-d ./inputs/cfg\_3x3y.txt`

- Možnost: `--maxGS=SIZE` — Nastavení maximální délky generativních prvků (viz oddíl 4.5.1).

  Výchozí hodnota je 100.

  Příklad: `--maxGS 10` = nastaví maximální délku generativních prvků na 10.

- Možnost: `--multiline` — Specifikuje, že každý vstupní soubor má být brán jako jeden dlouhý vstup.

  Podpora této možnosti závisí na zvoleném problému.

- Možnost: `-o, --output=OUTPUT_FILE` — Specifikuje soubor, do kterého bude uložen výsledek výpočtu.

  Pokud není žádný soubor specifikován, tak bude výsledek vypsán na standardní výstup.

  Příklad: `-o output.txt` = výsledky budou zapsány do souboru „output.txt“.

- Možnost: `-p, --problem=PROBLEM` — Volba problému.

  Problém se specifikuje pomocí identifikátoru problému z obrázku B.2. Výchozím problémem je problém členství.

  Příklad: `-p alphabet` = zvolení problému abecedy.

- Možnost: `-t, --timeout=TIMEOUT` — Určení délky časového limitu ve vteřinách pro výpočet každé instance algoritmu.

  Výchozí hodnota je 10 vteřin. Zadáním nuly či záporné hodnoty je možné vypnout funkci časového limitu, což se silně nedoporučuje!

  Důležité je zmínit, že maximální celkový čas výpočtu je roven násobku této hodnoty počtem jednotlivých instancí algoritmů.

  Příklad: `-t 30` = nastaví časoví limit pro výpočet jedné instance algoritmu na třicet vteřin.

- Možnost: `--threads=THREADS` — Určuje maximální počet vláken dostupných pro paralelní výpočet algoritmů

  Výchozí počet dostupných vláken je o jedna menší, než počet logických procesorů systému, na kterém je aplikace spuštěna.

  Příklad: `--threads=4` = omezení počtu vláken na čtyři vlákna.

- Možnost: `--tree` — Přidá vygenerované derivační stromy (či obecně grafy) do úspěšných výsledků simulace gramatik

  Podpora této možnosti závisí na zvoleném problému.

```
Loaded models:
  cfg     Context Free Grammar
  fa      Finite Automaton
  gjfa    General Jumping Finite Automaton, you can add 'leftmost-heuristic: true' as a parameter to instance description to
            allow left-most read heuristic on suitable rules for execution speed-up
  nsmta   N-Stack M-Tape Automaton, turing complete
  wcfa    Watson-Crick Finite Automaton, expects that each input has exactly 2 tapes

Loaded problems:
  alphabet     constructs instance's alphabets from declared instance rules. Helpful when diagnosing semantic errors in instance
                 descriptions
  membership   Membership Problem, tests whether instances of a model accept inputs, i.e. whether the inputs are members of
                 language defined by a instance of a model

Loaded algorithms:
  acceptautomata   Automata Acceptance Algorithm, default for membership problem with automata, uses BFS, tests whether an
                     automaton accepts inputs, i.e. whether the inputs are members of language defined by an automaton
  acceptgrammar    Grammar Acceptance Algorithm, default for membership problem with grammars, BFS based, tests whether a
                     grammar accept inputs, i.e. whether the inputs are members of language defined by a grammar
  alphabet         default general algorithm for alphabet problem, supports automata and grammars
```

Obrázek B.2: **Dynamicky tvořený seznam načtených komponent.** V levém sloupci jsou identifikátory komponent, v pravém sloupci jsou komponenty stručně popsány.

Příloha C

Návod na sestavení a spuštění programu

Tato příloha obsahuje návod na sestavení a spuštění aplikace spolu s příklady použití. Program byl implementován a testován na systému Windows 10, ale jelikož je jazyk Java multiplatformní, tak by měl program fungovat i na ostatních operačních systémech. Příklady použití lze nalézt v souboru README.md v kořenovém adresáři na paměťovém médiu.

C.1 Sestavení aplikace

Požadavky pro sestavení:
– Nainstalovaný vývojářský balíček Oracle Java Development Kit verze 16 či vyšší
– Nainstalovaný program Maven verze 3.8.1 či vyšší

Pro sestavení všech modulů aplikace zadejte příkaz `mvn package` v kořenovém adresáři aplikace. Alternativně je možné nový modul sestavit s využití binárních souborů již přeložených modulů.

C.2 Spuštění aplikace

Požadavky pro spuštění:
– Nainstalované prostředí Oracle Java Runtime Environment verze 16 či vyšší

Aplikaci je možné spustit pomocí následujících kroků:

1. Přesuňte se do složky `./build`.
2. Zapněte OSGi systém příkazem:
   `java -jar org.eclipse.osgi_3.17.100.v20211104-1730.jar -console`
3. Po zapnutí OSGi by měly všechny moduly zobrazit ladící informaci o tom, že jsou zapnuty.
4. Nyní je možné zadat příkaz `flf` s parametry popsanými v příloze B. Nápovědu pro použití programu je také možné zobrazit příkazem `flf -h`.

C.3 Přidání nového modulu

Nový modul je do aplikace možné přidat pomocí následujících kroků:

1. Vytvořte kopii složky `./newModule` a pojmenujte ji jménem vašeho nového modulu.
2. Zvolte název nového modulu v souboru `pom.xml`, a poté upravte název Java balíčku (angl. *package*) ve zdrojovém kódu odpovídajícím způsobem.
3. Přidejte implementaci komponent či rozhraní do vaše modulu.
4. *V případě komponentního modulu:* Registrujte všechny nové komponenty v aktivátoru modulu pomocí registru komponent.
5. *Pro sestavení modulu pomocí nástroje Maven:*
   Přidejte `<module>maven-identifikátor</module>` mezi moduly do souboru `pom.xml` v kořenové složce aplikace. Řetězec `maven-identifikátor` nahraďte názvem nového modulu z kroku 2.
6. Sestavte projekt pomocí instrukcí v podkapitole C.1. Alternativně si můžete vytvořit vlastní příkaz čistě pro sestavení nového modulu s využitím binárních souborů existujících modulů.
7. *Pro spuštění aplikace s novým modulem:*
   Přidejte `../jméno-složky/target/maven-identifikátor-0.0.1-SNAPSHOT@start,` do souboru `./build/configuration/config.ini`. Řetězec `jméno-složky` nahraďte jménem složky z kroku 1. Řetězec `maven-identifikátor` nahraďte názvem nového modulu z kroku 2.
8. Nyní je možné spustit aplikaci spolu s vaším novým modulem pomocí instrukcí v podkapitole C.2.