



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název:	Návrh a implementace knihovny pro parsování bezkontextových gramatik
Student:	Patrik Valkovi
Vedoucí:	Ing. Jan Trávní ek
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Nastudujte formální jazyky a gramatiky, se zam ením na bezkontextové gramatiky.

Nastudujte algoritmy pro transformace gramatik s d razem na algoritmy pro tvorbu Chomského normální formy.

Za použití existujících algoritm navrhnete a implementujete knihovnu, která na základ dodané gramatiky převede seznam lexikálních symbolů na abstraktní syntaktický strom a

- reprezentuje libovolnou gramatiku,
- transformuje obecnou bezkontextovou gramatiku na bezkontextovou gramatiku v Chomského normální formě,
- parsuje obecnou bezkontextovou gramatiku,
- abstraktní syntaktický strom v Chomského normální formě převede na abstraktní syntaktický strom odpovídající gramatice před transformací,
- umožní použití atributových gramatik.

Volba implementa ní platformy je sou ástí práce.

Implementaci ádné dokumentujte a otestujte.

Vytvořte příklady demonstrující schopnosti knihovny. Příklady konzultujte s vedoucím práce.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
ředitel katedry

V Praze dne 1. listopadu 2017

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
KATEDRA SOFTWAREVÉHO INŽENÝRSTVÍ



Bakalářská práce

Návrh a implementace knihovny pro parsování bezkontextových gramatik

Patrik Valkovič

Vedoucí práce: Ing. Jan Trávníček

10. května 2018

Poděkování

Chtěl bych poděkovat své rodině, přítelkyni a přátelům za jejich nejen psychickou podporu během studia i během psaní této práce.

Také bych chtěl poděkovat mému vedoucímu práce Ing. Janu Trávníčkovi za vedení, konzultace, cenné rady a připomínky, které mi během psaní této práce poskytl. Nebýt jeho aktivity v předchozích semestrech, tato práce by nevznikla. Dále bych chtěl poděkovat Ing. Miroslavu Hrončokovi za konzultace, code review a pomoc s technickými záležitostmi.

Nakonec bych chtěl poděkovat Ing. Elišce Šestákové a doc. Ing. Janu Janouškovi, Ph.D. za jejich přístup ve výuce předmětů, ze kterých tato práce vychází.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 10. května 2018

.....

České vysoké učení technické v Praze
Fakulta informačních technologií

© 2018 Patrik Valkovič. Všechna práva vyhrazena.

Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Valkovič, Patrik. *Návrh a implementace knihovny pro parsování bezkontextových gramatik*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Cílem práce je vytvořit knihovnu, která striktně oddělí syntaktickou a sémantickou část zpracování strukturovaného textu při zachování snadného použití a jednoduchosti. Práce se zaměřuje na parsování bezkontextových gramatik v jazyce Python. Pro implementaci byl zvolen Cocke-Younger-Kasami algoritmus z důvodu největší robustnosti v oblasti bezkontextových gramatik. Pro zjednodušení práce knihovna implementuje transformace gramatik do Chomského normální formy i jejich opačnou verzi nad parsovacím stromem. Tím knihovna poskytuje univerzální nástroj pro parsování.

Knihovna byla úspěšně implementovaná a publikována. Funkčnost knihovny je demonstrována na lambda kalkulu, jenž je parsován a interpretován.

Klíčová slova gramatiky, parsování, Chomského normální forma, Cocke-Younger-Kasami algoritmus, Python, lambda kalkulus

Abstract

The goal of this thesis is to develop library that strictly separate syntactic and semantic part of the parsing process. Library is suppose to be simple and easy to use. Library parsing process uses context-free grammars and Cocke-Younger-Kasami algorithm, because of it's versatility. Library is developed in Python programming language.

To simplify parsing process, the library implements transformations into Chomsky normal form. Moreover, it also implements backward transformations of the parsed tree. For that particular reasons, library provides complex parsing tool.

The library was successfully implemented and published. The functionality of the library is demonstrated on lambda calculus interpreter, which functionality is to parse and interpret lambda calculus.

Keywords grammars, parsing, Cocke-Younger-Kasami algorithm, Chomsky normal form, Python, lambda calculus

Obsah

Úvod	1
Cíle práce	3
1 Teoretický základ	5
1.1 Formální jazyky a gramatiky	5
1.2 Bezkontextové gramatiky	8
1.3 Formální překlady	14
1.4 Atributované gramatiky	15
1.5 Chomského normální forma	18
2 Operace s gramatikami	21
2.1 Problémy transformací	21
2.2 Převod do Chomského normální formy	22
2.3 Cocke-Younger-Kasami algoritmus	27
3 Analýza existujících řešení a návrh knihovny	31
3.1 Požadavky	31
3.2 Volba platformy a programovacího jazyka	34
3.3 Existující řešení	34
3.4 Návrh knihovny	38
4 Realizace	43
4.1 Příprava prostředí	43
4.2 Reprezentace gramatik	44
4.3 Transformace a parsování	47
4.4 Testování	50
5 Demonstrace knihovny na příkladech	51
5.1 Parser aritmetických výrazů	51

5.2	Parser regulárních výrazů	53
5.3	Interpret lambda kalkulu	54
	Závěr	59
	Literatura	61
	A Obrazová příloha	65
	B Seznam použitých zkratk	73
	C Obsah příloženého CD	75

Seznam zdrojových kódů

2.1	Pseudokód algoritmu odstraňující negenerující symboly	23
2.2	Pseudokód algoritmu odstraňující nedostupné symboly	24
2.3	Pseudokód algoritmu hledající neterminály generující ε	24
2.4	Pseudokód algoritmu odstraňující pravidlo	25
2.5	Pseudokód algoritmu odstraňující ε pravidla	25
2.6	Pseudokód algoritmu odstraňující jednoduchá pravidla	26
2.7	Pseudokód CYK algoritmu	29
3.1	Parsování slov „hello“ a „world“ knihovnou Pyrser [33, Introduction section]	37
5.1	Definice atributovaného neterminálu	52
5.2	Pravidlo <i>MultipleDivide</i> \rightarrow <i>MultipleDivide/Number</i>	52
5.3	Definice aritmetické gramatiky	53
5.4	Proces parsování aritmetických výrazů	53
5.5	Implementace metody <i>get</i> neterminálu <i>Concat</i>	54
5.6	Generování tokenů symbolizující proměnnou	56
5.7	Terminál reprezentující číslo	56

Seznam obrázků

1.1	Ukázka derivačního stromu	10
1.2	Derivační stromy pro nejednoznačné slovo	11
1.3	Rozdělení bezkontextových jazyků	14
2.1	Ukázka gramatiky představující „if statement“	22
2.2	Očekávaný a reálný syntaktický strom pro „if statement“	22
2.3	Kroky CYK algoritmu	29
3.1	Diagram modulů a jejich závislost	39
3.2	Třídy v modulu reprezentující gramatiku	39
3.3	Rozhraní třídy reprezentující gramatiku	39
3.4	Rozšíření gramatiky metodou Pipes and Filters	40
3.5	Rozhraní třídy Rule	40
3.6	Rozhraní pro tvorbu AST	41
5.1	Ukázka běhu parseru regulárních výrazů	55
5.2	Ukázka běhu interpretu lambda kalkulu	58
A.1	Ukázka překladové gramatiky	65
A.2	Vyhodnocení atributů u atributované gramatiky	66
A.3	Výpočet vlastností pro třídu Rule	66
A.4	Reprezentace syntaktického stromu pro neomezené gramatiky	67
A.5	Kompletní návrh reprezentující gramatiku	68
A.6	Hierarchy vyjímek v modulu grammpy	68
A.7	Class diagram implementace modulu grammpy	69
A.8	Class diagram implementace modulu grammpy-transforms	70
A.9	Class diagram pravidel v modulu grammpy-transforms	70
A.10	Class diagram neterminálů v modulu grammpy-transforms	71

Úvod

Během studia jsem opakovaně narážel na problém zpracování strukturovaného textu, ať už to byla validace uživatelského vstupu nebo práce se strukturovanými formáty jako jsou regulární výrazy, XML nebo JSON. Bohužel jsem nenašel žádný univerzální nástroj, který by byl snadno rozšiřitelný a dovolil definovat vlastní pravidla pro parsovaný text.

Zpracování strukturovaného textu (tzv. parsování) je jednou z hlavních činností, které dnes po počítačích požadujeme [24]. Nejedná se pouze o zpracování dokumentů, jako již zmiňovaný XML nebo JSON, ale také ověření uživatelských vstupů, práce s regulárními výrazy a v neposlední řadě také zpracování programovacích jazyků. Ačkoliv existuje řada nástrojů pro zpracování strukturovaného textu (například GCC), jsou zpravidla určeny pro specifické činnosti (kompilace programovacích jazyků v případě GCC). Jejich rozšíření v mnoha případech není vůbec možné (například MSVC kompilér [30]) nebo je obtížné.

V rámci této práce bude navržena knihovna, která by oddělila syntaktickou analýzu od sémantiky a tím dovolila snadnou rozšiřitelnost, a to i v případě existujících programů či knihoven, z této knihovny vycházejících. Knihovna bude tvořit univerzální platformu, která je nezávislá na použité metodě parsování a tím dosahuje maximální flexibility.

Pro parsování strukturovaného textu existuje několik známých algoritmů, které by měly být podporovány. Tato práce se zabývá primárně Cocke-Younger-Kasami algoritmem (dále jen CYK) z důvodu jeho obecnosti. Pro aplikování CYK algoritmu musí být data ve správném formátu, toho lze jednoznačně a deterministicky docílit a popis transformací je taktéž součástí této práce.

Práce je rozdělena do tří částí. První z nich popisuje teoretické poznatky, definuje operace požadované knihovnou a jejich vlastnosti. Jedná se o první dvě kapitoly. Druhá část (3. a 4. kapitola) je implementační a zabývá se návrhem, implementací a testováním řešení. V poslední části resp. kapitole jsou předkládány složitější příklady a diskutovány možnosti dalšího rozšíření knihovny.

Cíle práce

Cílem práce je implementovat knihovnu poskytující univerzální platformu pro proces parsování. Dále je cílem práce nastudovat formální gramatiky a navrhnout jejich vhodnou reprezentaci s ohledem na jejich použití při parsovacím procesu. Při návrhu musí být brány v úvahu běžné metody parsování z důvodu možných budoucích rozšíření knihovny. Významnou částí práce je nastudování, implementace a otestování CYK algoritmu současně s převodem gramatiky na Chomského normální formy. Zvláštní pozornost je věnována tvorbě abstraktního syntaktického stromu u CYK algoritmu a jeho zpětným transformacím z Chomského normální formy. V neposlední řadě je důležitým bodem práce demonstrovat fungování knihovny na složitějších příkladech, jako jsou jednoduché programovací jazyky.

Teoretický základ

V následující kapitole jsou nejdříve rozebrány základní termíny, které jsou použity v práci. Následuje základní popis existujících algoritmů, kde je CYK algoritmus popsán podrobněji a to včetně transformací do Chomského normální formy (dále jen CNF).

1.1 Formální jazyky a gramatiky

Pro popis strukturovaného textu se využívají formální jazyky [28]. Počátky sahají do 50. let 20. století, kdy Noam Chomsky vytvořil na základě poznatků Alana Turinga, Axele Thuea a Emila Posty definici formální gramatiky. Formální gramatiky se využívají především v teoretické informatice pro popis struktury programovacích jazyků a výpočtových modelů [19].

Vzhledem k tomu, že knihovna bude pracovat se základními elementy teorie jazyků (konkrétně reprezentace gramatik), považujeme za nutné na úvod uvést základní definice z důvodu jejich sjednocení, které se v různých publikacích mohou lišit. Definice vycházejí z materiálů Gerharda Jägera a Jamese Rogerse [23].

Symbol je elementární, dále nedělitelný objekt. Může se jednat o písmeno, číslo, ale i tón, frekvenci či jinou námi definovanou entitu.

Abeceda je konečná množina symbolů.

Slovo je sekvence symbolů z abecedy.

Formální jazyk je množina slov. Množina může být i nekonečná.

Formální gramatika umožňuje popsat formální jazyk konečným výrazem bez toho, aniž bychom jej museli definovat výčtem [28]. Pro definici gramatiky budeme potřebovat následující pojmy:

Terminál je symbol formálního jazyka. Dále v textu budeme množinu terminálů (či terminálních symbolů) označovat jako T , prvky této množiny malými písmeny.

Neterminál je symbol abecedy, která je rozdílná od abecedy terminálů. Abecedy terminálů a neterminálů jsou navzájem disjunktní. Množinu neterminálů (nebo také neterminálních symbolů) označujeme jako N , prvky této množiny velkými písmeny.

Počáteční symbol je konkrétní symbol z množiny neterminálů. Někdy se počáteční symbol označuje i jako symbol startovací. Nebude-li v textu uvedeno jinak, budeme jako počáteční symbol uvažovat neterminál S .

Epsilon je speciální symbol, který označuje slovo nulové délky. Dále v textu značíme jako ε .

Iterace je opakování symbolu (resp. slova) 0 až neomezeně mnohokrát. Značíme x^* (například $a^* = \{\varepsilon, a, aa, \dots, a^n\}, n \geq 0$).

Pravidlo je prvek kartézského součinu $(T \cup N)^* \times (T \cup N)^*$. V práci budeme pravidla psát ve tvaru $\alpha \rightarrow \beta$, kde (α, β) je prvek zmíněného kartézského součinu. Zápis chápeme jako „ α může být nahrazeno β “.

Gramatika je čtveřice $G = (N, T, S, R)$, kde R je množina pravidel gramatiky a S je počáteční symbol.

Pro další práci budeme potřebovat definovat pojmy související s převodem gramatik a prací s nimi [28].

Přímá derivace je nahrazení $\gamma\alpha\delta$ slovem $\gamma\beta\delta$, kde $\alpha, \beta, \gamma, \delta \in (T \cup N)^*$. Ve zbytku práce budeme předpokládat pouze derivace nad konkrétní gramatikou. Pro přímou derivaci nad gramatikou musí v gramatice existovat pravidlo ve tvaru $\alpha \rightarrow \beta$. Přímou derivaci budeme značit jako $\gamma\alpha\delta \Rightarrow^1 \gamma\beta\delta$.

Nultá derivace je reflexivní uzávěr přímé derivace. Každý prvek je v relaci nulté derivace sám se sebou a značíme $\alpha \Rightarrow^0 \alpha$

K -tá derivace je opakované použití přímé derivace. Výsledek $k-1$ derivace využijeme pro výpočet k -té derivace zřetěžením s přímou derivací. Označujeme jako $\alpha \Rightarrow^k \beta$ a definujeme indukci.

$$(\alpha \Rightarrow^k \beta) \Leftrightarrow ((\alpha \Rightarrow^{k-1} \gamma) \wedge (\gamma \Rightarrow^1 \beta))$$

Derivace je v práci chápána jako tranzitivní a reflexivní uzávěr přímé derivace. Budeme značit jako $\alpha \Rightarrow \beta$ a jedná se o ekvivalentní zápisu $\alpha \Rightarrow^i \beta$ pro $i \geq 0$.

Větná forma je řetězec α vygenerovaný gramatikou $G = (N, T, S, R)$, jestliže existuje derivace $S \Rightarrow \alpha, \alpha \in (T \cup N)^*$.

Věta je větná forma, která obsahuje pouze terminální symboly. Dále v práci je věta a slovo jazyka považováno za synonyma.

Generovaný jazyk L gramatikou $G = (N, T, S, R)$ je takový jazyk, jenž obsahuje všechny věty generované gramatikou.

$$L(G) = \{w : w \in T^*, \exists S \Rightarrow w\}$$

Ekvivalentní gramatiky jsou gramatiky G_1 a G_2 , které generují stejný jazyk. Formálně $L(G_1) = L(G_2)$.

Noam Chomsky rozdělil gramatiky do čtyř kategorií [28]. Se zvyšující se kategorií se zvyšuje komplexita gramatiky a tím i její obecnost. Tyto kategorie se označují jako *typ 0*, *typ 1*, *typ 2* a *typ 3*, kde *typ 0* je z nich nejobecnější [8]. Jazyky generované konkrétnějšími skupinami gramatik jsou podmnožinou jazyků generovaných gramatikami obecnějšími. Jazyky, které tyto gramatiky popisují, se nazývají *rekurzivně spočetné* (pro *Typ 0* gramatiky), *kontextové*, *bezkontextové* a *regulární* (pro *Typ 3* gramatiky). V některých zdrojích toto pojmenování „zdědily“ i samotné gramatiky, proto budeme v dále textu označovat synonymem

- *regulární gramatiky* jako gramatiky typu 3,
- *bezkontextové gramatiky* jako gramatiky typu 2,
- *kontextové gramatiky* jako gramatiky typu 1,
- *neomezené gramatiky* jako gramatiky typu 0.

Jednotlivé typy gramatik se mezi sebou liší tvarem pravidla. Z původní definice kartézského součinu $(T \cup N)^* \times (T \cup N)^*$ povoluje každá ze skupin pouze podmnožinu pravidel.

- Regulární gramatiky povolují pouze pravidla typu $A \rightarrow a, A \rightarrow aB$ a eventuálně pravidlo $S \rightarrow \varepsilon$, pokud se počáteční symbol S nevyskytuje na pravé straně pravidla.
- Bezkontextové gramatiky povolují pravidla typu $A \rightarrow (T \cup N)^*$.

- Kontextové gramatiky povolují pravidla typu $\gamma A \delta \rightarrow \gamma \alpha \delta$, kde $\alpha \in (T \cup N)(T \cup N)^*$, $\gamma, \delta \in (T \cup N)^*$ a eventuálně $S \rightarrow \varepsilon$, pokud počáteční symbol S není na pravé straně pravidla.
- Neomezené gramatiky povolují pravidla typu

$$(T \cup N)^* X (T \cup N)^* \rightarrow (T \cup N)^*$$

kde X je neterminál

Všimněme si, že bezkontextová gramatika není nutně gramatikou kontextovou, nicméně jazyk generovaný libovolnou bezkontextovou gramatikou lze popsat gramatikou kontextovou. Ke každé kategorii byl přiřazen i model, který je schopen všechny jazyky dané kategorie rozpoznat. Dále v textu bude zmíněn pouze zásobníkový automat, který je přiřazen k bezkontextovým gramatikám [28]. Zbývající modely jsou vzhledem k zaměření práce nepodstatné.

1.2 Bezkontextové gramatiky

Dále se v práci zaměříme pouze na gramatiky bezkontextové. Důvod je ten, že regulární gramatiky mají malou vyjadřovací schopnost a nedovedou popsat běžné jazykové konstrukce. Pro příklad uveďme párování složených závorek pro jazyky založených na syntaxi jazyka C.

Důkaz, že regulární jazyky nejsou schopny popsat párování závorek [36]:

$$L = \{ \{ \}^n; n \geq 0 \}$$

1. Předpokládáme, že jazyk L je regulární. Pak na základě pumpping lemmatu [31] musí platit:

$$(\exists p \geq 1) (\forall w \in L) |w| \geq p$$

\Rightarrow

$$(\exists x, y, z \in T^*) (w = xyz \wedge |xy| \leq p \wedge |y| \geq 1 \wedge (\forall k \geq 0) xy^k z \in L)$$

2. Ukážeme, že platí negace této vlastnosti. Neboli, že

$$(\forall p \geq 1) (\exists w \in L) [|w| \geq p \wedge (\forall x, y, z \in T^*)$$

$$((w = xyz \wedge |xy| \leq p \wedge |y| \geq 1) \Rightarrow (\exists k \geq 0) xy^k z \notin L)]$$

Tím dostaneme spor s předpokladem, že jazyk L je regulární. Důkaz neregularity jazyka L tak bude hotov.

- a) $\forall p \geq 1$ volíme větu w tak, aby $w \in L \wedge |w| \geq p : w = \{ \}^{p+1}$.

b) Najdeme všechny rozklady xyz pro zvolenou větu w tak, aby

$$w = xyz \wedge |xy| \leq p \wedge |y| \geq 1$$

$$\begin{aligned} x &= \{^r & r &\geq 0 \\ y &= \{^s & s &\geq 1, r + s \leq p \\ z &= \{^t\}^{p+1} & t &\geq 0, r + s + t = p + 1 \end{aligned}$$

c) Volíme k tak, aby $k \geq 0 \wedge xy^kz \notin L$:

$$k = 0 : xy^kz = xy^0z = \{^r\{^t\}^{p+1} = \{^{r+t}\}^{p+1} \notin L$$

protože $r + t = p + 1 - s \wedge s \geq 1 \Rightarrow r + t < p + 1$.

d) Dostali jsme spor s předpokladem, že jazyk L je regulární jazyk. Jazyk L tedy není regulární.

□

Na druhé straně parsování kontextových jazyků je příliš náročné. To je způsobeno tím, že kontextové jazyky jsou nedeterministické a z toho přímo vyplývá exponenciální složitost parsování [27]. Nepoužití kontextových jazyků s sebou nese komplikace, protože většina jazyků využívá vlastnosti kontextových gramatik. Pro příklad uveďme deklaraci proměnné před jejím použitím [25].

Drtivá většina dnešních parserů je založených na bezkontextových gramatikách [37]. Ty mají dostatečnou abstrakci pro popis běžně používaných konstrukcí programovacích jazyků. Omezení, které bezkontextové gramatiky mají, a díky kterým nelze popsat všechny jazykové konstrukce, se řeší přidáním kódem přímo v parseru, dodatečným zpracováním výstupu parseru nebo odklonem od formální definice gramatiky a její modifikací pro účely parsování konkrétního programovacího jazyka. Pro příklad uveďme deklaraci proměnné před jejím použitím, řešeno tabulkou symbolů během procesu parsování, nebo if-else problém [1] řešený v případě LL parserů úpravou LL parsovací tabulky (viz kapitola 1.2.3) [2].

Bezkontextové gramatiky můžeme dále rozdělit na několik skupin [28]. Každá ze skupin má své požadavky na podobu gramatiky a z toho vyplývající dopady na samotný proces parsování. Pro jednotlivé skupiny jsou známy efektivní algoritmy, které disponují nejnižší možnou složitostí. Nemusí se vždy jednat o jinou třídu složitosti, ale například i složitost asymptotickou či průměrnou. Dále uvedeme skupiny, na které se bezkontextové gramatiky dělí, s jejich požadavky a způsoby parsování.

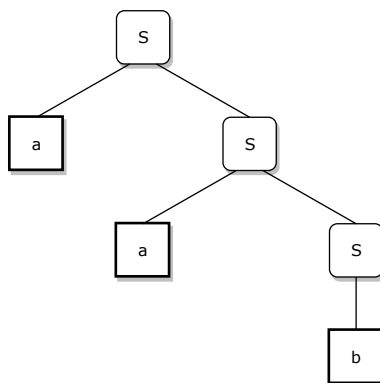
1.2.1 Jednoznačné bezkontextové gramatiky

Jednoznačné gramatiky jsou podmnožinou bezkontextových gramatik.

Derivační strom pro bezkontextové jazyky je orientovaný strom, který reprezentuje syntaktickou strukturu větné formy podle formální gramatiky. Přechod od rodiče k potomkům reprezentuje použití přímé derivace. Formálně musí mít derivační strom tyto vlastnosti:

- Uzly derivačního stromu jsou ohodnoceny terminálním symbolem, neterminálním symbolem nebo symbolem ε .
- Kořen stromu je ohodnocen počátečním symbolem gramatiky.
- Jestliže má uzel alespoň jednoho potomka, potom je ohodnocen neterminálním symbolem (viz kapitola 1.1).
- Jestliže n_1, n_2, \dots, n_k jsou bezprostřední následovníci uzlu n , který je ohodnocen symbolem A , a tyto uzly jsou zleva doprava ohodnoceny symboly A_1, A_2, \dots, A_k , pak $A \rightarrow A_1 A_2 \dots A_k$ je pravidlo gramatiky.
- Listy derivačního stromu tvoří zleva doprava větnou formu v gramatice G , která je výsledkem derivačního stromu.

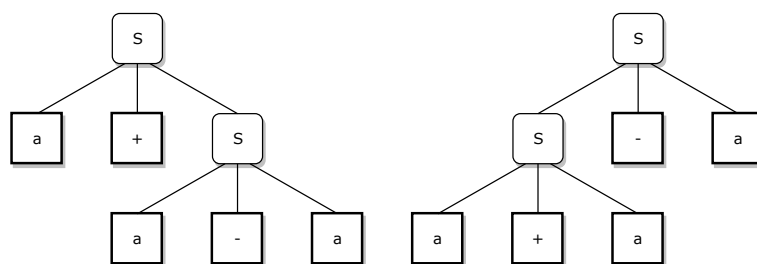
Například pro gramatiku $G = (\{S\}, \{a, b\}, S, \{S \rightarrow aS, S \rightarrow b\})$ a slovo $w = aab$ vypadá derivační strom následovně.



Obrázek 1.1: Ukázka derivačního stromu

Nejednoznačné slovo je takové slovo, pro které existuje v gramatice několik různých derivačních stromů. Pro příklad uveďme gramatiku obsahující sčítání a odčítání $G = (\{S\}, \{a, +, -\}, S, \{S \rightarrow S + S, S \rightarrow S - S, S \rightarrow a\})$ kde díky nejednoznačnosti není pevně dána přednost operací u slova $w = a + a - a$ (viz obrázek 1.2).

Jednoznačná gramatika je taková gramatika, která nemá v generujícím jazyce ani jedno nejednoznačné slovo.



Obrázek 1.2: Derivační stromy pro nejednoznačné slovo

1.2.2 Deterministické bezkontextové gramatiky

Deterministické gramatiky, nebo také $LR(k)$ gramatiky, jsou gramatiky ekvivalentní s modelem deterministického zásobníkového automatu. Jedná se o podmnožinu jednoznačných gramatik [28]. Jak již bylo řečeno, zásobníkový automat je model, který byl přiřazen právě k bezkontextovým gramatikám jako jejich ekvivalent – dokážou popsat stejnou množinu jazyků. Následující popis vychází z materiálu [28].

Deterministický stavový automat je pětice $M = (Q, T, \delta, q_0, F)$. Q je množina vnitřních stavů, T je vstupní abeceda (ekvivalent terminálů), δ je zobrazení $Q \times X \rightarrow Q$, q_0 je počáteční stav automatu ($q_0 \in Q$) a F je množina koncových stavů. Automat na základě aktuálního stavu a dalšího znaku na vstupu přechází do stavu jiného, a to podle zobrazení δ . Tento proces končí až ve chvíli, kdy je přečteno celé slovo. Pokud není pro vstupní symbol definována přechodová funkce, proces končí a říkáme, že automat slovo nepřijal. Pokud je po přečtení celého slova automat v jednom z koncových stavů (množina F), potom tvrdíme, že automat slovo přijal.

Nedeterministický stavový automat je deterministický stavový automat, pouze s rozdílnou definicí zobrazení δ . Pro jeho nedeterministickou verzi je zobrazení δ definováno jako $Q \times T \rightarrow 2^Q$, kde 2^Q je potenční množina. To znamená, že na základě stavu a následujícího znaku vstupního slova může automat přejít do více stavů současně. Pro další znak slova se provádí stejný postup pro každý jednotlivý stav, a to nezávisle na ostatních stavech. Nedeterministický stavový automat slovo přijímá, jestliže alespoň jedna z větví skončila v koncovém stavu.

Nedeterministický zásobníkový automat je automat definovaný sedmicí $K = (Q, T, Z, \delta, q_0, Z_0, F)$. Od stavového automatu se liší abecedou zásobníku Z , počátečním symbolem zásobníku $Z_0 \in Z$ a pravidly δ tvaru $Q \times (T \cup \{\varepsilon\}) \times \theta_S \rightarrow Q \times \theta_E$ kde $\theta_S, \theta_E \in Z^*$. Přejít probíhá na základě aktuálního stavu, dalšího znaku vstupního slova na základě libovolného explicitního

počtu znaků na zásobníku θ_S . Tyto znaky jsou ze zásobníku odebrány, automat se přesune do nového stavu a vloží na zásobník libovolný (i nulový) počet znaků θ_E .

Stejně jako u nedeterministického stavového automatu, paralelní přechody jsou na sebe navzájem nezávislé, tj. operace se zásobníkem u jednoho přechodu neovlivní stav zásobníku u zbylých přechodů. Automat končí ve chvíli, kdy není definována přechodová funkce, nebo kdy je přečteno celé vstupní slovo. Automat má dvě možnosti, jak může slovo přijmout. Stejně jako u nedeterministického stavového automatu, nedeterministický zásobníkový automat slovo přijímá, jestliže jedna z větví skončí v koncovém stavu. Automat také slovo přijímá, pokud alespoň jedna větev odebrala ze zásobníku všechny symboly a přečetla celé vstupní slovo. O tom, kterým způsobem nedeterministický zásobníkový automat slovo přijímá, se musíme rozhodnout při definici automatu.

Deterministický zásobníkový automat je zásobníkový automat, pro který existuje v každém stavu maximálně jeden přechod, a to pro libovolný vstupní symbol i pro libovolné slovo na zásobníku. Nedeterminismus u zásobníkového automatu je způsoben situací, při níž existují přechody vycházející ze stejného stavu na stejný vstupní symbol (resp. symbol ε), a sekvence symbolů θ_{S1} jednoho přechodu je předponou sekvence θ_{S2} přechodu druhého. Pro příklad uveďme přechody $(Q_1, a, Z_1) = (Q_2, \varepsilon)$ a $(Q_1, a, Z_1 Z_2) = (Q_2, \varepsilon)$. Je-li automat ve stavu Q_1 , na vstupu je symbol a a na zásobníku jsou symboly $Z_1 Z_2$, potom vyhovují oba přechody a zásobník se stává nedeterministickým. Stejně jako u jeho nedeterministické verze, deterministický zásobníkový automat slovo přijímá, jestliže přečte celé slovo a skončí v koncovém stavu, nebo přečte-li celé slovo a ze zásobníku odebere všechny symboly. Způsob přijetí je nutný definovat předem.

Deterministické bezkontextové gramatiky jsou gramatiky, pro které lze sestrojít deterministický zásobníkový automat.

Determinismus u zásobníkového automatu je pro $LR(k)$ gramatiky (resp. $LR(k)$ parseery) klíčový. Dovoluje totiž parsování v lineárním čase v závislosti na délce slova. $LR(k)$ parseery byly prvně popsány Donald Knuthem, ale bylo od nich opuštěno z důvodu velkých paměťových nároků na parsovací algoritmus. S ohledem na zvyšující se výkon počítačů se k tomuto algoritmu opět vracíme [7].

Pro libovolnou deterministickou bezkontextovou gramatiku lze sestrojít $LR(k)$ parser, jedná se tedy o nejuniverzálnější parser s lineární časovou složitostí. Pro parsování je použita tzv. bottom-up metoda založená právě na fungování deterministického zásobníkového automatu [7].

Zjednodušeně se na zásobník postupně vkládají znaky vstupního slova. Ve chvíli, kdy existuje pravidlo, které se derivuje na symboly v zásobníku, je

pravidlo použito a symboly v zásobníku jsou nahrazeny levou stranou pravidla. Tento postup nutně nemusí vést na deterministický zásobníkový automat. Aby kompilér věděl, které pravidlo použít, používá tzv. *LR parsing table*. Ta je předpočítaná a obsahuje vyhovující pravidla v závislosti na dalších znacích vstupního slova. Parser na základě k nepřechtených znaků vstupního slova a aktuálního stavu zásobníku vyhledá v tabulce příslušné pravidlo, které použije (odtud *LR(k)* parsery). Tím je zajištěn determinismus a lineární časová složitost [28].

1.2.3 LL gramatiky

LL(k) gramatiky [6] jsou podobně jako *LR(k)* gramatiky založeny na fungování deterministického zásobníkového automatu, ale tentokrát tzv. top-down metodou. Ta nejprve vloží na zásobník startovací symbol. Existuje-li pravidlo, které přepisuje symbol na vrcholu zásobníku, je použito a symbol na zásobníku je nahrazen pravou stranou pravidla. Ve chvíli, kdy je symbol na vstupu stejný jako symbol na vrcholu zásobníku, je symbol ze zásobníku odebrán a symbol ze vstupu přečten.

Tento postup téměř jistě nevede na deterministický zásobníkový automat, protože je symbol expandován bez znalosti vstupu (tedy nedeterministicky). Proto *LL(k)* parsery (podobně jako *LR(k)* parsery) tvoří tzv. *LL parsing table*, která plní stejnou funkci (odtud *LL(k)* parsery).

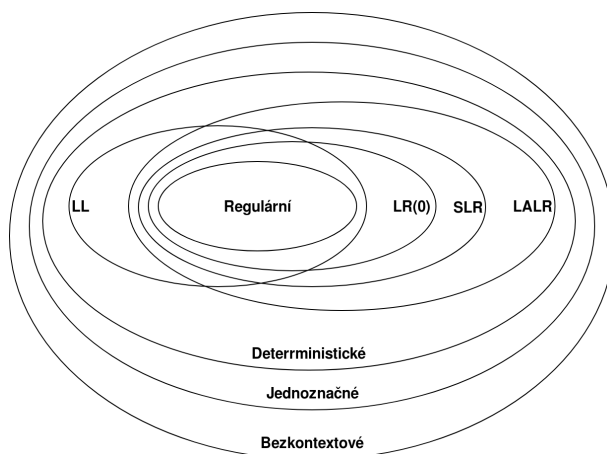
LL(k) gramatiky nejsou tak univerzální jako *LR(k)* gramatiky (jsou jejich podmnožinou), protože se při parsování musí rozhodnout o použití pravidla ještě před tím, než je jakýkoliv vstup přečten (na rozdíl od *LR(k)* parserů, které se rozhodují až po přečtení). Na druhou stranu se díky tomu v *LL parsing table* zajímáme pouze o levou stranu pravidel, zatímco v *LR parsing table* je to strana pravá. To vede k menšímu počtu záznamů a tím k menší zátěži na paměť [6].

1.2.4 LR(0), SLR a LALR gramatiky

LR(0), *SLR* a *LALR* gramatiky jsou zjednodušené *LR(k)* gramatik [7]. Ačkoliv dokáže *LR(k)* gramatika zpracovat libovolnou deterministickou gramatiku, paměťová složitost *LR(k)* parserů byla v minulosti příliš velká. *LR(0)* parsery fungují bez paměťové režie (parsery mají nulový náhled, tedy *LR parse table* není použita), ale mají malou vyjadřovací schopnost.

Z toho důvodu popsal Frank DeRemer dvě zjednodušené verze *LR(k)* parserů, konkrétně Simple LR Parser (*SLR*) a Look-Ahead LR Parser (*LALR*) [12]. Tyto parsery si ponechaly lineární časovou složitost, ale na rozdíl od *LR(k)* parserů jsou méně náročné na paměť.

Vzhledem k počtu skupin, které byly výše zmíněny, je přiložen graf (obrázek 1.3) ukazující bezkontextové gramatiky, jejich rozdělení a vzájemnou inkluzi a exkluzi jednotlivých podmnožin.



Obrázek 1.3: Rozdělení bezkontextových jazyků

S již zmíněnými parsovacími metody se nebudeme dále zabývat, protože nejsou předmětem této práce. Byly vyjmenovány z důvodu, že na ně knihovna musí pamatovat pro budoucí rozšíření. Práce se dále zabývá CYK algoritmem, který je detailněji popsán v další části. Pro použití CYK algoritmu je nutné mít gramatiku v CNF a tomu se věnuje sekce 1.5.

1.3 Formální překlady

Překlad [3] je operace, která každému vstupu přiřazuje nějaký výstup. Překlady mezi různými světovými jazyky pravděpodobně není nutné zmiňovat, ale může se jednat i o převod matematických zápisů mezi prefixovou, postfixovou a infixovou notací. Dalším příkladem je převod mezi číselnými soustavami nebo kódování (například binární data na Base64).

Formální překlad je relace $Z \subseteq L \times V$, kde L je libovolný jazyk a V jazyk překladů. Relace přiřazuje každému slovu jazyka L jeho překlad patřící do V .

Překladová gramatika je $PG = (N, T, D, S, R)$, T je množina terminálů (resp. vstupních symbolů), N je množina neterminálů, D je výstupní abeceda (disjunktní s abecedou terminálů a neterminálů), S je počáteční symbol a R je množina pravidel $R \subseteq (T \cup N \cup D)^*$.

Do definice gramatik přibyla v překladové gramatice výstupní abeceda (či množina výstupních symbolů). Výstupní symboly budeme dále v textu označovat malými písmeny v kruhu – \textcircled{a} , \textcircled{x} .

Homomorfismus je zobrazení $h \subseteq X \times Y^*$, kde X a Y jsou abecedy.

Rozšířený homomorfismus je homomorfismus $h : X^* \times Y^*$ definovaný indukci.

$$h(\varepsilon) = \varepsilon$$

$$h(xa) = h(x)h(a), x \in X^*, a \in X$$

Vstupní homomorfismus je rozšířený homomorfismus, který nahrazuje všechny symboly $x \in D$ symbolem ε .

Výstupní homomorfismus je rozšířený homomorfismus, který nahrazuje všechny symboly $x \in T \cup N$ symbolem ε .

Překlad zpravidla probíhá tak, že je naparsováno vstupní slovo (odtud vstupní homomorfismus) a podle použitých pravidel je výsledkem výstupní slovo (odtud výstupní homomorfismus). Samotný derivační strom (resp. větná forma) obsahuje jak vstupní, tak výstupní symboly.

Předpokládejme $G = (\{S, A, P, K\}, \{a, +, *\}, \{\textcircled{a}, \oplus, \ominus\}, S, R)$, kde R:

$$\begin{array}{ll} S \rightarrow a\textcircled{a}A & A \rightarrow *Kz \\ S \rightarrow a\textcircled{a} & A \rightarrow +P \\ K \rightarrow a\textcircled{a}(*A) & P \rightarrow a\textcircled{a}\oplus A \\ K \rightarrow a\textcircled{a} * & P \rightarrow a\textcircled{a}\oplus \end{array}$$

Pro vstupní slovo $w_i = a + a * a$ bude překlad $w_o = \textcircled{a}\textcircled{a}\oplus\textcircled{a}*$. Derivační strom je v příloze na obrázku A.1.

Ačkoliv by se to z názvu „Překladové gramatiky“ dalo usuzovat, tyto gramatiky ve skutečnosti nepřekládají programy do spustitelné podoby. Překladové gramatiky nejsou schopny dodat sémantickou stránku překladu, tedy to, jak se bude program chovat. K tomu slouží gramatiky atributované.

1.4 Atributované gramatiky

Již zmiňované parsovací metody řeší pouze skladbu vstupního slova – jeho syntax. Jeho skladba má ovšem i specifický význam – hovoříme o sémantickém významu. Následující kapitola vychází primárně z materiálu [41].

Syntax je množina pravidel a principů, které řídí strukturu věty daného jazyka, zpravidla definicí povolených kombinací symbolů. Syntax zajišťuje ověření, zda je vstupní věta správně strukturována.

Správnou syntaxi vstupní věty vynutíme právě použitím gramatik. Ve většině případů nechceme pouze ověřit, zda je vstupní věta syntakticky správně, ale požadujeme její další zpracování. Pokud již máme vytvořenou syntaxi, nezpracováváme samotný text, ale jeho význam. Z tohoto důvodu se snažíme

text během procesu parsování převést do jiných struktur, vhodnějších pro další zpracování. Jednou z takových struktur je abstraktní syntaktický strom.

Abstraktní syntaktický strom je stromovou reprezentací abstraktní syntaktické struktury zdrojového kódu. Uzly abstraktního derivačního stromu reprezentují konstrukce ve zdrojovém kódu. Abstraktní syntaktický strom dodržuje pravidla syntaxe, uzly a hrany derivačního stromu jsou převedeny do struktur vhodných k dalšímu zpracování. Samotná podoba těchto struktur se může v závislosti na dalším zpracování lišit. Svým způsobem se jedná o reprezentaci derivačního stromu datovými strukturami. Abstraktní syntaktický strom budeme dále v textu označovat jako AST (z anglického Abstract Syntax Tree).

Abstraktní syntaktický strom reprezentuje syntaktickou stránku programovacích jazyků. Respektuje správnou syntax, ale nedefinuje význam, tedy sémantiku. Sémantika musí být přidána dodatečně.

Sémantika přidává význam syntakticky korektním větám. V kontextu programovacích jazyků sémantika popisuje postup, který zařízení provádí během spuštění daného programu.

Pro definici atributované gramatiky, která dodává sémantický význam, potřebujeme nadefinovat další pojmy.

Atribut je veličina, která může nabývat hodnot z definované množiny. Atribut můžeme přirovnat k proměnné v programovacích jazycích.

Atributovaný symbol je symbol abecedy, ke kterému je přiřazena konečná množina atributů. Tato množina může být i prázdná. Přístup k atributu a přiřazeného symbolu x budeme značit jako $x.a$. Atribut a s hodnotou 1 pro symbol x budeme značit jako $x.a[1]$. Jestliže bude atributů více, budeme je oddělovat tečkou, tedy symbol x s jeho atributy $a = 1$ a $b = 2$ budeme značit jako $x.a[1].b[2]$.

Atributovaný řetězec nad abecedou A je řetězec atributovaných symbolů z A .

Atributovaný překlad je relace $T^* \times D^*$, kde T^* je množina vstupních atributovaných řetězců a D^* množina výstupních atributovaných řetězců.

Vztáhneme-li kontext zpět na gramatiky a AST, můžeme každému terminálu a neterminálu přiřadit množinu atributů. Dále budeme uvažovat pouze o jednom atributu v . Pomocí atributů můžeme mít několik způsobů, jak reprezentovat číslo 253.

- Neterminál x reprezentující celé číslo jako $x.v[253]$.
- Neterminál x reprezentující cifru a pravidla $c \rightarrow xc, c \rightarrow \varepsilon$, kde c reprezentuje celé číslo. Pro číslo 253 by $c \Rightarrow x_1c \Rightarrow x_1x_2c \Rightarrow x_1x_2x_3c \Rightarrow x_1x_2x_3$, kde $x_1.v[2], x_2.v[5], x_3.v[3]$.
- Neterminály x_0, x_1, \dots, x_9 s pevně danými atributy $x_k.v[k]$.

Konkrétní výběr se může lišit podle způsobu dalšího zpracování a kontextu gramatiky. Z pohledu sémantiky bychom ovšem potřebovali, abychom ke všem třem variantám přistupovali stejně, tedy očekáváme, že kořen AST reprezentující číslo bude mít atribut obsahující hodnotu čísla. Přitom chceme zůstat nezávislí na způsobu reprezentace.

Jak je z popisu patrné, hodnoty neterminálních atributů nemůžeme volit pevně, ale musí se měnit v závislosti na pozici v AST – nejčastěji na základě atributů potomků nebo rodiče. Toho dosáhneme sémantickými pravidly.

Sémantická funkce je funkce, která přiřadí atributu hodnotu na základě jiných atributů. Jedná se o funkci $f = (a_1, a_2, \dots, a_n)$, kde a_1, a_2, \dots, a_n jsou atributy, na kterých je funkce závislá. Funkce je zapsána libovolným, pro danou situaci vhodným způsobem (například programovacím jazykem uvnitř překladače).

Sémantické pravidlo je pravidlo, které má navíc sémantickou funkci.

Atributovaná překladová gramatika je $APG = (N, T, D, SR, S, A, V)$, kde N je abeceda neterminálů, T je abeceda terminálů, D je abeceda výstupních symbolů, SR jsou sémantická pravidla, S je počáteční symbol $S \in N$, A je množina atributů a V je zobrazení $V \subseteq (T \cup N) \times A^*$, tedy množina přiřazující každému terminálu či neterminálu množinu atributů.

Dědičný atribut je atribut v symbolu x , jehož sémantická funkce f závisí pouze na dědičných attributech symbolu x nebo na dědičných attributech jeho rodiče.

Syntetizovaný atribut je atribut v symbolu x , jehož sémantická funkce závisí pouze na syntetizovaných či dědičných attributech symbolu x , nebo na syntetizovaných či dědičných attributech jeho potomků.

Vyhodnocení atributů je proces, během kterého jsou vyhodnoceny sémantické funkce a nastaveny hodnoty atributů. Pro správné vyhodnocení musí být splněno:

- hodnoty dědičných atributů počátečního symbolu musí být známy,

- hodnoty syntetizovaných atributů vstupních symbolů musí být známy,
- hodnota atributu musí záviset pouze na již známých hodnotách atributů.

Poslední bod vylučuje situace, které by mohly vést k zacyklení výpočtu. Nejjednodušším příkladem je atribut a závisející na atributu b , přitom atribut b zpětně závisí na atributu a . Pokud budou atributy záviset pouze na již známých hodnotách atributů, k zacyklení dojít nemůže.

Jako příklad atributované gramatiky uvedeme gramatiku vyhodnocující aritmetické operace $+$ a $*$. Pro všechny symboly předpokládáme syntetizovaný atribut v . Pravidla lze nalézt v tabulce 1.1. Symboly, které se v pravidlu vykytují opakovaně je nutné v sémantických funkcích rozlišit, proto jsou symboly indexovány. V obrazové příloze je na obrázku A.2 zobrazeno vyhodnocení vstupního slova $a.v[3] + a.v[2] * a.v[4]$.

Tabulka 1.1: Atributovaná gramatika vyhodnocující aritmetické operace

Pravidlo	Sémantika
$S \rightarrow E$	$S.v = E.v$
$E_1 \rightarrow E_2 + T$	$E_1.v = E_2.v + T.v$
$E \rightarrow T$	$E.v = T.v$
$T_1 \rightarrow T_2 * F$	$T_1.v = T_2.v * F.v$
$T \rightarrow F$	$T.v = F.v$
$F \rightarrow a$	$F.v = a.v$
$F \rightarrow (E)$	$F.v = E.v$

1.5 Chomského normální forma

Chomského normální forma (CNF) je speciální formát gramatiky vyžadována CYK algoritmem. Libovolnou bezkontextovou gramatiku lze převést do CNF [28]. Pro tuto transformaci jsou známy a popsány algoritmy, které jsou blíže rozebrány v další kapitole.

Chomského normální forma povoluje pouze pravidla následujících typů:

- $A \rightarrow BC$,
- $A \rightarrow a$,
- $S \rightarrow \varepsilon$, kde S se nesmí vyskytovat na pravé straně žádného pravidla.

Před tím, než blíže popíšeme algoritmy pro transformaci gramatik, sloužící k převodu do CNF, potřebujeme dodefinovat další pojmy, které se v kontextu těchto transformací vyskytují. Pojmy vycházejí z materiálu [28].

Negenerující symbol je symbol, který negeneruje žádnou větu. Pokud je počáteční symbol S negenerující, gramatika negeneruje žádné slovo, a tedy generuje prázdný jazyk.

Dostupný symbol je symbol, který se vyskytuje v nějaké větné formě. Formálně se jedná o symbol $X : S \Rightarrow \alpha X \beta; \alpha, \beta \in (T \cup N)^*$.

Nedostupný symbol je symbol, který není dostupný.

ε **pravidlo** je pravidlo ve tvaru $X \rightarrow \varepsilon$.

Zkracující pravidlo je pravidlo, jehož počet symbolů na levé straně je větší než počet symbolů na straně pravé. Délka ε symbolu je považována za nulovou.

Zkracující gramatika je gramatika, která má alespoň jedno pravidlo zkracující.

Symbol derivovatelný na ε přímo je takový symbol X , pro nějž platí $X \Rightarrow^1 \varepsilon$. V gramatice musí existovat pravidlo $X \rightarrow \varepsilon$.

Symbol derivovatelný na ε nepřímo je takový symbol X , pro nějž platí $X \Rightarrow^i \varepsilon; i > 1$.

Jednoduché pravidlo je pravidlo tvaru $A \rightarrow B$, tedy pravidlo přepisující neterminál na neterminál.

Operace s gramatikami

Operace s gramatikami můžeme obecně rozdělit na transformační (mění podobu gramatiky) a parsovací (zpracovávají text). V rámci parsovacích operací se snažíme o nejefektivnější validaci vstupu za současného převodu do struktur vhodných pro další zpracování. Tyto struktury se mohou v závislosti na úrovni abstrakce a potřebě zasahovat do procesu zpracování měnit. Běžně používanou reprezentací je již zmiňovaný abstraktní syntaktický strom, kterým se budeme zabývat dále v práci. Další běžně známou reprezentací je Java bytecode, který je překládán až na cílové platformě při spuštění aplikace. GCC například podporuje několik reprezentací: Register Transfer Language, LLVM, Static single assignment form a další.

Z hlediska transformačních operací jde o vytvoření nové, optimalizované gramatiky, která bude generovat stejný jazyk jako gramatika původní. To je případ i algoritmů pro převod gramatiky do CNF. Tyto transformace jsou známé a popsány. Transformace, které mění generující jazyk, jsou méně obvyklé, protože vyžadují širší kontext a náročnou intelektuální práci, kterou je téměř nemožno algoritmizovat. Tyto transformace dovoluují změnit gramatiku do podoby, kterou je možno parsovat rychlejším parsovacím algoritmem. Pokud je generovaný jazyk podmnožinou (resp. nadmnožinou nebo kombinací) původního jazyka, musí se rozdíly zpracovat v pozdějších fázích překladu.

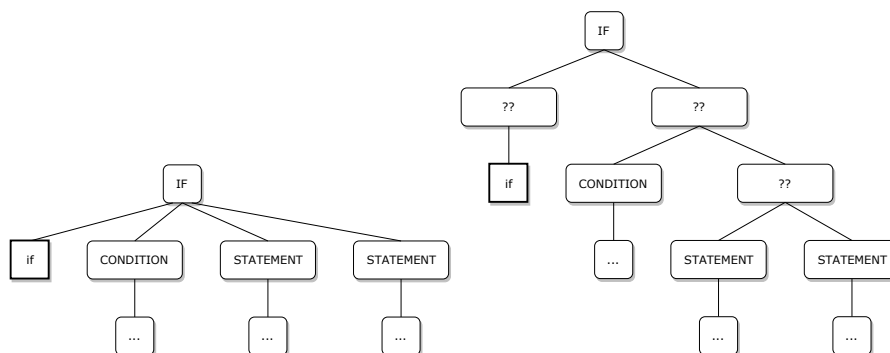
2.1 Problémy transformací

Na druhou stranu i transformace, neměníci generovaný jazyk, mohou měnit syntaktickou strukturu. To je nežádoucí, protože v rámci atributovaných gramatik (či jiného způsobů zpracování) předpokládáme strukturu definovanou naší gramatikou. Aplikováním transformací se gramatika (a tím i syntaktická struktura) změní.

Pro představu budeme situaci demonstrovat na jednoduché gramatice (obrázek 2.1), převedené do CNF. Po naporsování vstupu bychom očekávali vý-

STATEMENT \rightarrow if CONDITION STATEMENT STATEMENT
 STATEMENT \rightarrow ...
 CONDITION \rightarrow ...

Obrázek 2.1: Ukázka gramatiky představující „if statement“



Obrázek 2.2: Očekávaný a reálný syntaktický strom pro „if statement“

stup jako na obrázku 2.2 vlevo. Ale vzhledem k transformacím gramatiky bude výsledek odlišný (obrázek 2.2 vpravo).

To je z hlediska sémantické analýzy komplikace, protože v době tvorby gramatiky netušíme, jak bude gramatika transformovaná a to nám brání v dodání sémantického významu pro modifikovaný AST. Z tohoto důvodu musí knihovna obsahovat i funkcionalitu ke zpětné rekonstrukci syntaktického stromu podle původní netransformované gramatiky.

2.2 Převod do Chomského normální formy

Práce dále popisuje všechny transformace i s jejich pseudokódem. Po pseudokódu následuje popis zpětné transformace (pokud dává smysl), možné dopady na původní transformaci a popis změn v parsovacím procesu. Jednotlivé algoritmy, jejich popis a pseudokód vychází z materiálu [28].

2.2.1 Odstranění negenerujících symbolů

Vhodným příkladem může být gramatika pouze s jediným pravidlem $S \rightarrow xS$. Všechny větné formy generované takovou gramatikou obsahují neterminál, generovaný jazyk je tedy prázdný.

Odstranění negenerujících symbolů nemění syntaxi gramatiky (protože odstraněná část se nemůže vyskytnout v žádné větné formě generujícího slova), ale zjednodušuje ji a tím snižuje časové a paměťové nároky na algoritmy, které dále s gramatikou pracují.

Algoritmus pro odstranění negenerujících symbolů si drží množinu symbolů, které tvoří libovolnou větu. Na počátku jsou v této množině pouze terminály. Algoritmus poté hledá pravidla, které se přepíše na již generující symboly. Pokud takové pravidlo nalezne, přidá symbol na levé straně pravidla mezi generující symboly a postup opakuje.

Algoritmus končí ve chvíli, kdy jsou zpracována všechna pravidla a žádný další generující symbol není přidán. Algoritmus poté z gramatiky vymaže všechny negenerující symboly a s nimi i všechna pravidla, která tyto symboly obsahují. Za předpokladu, že je gramatika konečná, algoritmus skončí po konečném počtu kroků. Pseudokód algoritmu je kód 2.1.

```
odstranění_negenerujících_symbolů(gramatika):
    generující = gramatika.terminály
    opakuj dokud se generující množina nezměnila:
        pro každé pravidlo v gramatice:
            jsou-li všechny symboly pravé strany generující:
                generující.přidej(pravá strana pravidla)

    odstranit = gramatika.nonterminály - generující
    gramatika.odstran(odstranit)
```

Kód 2.1: Pseudokód algoritmu odstraňující negenerující symboly

Pro algoritmus není nutné implementovat zpětnou transformaci, protože nemění syntaktickou strukturu gramatiky, ale pouze gramatiku optimalizuje.

Speciální situace nastává ve chvíli, kdy algoritmus označí počáteční symbol gramatiky jako negenerující. V takovém případě gramatika generuje prázdný jazyk a v dalších algoritmech nemusíme pokračovat.

2.2.2 Odstranění nedostupných symbolů

Odstranění nedostupných symbolů je, stejně jako odstranění negenerujících symbolů, pouze optimalizační technika, která snižuje velikost gramatiky a tím i časové a paměťové nároky dalších algoritmů, které s gramatikou pracují.

Algoritmus pro odstranění nedostupných symbolů je velmi podobný algoritmu pro odstranění negenerujících symbolů, pouze prochází gramatiku v opačném směru. Algoritmus si drží množinu dostupných symbolů, do které na počátku vloží počáteční symbol. Poté prochází pravidla a pro ta, jejichž levá strana je obsažena v množině, vloží do množiny dostupných symbolů také všechny symboly pravé strany. Pseudokód je kód 2.2.

Na rozdíl od algoritmu pro odstranění negenerujících symbolů mohou být během běhu algoritmu odstraněny i terminály. Terminály mohou být odstraněny všechny, a to ve chvíli, kdy gramatika generuje pouze ε .

```
odstranění_nedostupných_symbolů(gramatika):
    dostupné = gramatika.počáteční_symbol
    opakuj dokud se množina dostupných symbolů nezměnila:
        pro každé pravidlo v gramatice:
            je-li levá strana pravidla v dostupné:
                dostupné.přidej(pravá strana)

    odstranit = gramatika.nonterminály
                + gramatika.terminály
                - dostupné
    gramatika.odstran(odstranit)
```

Kód 2.2: Pseudokód algoritmu odstraňující nedostupné symboly

2.2.3 Odstranění ε pravidel

Bezkontextové gramatiky mají obecný tvar pravidla $X \rightarrow (T \cup N)^*$. V takové gramatice lze (na rozdíl od gramatik regulárních a kontextových) derivovat libovolný symbol na ε (obsahuje-li gramatika příslušná pravidla). To by komplikovalo parsovací proces, protože by byla gramatika zkracující.

Cílem algoritmu je transformovat gramatiku s ε pravidly na gramatiku bez ε pravidel a tím ji udělat nezkracující.

Algoritmus nejprve nalezne všechny neterminály, které jsou přímo nebo nepřímo derivovatelné na ε . Postup je velmi podobný jako u algoritmu pro odstranění negenerujících symbolů, pouze s tím rozdílem, že na počátku algoritmu vložíme do generujících symbolů pouze ε . Pseudokód je kód 2.3.

```
nalezení_neterminálů_přepisovatelných_na_epsilon(gramatika):
    generujici = vytvoř množinu
    opakuj dokud se generujici množina nezměnila:
        pro každé pravidlo v gramatice:
            jsou-li všechny symboly pravé strany přepisovatelné:
                generujici.přidej(pravá strana)

    vrať přepisovatelné
```

Kód 2.3: Pseudokód algoritmu hledající neterminály generující ε

Poté algoritmus projde všechna pravidla. Pravidla, která se přímo přepisují na ε jsou z gramatiky vyloučena. Touto operací dojde k modifikaci gramatiky a tím i ke změně generujícího jazyka. To je pro další práci nepřípustné, protože jako výstup požadujeme gramatiku generující stejný jazyk.

Vyloučení pravidla je proces, který z gramatiky odebere pravidlo bez toho, aniž by byl změněn generující jazyk. Necht $G = (N, T, S, R)$ je bezkontextová gramatika a v ní pravidlo $\rho = A \rightarrow \alpha$, které vyloučíme. Jestliže ke všem pravidlům $X \rightarrow \gamma A \delta$ přidáme pravidlo $X \rightarrow \gamma \alpha \delta$, potom se generovaný jazyk nezmění.

Věta nám říká, že nahradíme-li ve všech pravidlech neterminál, který je na levé straně vyloučeného pravidla, pravou stranou vyloučeného pravidla, poté gramatika popisuje stejný jazyk. Pseudokód je kód 2.4.

```
odstraň_pravidlo(gramatika, pravidlo):
    gramatika.odstraň(pravidlo)
    pro každé pravidlo p_1 v gramatice:
        pro každý symbol v p_1.pravá_strana:
            jestliže symbol = pravidlo.levá_strana:
                nové = nahraď symbol pravou stranou pravidla
                gramatika.přidej(nové)
```

Kód 2.4: Pseudokód algoritmu odstraňující pravidlo

V našem případě odstraňujeme pouze ε pravidla (pravidla typu $X \rightarrow \varepsilon$). Během odstranění se modifikovaná pravidla zkracují (například z pravidel $\{A \rightarrow B, B \rightarrow \varepsilon\}$ vznikne $\{A \rightarrow B, A \rightarrow \varepsilon\}$), může se tedy stát, že vznikne nové ε pravidlo. Algoritmus proto opakuje postup tak dlouho, dokud v gramatice existují ε pravidla. Pseudokód je kód 2.5.

```
odstranění_epsilon_pravidel(gramatika):
    dokud v gramatice existují epsilon pravidla:
        pravidlo = najdi_epsilon_pravidlo(gramatika)
        odstraň_pravidlo(pravidlo)
```

Kód 2.5: Pseudokód algoritmu odstraňující ε pravidla

Tato verze algoritmu má velkou časovou složitost, protože opakovaně prochází všechna pravidla v gramatice. Algoritmus lze optimalizovat použitím fronty a množiny symbolů derivovatelných na ε . Algoritmus poté prochází každé pravidlo pouze jednou.

Pro algoritmus odstranění ε symbolů již potřebujeme algoritmus zpětné transformace syntaktického stromu. Při transformaci se pro nově vytvořená pravidla (tedy pravidla s odstraněným ε symbolem) uloží pravidlo, ze kterého nové pravidlo vzniklo, a symbol, který byl přepsán na ε . Na základě těchto informací je algoritmus schopen určit originální pravidlo a syntaktický strom transformovat tak, aby odpovídal originální gramatice.

2.2.4 Odstranění jednoduchých pravidel

Algoritmus pro odstranění jednoduchých pravidel je podobný tomu pro odstranění ε pravidel. Algoritmus ve své nejjednodušší implementaci vylučuje jednoduchá pravidla až do doby, pokud v gramatice ještě nějaká existují. Optimalizovaná verze algoritmu má několik kroků, ale má nižší časové a paměťové nároky. Jedná se o pseudokód 2.6.

1. Algoritmus vytvoří tranzitivní uzávěr jednoduchých pravidel.
2. Algoritmus odstraní z gramatiky všechna jednoduchá pravidla.
3. Algoritmus prochází zbylá pravidla a vybere ta, pro která v gramatice existuje neterminál derivovatelný jednoduchými pravidly na levou stranu procházeného pravidla.

Dále předpokládáme, že mluvíme o obecném pravidlu $X \rightarrow Ab$ a neterminálu Z takovém, že existuje derivece $Z \Rightarrow X$ za použití pouze jednoduchých pravidel.

4. Algoritmus vytvoří nové pravidlo. Na levé straně pravidla bude procházený neterminál a na pravé straně pravá strana procházeného pravidla. Algoritmus tedy vytvoří pravidlo $Z \rightarrow Ab$.
5. Vytvořené pravidlo přidá do gramatiky

```
odstranění_jednoduchých_pravidel(gramatika):  
    uzávěr = tranzitivní uzávěr jednoduchých pravidel  
    pro každé pravidlo p_1 gramatiky:  
        jestliže je p_1 jednoduché:  
            gramatika.odstraň(p_1)  
            pokračuj s dalším pravidlem  
    pro každý neterminál n_1:  
        pokud lze derivovat n_1 na p_1.levá_strana  
            nové = pravidlo(n_1, p_1.pravá_strana)  
            gramatika.přidej(nové)
```

Kód 2.6: Pseudokód algoritmu odstraňující jednoduchá pravidla

Při algoritmu zpětné transformace je nutné pro nově vytvořené pravidlo uložit sérii jednoduchých pravidel, na základě kterých bylo pravidlo vytvořeno. Algoritmus dokáže podle této informace určit, která jednoduchá pravidla byla použita při odstranění, a z toho vyvodit správnou podobu syntaktického stromu.

2.2.5 Převod na Chomského normální tvar

Připomeňme, že CNF vyžaduje pravidla ve třech podobách:

- $A \rightarrow BC$,
- $A \rightarrow a$,
- $S \rightarrow \varepsilon$.

Po dokončení předchozích algoritmů máme jistotu, že se v gramatice nevykazuje pravidlo derivující se na ε nebo jednoduchá pravidlo. Transformace do CNF již probíhá jednoduše podle následujícího postupu.

- Pravidla vyhovující CNF ponecháme.
- Pravidla tvaru $A \rightarrow \alpha a \beta$ nahradíme pravidly $\{A \rightarrow \alpha a' \beta, a' \rightarrow a\}$ kde a je terminál a a' je nový neterminál, dosud v gramatice nepoužitý.
- Pravidla delší než dva neterminály rozdělíme vytvořením nových neterminálů a pravidel. Pravidlo

$$A \rightarrow B_1 B_2 \cdots B_k$$

rozdělíme na

$$\{ A \rightarrow B_1 \langle B_2 \cdots B_k \rangle, \langle B_2 \cdots B_k \rangle \rightarrow B_2 \langle B_3 \cdots B_k \rangle, \cdots, \\ \langle B_{k-2} B_{k-1} B_k \rangle \rightarrow B_{k-2} \langle B_{k-1} B_k \rangle, \langle B_{k-1} B_k \rangle \rightarrow B_{k-1} B_k \}$$

kde $\langle \cdots \rangle$ jsou nové neterminály, dosud v gramatice nepoužité.

Pro algoritmus zpětné rekonstrukce nám postačí ukládat, o který ze dvou bodů výše se jedná, současně s pravidlem, ze kterého bylo nové pravidlo vytvořeno. Poté stačí uzel v syntaktickém stromu nahradit jeho správným ekvivalentem. V případě druhého bodu nahradíme uzel samotným terminálem, ve třetím případě (rozdělení pravidla) získáme zbytek neterminálů jako potomky pravého potomka.

2.3 Cocke-Younger-Kasami algoritmus

Cocke-Younger-Kasami algoritmus [18] (nebo též CYK) je jeden z parsovacích algoritmů pro bezkontextové gramatiky. Na rozdíl od zmíněných parsovacích algoritmů (jako LL a LR algoritmy) je schopen zpracovat libovolnou bezkontextovou gramatiku včetně nejednoznačné. Algoritmus je v takovém případě nedeterministický, ale konečný a CYK je schopen vstupní slovo naporsovat. Jedná se o hlavní vlastnost, kvůli které byl algoritmus pro tuto práci vybrán.

Jeho nevýhodou je zvýšená časová a paměťová náročnost. Časová složitost je v nejhorším případě $\mathcal{O}(n^3 \cdot |G|)$, kde $|G|$ je počet pravidel v gramatice a n je délka vstupního slova. Paměťová složitost je v nejhorším případě $\mathcal{O}(n^2 \cdot |G|)$. Další nevýhodou je neschopnost odhalovat chyby. Algoritmy pracující se slovem lineárně jsou schopny označit konkrétní symbol nevyhovující definici gramatiky. CYK algoritmus odhalí nevalidní vstup, ale vzhledem k jeho konstrukci není schopen určit místo, na kterém k chybě došlo.

CYK algoritmus je založen na dynamickém programování. Algoritmus nejdříve najde neterminály, které se přímo derivují na terminály vstupního slova (tj. použije všechny možné $X \rightarrow a$ pravidla). Algoritmus dále hledá všechny neterminály, které se derivují na podslova délky 2, 3, \dots , n obsažené ve vstupním slovu. Přitom využívá faktu, že slovo délky k jsou dvě slova délek $(k-1, 1)$, $(k-2, 2)$, \dots , $(2, k-2)$, $(1, k-1)$. Pokud se v neterminálech, které generují celé vstupní slovo, vykytuje startovací symbol, potom je vstup úspěšně naparsován a algoritmus může vytvořit syntaktický strom.

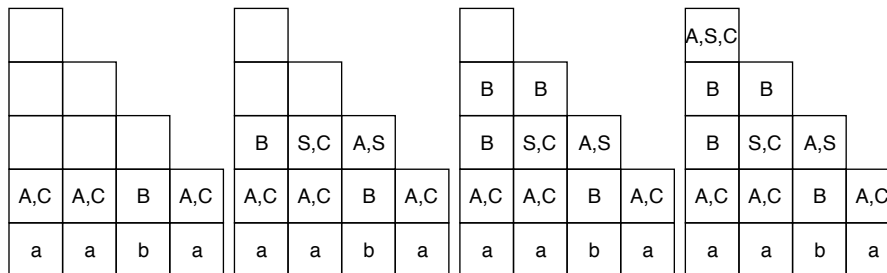
Běh algoritmu budeme dále v textu demonstrovat tabulkou. První řádek tabulky budou neterminály generující vstupní slovo (tedy slovo délky n), druhý řádek část slova délky $n-1$, třetí řádek část slova délky $n-2$ atd. Na posledním řádku tabulky budou neterminály, které se přímo derivují na symboly vstupního slova.

Proces budeme demonstrovat na gramatice $G = (\{S, A, B, C\}, \{a, b\}, S, \{S \rightarrow AB, S \rightarrow BC, A \rightarrow BA, A \rightarrow a, B \rightarrow CC, B \rightarrow b, C \rightarrow AB, C \rightarrow a, \})$ a slovu $w = aaba$.

Algoritmus nejdříve zjistí, které neterminály se přímo derivují na symboly slova (například neterminály A a C se derivují na a). Ve druhém kroku tvoří slovo délky 2. Například neterminál $B \Rightarrow CC \Rightarrow aa$. Ve třetím kroku tvoří slova délky 3, tzn. kombinace slova délky 1 a slova délky 2 nebo slovo délky 2 a slovo délky 1. Pro neterminál B se použilo pravidlo $B \rightarrow CC$, kde první neterminál C je na třetím řádku a druhém sloupci (slovo délky 2) a druhý neterminál C na čtvrtém a řádku čtvrtém sloupci (slovo délky 1). Neterminál B skutečně generuje slovo délky 3, tedy $B \Rightarrow CC \Rightarrow ABC \Rightarrow aba$. V posledním kroku algoritmus detekuje, které neterminály lze derivovat do vstupního slova. V prvním řádku se vykytuje počáteční symbol S , a tedy je slovo generováno pravidly $S \Rightarrow AB \Rightarrow ACC \Rightarrow AABC \Rightarrow aaba$.

Kód 2.7 poté slouží jako pseudokód CYK algoritmu.

Pro samotnou konstrukci syntaktického stromu je potřeba algoritmus upravit. CYK algoritmus během svého běhu nebude do tabulky ukládat neterminály, které se derivují na slovo, ale uloží si použité pravidlo a pozice symbolů pravé strany pravidla. Díky těmto informacím může algoritmus po úspěšném parsování sestavit syntaktický strom bez dodatečného procházení gramatiky nebo vstupního slova.



Obrázek 2.3: Kroky CYK algoritmu

```

pozice(řádek, sloupec):
    kombinace = vytvoř množinu
    počet_kombinací = délka slova - řádek
    pro v jdoucí do počet_kombinací:
        b_1 = bod(řádek + v + 1, sloupec)
        b_2 = bod(řádek + počet_kombinací - v,
                 sloupec + počet_kombinací - v)
        kombinace.přidej((b_1, b_2))
    vrať kombinace

cyk(gramatika, slovo):
    pole = vytvoř dvojrozměrné pole velikosti délky slova
    pro každý symbol slova:
        pravidla = najdi pravidla derivující se na symbol
        pole[délka slova][index] = pravidla.levá_strana
    pro každý řádek i:
        pro každý sloupec j:
            pro každý prvek (b_1, b_2) v pozice(i, j):
                první_s = pole[b_1.x][b_1.y]
                druhý_s = pole[b_2.x][b_2.y]
                pro každé pravidlo p_1:
                    pokud p_1.pravá_strana = (první_s, druhý_s)
                        pole[i][j] += p_1.levá_strana
    jestliže je gramatika.počáteční_symbol v pole[0][0]:
        úspěch
    jinak:
        neúspěch

```

Kód 2.7: Pseudokód CYK algoritmu

Analýza existujících řešení a návrh knihovny

Následující kapitola nejprve analyzuje požadavky, na základě kterých volíme cílovou platformu a programovací jazyk. Dále jsou analyzována již existující řešení nezávisle na zvolené technologii. Těmito řešeními se knihovna z velké části inspiruje. Na základě předchozích informací jsou navrženy základní bloky knihovny a jejich architektura.

3.1 Požadavky

Požadavky z velké části reflektují cíle práce. Pro zlepšení přehlednosti a lepší orientaci v problematice budou v této kapitole požadavky vypsané a problémové z nich blíže rozebrány.

Požadavky jsou analyzovány metodou FURPS. Metoda FURPS byla vytvořena společností Hewlett-Packard a rozděluje typy požadavků na funkčnost, použití, spolehlivost, výkon a udržitelnost [5].

3.1.1 Požadavky na funkčnost

Požadavky na funkčnost udávají, jaké chování má aplikace (resp. knihovna) mít a reflektují funkcionalitu, kterou od aplikace (resp. knihovny) požadujeme.

P.f1 Knihovna musí být schopná reprezentovat libovolnou gramatiku.

P.f2 Terminálem může být cokoliv, co si uživatel zadefinuje.

P.f3 Neterminály a terminály musí být v knihovně jednoznačně určeny.

P.f4 Knihovna musí být schopna reprezentovat libovolnou gramatiku.

P.f5 Gramatiku lze modifikovat.

3. ANALÝZA EXISTUJÍCÍCH ŘEŠENÍ A NÁVRH KNIHOVNY

- P.f6** Gramatice lze definovat sémantická pravidla (tj. podpora atributovaných a překladových gramatik).
- P.f7** Knihovna umožňuje definovat transformace gramatik.
- P.f8** Knihovna dokáže transformovat libovolnou bezkontextovou gramatiku do CNF.
- P.f9** Knihovna umožňuje parsování vstupu na základě gramatiky.
- P.f10** Knihovna podporuje rozšíření pro různé parsovací metody.
- P.f11** Knihovna implementuje parsování pomocí CYK algoritmu.
- P.f12** Výsledkem parsovacích metod je AST.
- P.f13** Knihovna umožní procházení a modifikaci AST.
- P.f14** Knihovna dokáže provést zpětné transformace nad AST v případě transformací do CNF.

3.1.2 Požadavky na použití

Požadavky na použití hodnotí aplikaci (resp. knihovnu) z hlediska uživatele. Do této kategorie spadá rozhraní aplikace, dokumentace, ukázkový materiál a další.

- P.u1** Knihovna musí být použitelná na operačních systémech Microsoft Windows, macOS a Linux.
- P.u2** Návrh knihovny musí umožnit snadné rozšíření, a to i pro existující programy.
- P.u3** Funkcionalita knihovny musí být plně dokumentována.
- P.u4** Dokumentace musí být přístupná online i offline.
- P.u5** U knihovny musí být přiloženy alespoň dva příklady, jak s knihovnou pracovat.
- P.u6** Zdrojové kódy knihovny musí být dostupné online.
- P.u7** Knihovna musí podporovat snadnou instalaci.
- P.u8** Knihovna musí být vyvíjena pod licencí GNU General Public License v3.0. Knihovna tedy bude vyvíjena jako Open-Source projekt.

3.1.3 Požadavky na spolehlivost

Požadavky na spolehlivost se zabývají stabilitou, spolehlivostí a správným fungování softwaru. V rámci metrik, používaných u těchto požadavků, se řeší četnost pádů aplikace, jejich závažnost, doba potřebná k zotavení a další.

P.r1 Knihovna musí být otestována automatizovanými testy.

P.r2 Jednotkové testy knihovny musí pokrýt minimálně 90 % kódu.

P.r3 Knihovna musí ve vývoji používat systém pro průběžnou integraci (CI systém). Každá změna je automaticky otestována.

P.r4 Knihovna do produkční verze nezahrne kód, který by byl v rozporu s uvedenými body.

3.1.4 Požadavky na výkon

Požadavky na výkon řeší odezvu aplikace, její rychlost, ale i využívání prostředků systému nebo rozložení zátěže.

V kontextu naší knihovny je výkon dán více či méně použitými algoritmy. Algoritmy jsou dány uvedenými funkčními požadavky a zadáním práce. Cílem knihovny v aktuálním stádiu vývoje je poskytnout tzv. minimum viable product, tedy poskytnout základní funkcionalitu pro první uživatele a získat zpětnou vazbu pro další vývoj. Předčasná optimalizace výkonu by mohla zkomplikovat vývoj základního produktu, a tak není výkon knihovny dále uvažován jako relevantní požadavek.

3.1.5 Požadavky na udržovatelnost

Požadavky na udržovatelnost hodnotí snadnost údržby a testovatelnosti aplikace či knihovny. Neméně důležitým aspektem je i přizpůsobitelnost a rozšiřitelnost aplikace či knihovny o novou funkcionalitu nebo vlastnosti.

Některé požadavky na udržovatelnost již byly zmíněny v požadavcích na spolehlivost, resp. použitelnost knihovny. Díky automatickým testům, CI, zpřístupnění zdrojových kódů online a licenci dle požadavku **P.u8** může kdokoliv knihovnu modifikovat či rozšířit, pokud aktuální stav knihovny považuje za nedostatečný. Aby byl tento proces co nejhladší, je reflektován požadavkem na udržovatelnost.

P.s1 Knihovna musí být vyvíjena pomocí nástrojů umožňujících spolupráci více uživatelů, modifikaci knihovny, ověřování a validaci změn a integraci změn do knihovny.

3.2 Volba platformy a programovacího jazyka

Platforma a programovací jazyk určují, na kterých zařízeních lze knihovnu používat, množství potenciálních uživatelů naší knihovny, ale i například stabilitu. Při výběru programovacího jazyka také musí být brána v potaz intermobilita mezi jazyky (například použití C kódu v Java kódu), která dále zvyšuje možnosti knihovny.

Protože se jedná o knihovnu, budeme uvažovat pouze o mainstream jazycích, tedy jazycích běžně používaných. Implementovat knihovnu v méně používaných nebo exotických jazycích se jeví jako neefektivní vzhledem k malé uživatelské základně. Za mainstream jazyky považujeme: C resp. C++, Java, C#, JavaScript, PHP a Python [38]. Všechny zmíněné jazyky jsou multiplatformní a lze je (někdy s menšími úpravami) použít na libovolné z platforem.

Další z požadavků je snadné rozšíření již existujících programů. Jeden ze způsobů, jak splnit tento požadavek, je použít nekompileované jazyky. Knihovna samotná potom není distribuována v binárním kódu či mezikódu, ale je distribuována jako zdrojový kód. To se netýká pouze knihoven, ale i programů. Díky tomu lze rozšiřovat i existující programy pouze s minimem úsilí. Z výše zmíněných jazyků se jedná o JavaScript, PHP a Python.

Jazyk PHP, jakožto jazyk určený pro webové aplikace, můžeme hned vyloučit. JavaScript je jazyk, který v poslední době zaznamenal velký boom a jehož uživatelská základna se neustále rozrůstá. V práci bychom nicméně chtěli použít jazyk, který má dlouhodobě stálou uživatelskou základnu, a to je jazyk Python [39].

3.3 Existující řešení

V této sekci jsou popsána vybraná existující řešení problému, která se věnují tématu podobnému této práci. Počet nástrojů, které řeší danou problematiku, je mnoho. Předmětem práce není analýza existujících řešení, ale tvorba řešení vlastního. Již existující, a zvláště běžně užívaná řešení, slouží primárně jako inspirace pro rozhraní knihovny.

Z existujících řešení byla vybrána tři, která se od sebe liší samotným konceptem parsování:

- ručně psaný parser,
- generátor parseru,
- knihovna Pysrser.

3.3.1 Ručně psaný parser

Ručně psané parsery jsou běžné pro komplikované situace. Například parser jazyka C++ v případě GCC parseru [15]. Stejný přístup zastává i parser Clang.

Důvodem je komplikovanost a nejednoznačnost gramatiky. Například u výrazu `foo * bar` není jisté, zda se jedná o deklaraci proměnné `bar` typu ukazatele na `foo`, nebo se jedná o násobení proměnných `foo` a `bar`.

Jedná se o problém zmiňovaný v první kapitole, tedy že parsování probíhá na základě bezkontextové gramatiky, ale velká část operací kontext vyžaduje – v tomto případě context `foo` a `bar`. Bude-li `foo` název typu, poté se jedná o deklaraci. Naopak, budou-li `foo` a `bar` proměnné, jedná se o operaci násobení.

Ručně psané parsery poskytují maximální možnou svobodu. Do parsovacího procesu lze neomezeně zasahovat – například si může parser již během parsování vytvářet tabulky definovaných symbolů, proměnných a typů. Na základě těchto tabulek dokáže ve výše zmíněném případě rozhodnout, zda se jedná o deklaraci proměnné či násobení dvou proměnných.

Z výše uvedeného je zřejmé, že se zároveň jedná o nejefektivnější metodu, jakou lze použít pro parsování. Metoda má jen nezbytnou paměťovou režii a při správném použití je časová složitost nejmenší možná.

Na druhou stranu se jedná o metodu velmi náročnou, protože libovolná změna gramatiky se promítne do samotného zdrojového kódu překladače. Takové změny jsou náročné, protože vyžadují znovu analyzovat celý produkt a zajistit jeho kvalitu po změně.

Vedlejším účinkem efektivity této metody je navíc složitý kód, protože velká část operací probíhá již během parsování a ne až v pozdější, samostatné fázi. To vede k zahuštění kódu, který mimo logiku parsování řeší i dodatečné operace, které ještě více umocňují problém s častou změnou gramatiky, protože se kromě samotného parsovacího procesu musí přepisovat i logika přímo s parsováním nesouvisející.

S předchozím bodem souvisí i složitost rozšíření. Protože je v ručně psaných parserech mnoho dodatečného kódu, změna již existujícího řešení vyžaduje přepis i kódu dodatečného, což celý proces komplikuje.

Řešení tvořící ručně psaný parseru je tedy vzhledem k zadání práce a k požadavkům na knihovnu nevhodné. U ručně psaných parserů se nicméně můžeme inspirovat jejich flexibilitou. Knihovna nemůže poskytnout tak velkou flexibilitu během procesu parsování, protože CYK algoritmus toho není schopen. Knihovna nicméně může poskytnout dostatečnou flexibilitu při zpracování atributů nebo obecněji v pozdějších fázích procesu parsování.

3.3.2 Generované parsery

Použití vygenerovaných parserů [9] je další ze způsobů, jakým parser vytvořit. Principem této metody je vytvoření gramatiky v notaci určené nástrojem. Tento soubor je následně zpracován a nástroj vygeneruje zdrojové kódy překladače. Ty musí být zpravidla dále zkompileovány k dosažení spustitelného souboru a tedy fungujícího překladače.

Způsob, jakým jsou generátory používány, se liší v závislosti na nástroji. Obecné vlastnosti, ve kterých se mohou generátory lišit, jsou popsány v následujících bodech.

- Parsovací algoritmus – tedy jeden z výše zmíněných algoritmů (LALR, SLR, $LR(k)$ a další). Některé nástroje (například Bison [16]) podporují několik různých algoritmů.
- Notace gramatiky – znamená jak je vstupní gramatika zapsána. Některé nástroje vyžadují zápis gramatiky ve speciální syntaxi v tzv. rozvinuté Backusova–Naurova formě (EBNF [22]). K té je zpravidla přidán kód (podle generovaného jazyka), který je do výsledného zdrojového kódu kompilery začleněn.

Na druhé straně existují nástroje, které dovolují definovat gramatiku programovacím jazykem [21]. Pro uživatele z toho vyplývá známá syntaxe a tím snadné použití. Bohužel ale zabraňuje možnosti generovat překladač v jiných programovacích jazycích, protože i gramatika samotná je na jazyk navázána.

- Umístění sémantiky – zda je gramatika a sémantika oddělena v samostatných souborech. V případě, kdy je sémantika oddělena od gramatiky, lze překladač vygenerovat pro různé programovací jazyky (v případě že to nástroj dovoluje) pouhou výměnou sémantických pravidel.
- Generovaný jazyk – tzn. zda je nástroj schopen generovat překladač ve více programovacích jazycích. Přístup se dále liší v tom, zda je nutné na začátku vývojového procesu zvolit programovací jazyk a používat jej po celou dobu vývoje, nebo generátor podporuje paralelní vývoj překladačů v různých programovacích jazycích. Konkrétní přístup z velké části určuje předchozí bod.
- Lexikální analyzátor – samotný překladač zpravidla potřebuje před zpracováním textu lexikální analyzátor. Ten zpracuje vstupní text (například preprocessing `#include <stdlib.h>`), odstraní bílé znaky a provádí další operace nevyžadující složitou logiku. Obecně existují tři přístupy.
 - Generované – nástroj na základě gramatiky vygeneruje lexikální analyzátor.
 - Dodané – vygenerovaný překladač potřebuje dodat lexikální analyzátor, který je potřeba napsat.
 - Interní – lexikální analyzátor je již vytvořen nástrojem. Preprocessing vstupní věty musí být vyřešen na úrovni gramatiky.
- Platforma a licence – jako u každého jiného softwaru, nástroje mohou být vázány na určitou platformu a mají licence udávající možnost jejich

použití. V našem případě je tato informace irelevantní, protože se nástroji pouze inspirováme, ale z hlediska použití jde o důležitou informaci.

Generované překladače jsou zpravidla méně efektivní než jejich ručně psané ekvivalenty. Na druhou stranu tvorba generovaného překladače není tak časově náročná a v případě nástrojů, které to podporují, lze vytvořit překladač v různých programovacích jazycích.

3.3.3 Knihovna Pырser

Pырser [33] je knihovna psaná v jazyce Python. Jedná se o knihovnu dovolující snadno popsat gramatiku za účelem parsování. Knihovna se zaměřuje primárně na proces parsování a tvorbu AST. Sémantika v knihovně je podporována pomocí tzv. hooks and nodes, ale jejich použití je komplikovanější, protože se kvůli těmto konstrukcím musí upravovat zápis gramatiky.

Pырser používá pro popis gramatiky vlastní jazyk, který je podobný EBNF. Knihovna taktéž neslouží jako obecný parser, ale zaměřuje se na tzv. parsing expression grammar (PEG). PEG umožňuje zapsat gramatiku pomocí rozšiřujících symbolů (`/`, `*`, `+`, `?`, `&`, `!`), které gramatiku zjednodušují a nevyžadují tak časté použití rekurze či dodatečných pravidel.

Jednoduchý příklad, ve kterém parser přijímá slova „hello“ a „world“ si lze prohlédnout na algoritmu 3.1.

```
from pyrser import grammar

class Helloworld(grammar.Grammar):
    entry = "main"
    grammar = """
        main = [{"hello" | "world"}+ eof]
    """

hw = Helloworld()
if hw.parse('hello hello world hello world world'):
    print('Parsed')
```

Kód 3.1: Parsování slov „hello“ a „world“ knihovnou Pырser [33, Introduction section]

Jak lze vidět, knihovna nefunguje jako generátor překladače, ale sama jako překladač slouží. To zvyšuje časovou složitost, protože se gramatika vytváří při každém spuštění programu a není nikde předvytvořena, jako je tomu u generátorů. Na druhou stranu díky tomu je program plně samostatný a nevyžaduje nástroje třetích stran ke spuštění.

Také si lze všimnout, že ačkoliv je gramatika popsána vlastním jazykem, v knihovně se využívají jen a pouze konstrukce jazyka Python. Uživatel

má tedy jistotu, že kdekoliv, kde je správný interpret jazyka Python, bude knihovna fungovat.

Výše uvedené způsoby tvorby překladače jsou brány jako inspirace pro vlastní tvorbu knihovny.

3.4 Návrh knihovny

Na základě požadavků a existujících řešení již můžeme rozhodnout o základních konceptech, které knihovna musí splňovat. Ačkoliv jsou požadavky již sepsány, uvedeme hlavní body, které ve velké míře ovlivňují samotný návrh knihovny.

- Na základě požadavku **P.r2** jsme již rozhodli, že knihovna nemůže záviset na externích nástrojích. Z tohoto hlediska bude knihovna podobná knihovně Pырser, která gramatiku vytváří (resp. zpracovává) až za běhu aplikace.
- Knihovna bude využívat pouze vlastnosti poskytované jazykem Python.
- Podle požadavku **P.f1** musí knihovna být schopná reprezentovat libovolnou gramatiku.
- Reprezentace sémantiky v knihovně bude formou Python kódu.

3.4.1 Rozdělení knihovny

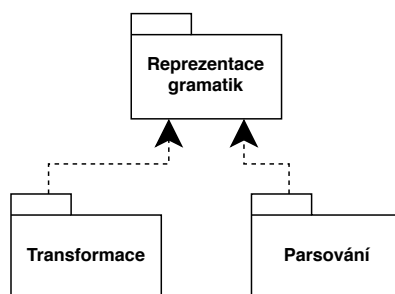
Knihovna bude rozdělena do tří nezávislých částí. Python takové části označuje jako moduly, a proto je tak dále v práci budeme označovat i my.

1. Modul pro reprezentaci gramatiky.
2. Modul pro transformaci gramatik.
3. Modul pro parsování.

Rozdělením na tři nezávislé moduly je zajištěna nízká provázanost uvnitř knihovny. Jednotlivé operace jsou nezávislé na ostatních a tím je dosažena maximální možná flexibilita (viz. obrázek 3.1).

3.4.2 Reprezentace gramatiky

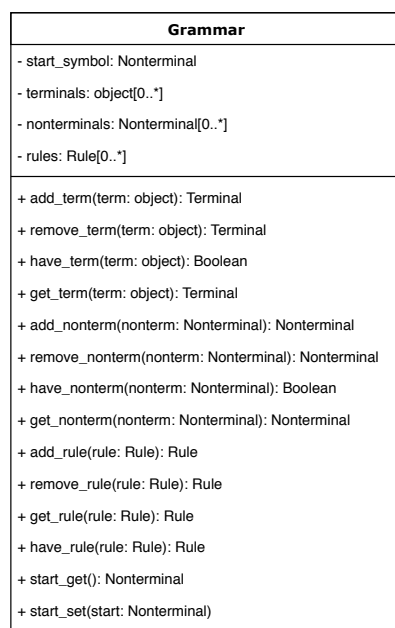
Modul pro reprezentaci gramatik bude obsahovat logiku, která pracuje s gramatikou. Obsahuje základní třídy sloužící k reprezentaci gramatiky, pravidel, neterminálů a terminálů.



Obrázek 3.1: Diagram modulů a jejich závislost



Obrázek 3.2: Třídy v modulu reprezentující gramatiku

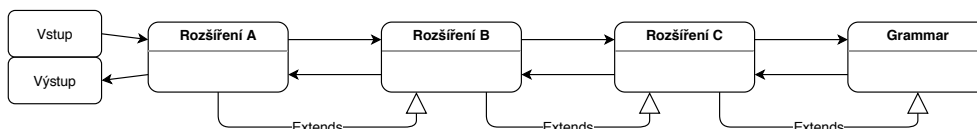


Obrázek 3.3: Rozhraní třídy reprezentující gramatiku

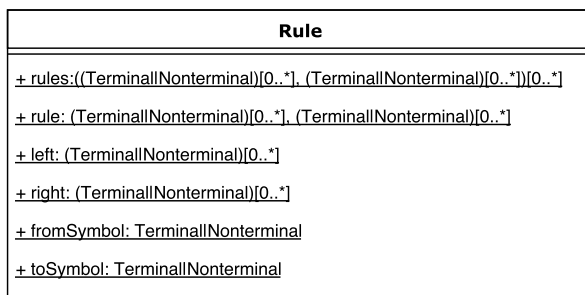
Třída reprezentující gramatiku bude jednoduchý kontejner, který implementuje tzv. CRUD (Create–Read–Update–Delete) operace, tedy operace umožňující přidání, zjištění stavu, aktualizaci a smazání prvku. Rozhraní je na obrázku 3.3.

Třída `Grammar` poskytuje pouze funkcionalitu nutnou k fungování knihovny. Rozhraní bude dále rozšířeno dědičností a vzorem *Pipes and Filters* (obrázek 3.4) [29]. Zděděné třídy mohou poskytovat širší rozhraní nebo větší svobodu v parametrech díky dynamickému typování jazyka Python.

3. ANALÝZA EXISTUJÍCÍCH ŘEŠENÍ A NÁVRH KNIHOVNY



Obrázek 3.4: Rozšíření gramatiky metodou Pipes and Filters



Obrázek 3.5: Rozhraní třídy Rule

Terminálem může být libovolný objekt jazyka Python, který je hashovatelný. Třída `Terminal` slouží pouze pro interní reprezentaci terminálu. Z instance lze původní terminál získat metodou `symbol()`. Operace s gramatikou nějakým způsobem pracující s terminály vracejí instanci tohoto typu (pokud není uvedeno jinak). Terminály budou mezi sebou porovnávány pomocí jejich hash hodnot, není tedy nutné, aby se vstupní slovo skládalo pouze z terminálů obsažených v gramatice. Jediné, co musí objekty splňovat, je dodržení stejných hash hodnot.

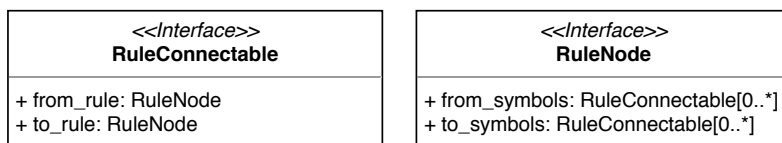
Třída `Nonterminal` bude sloužit jako báze pro všechny neterminály použité v gramatice. Ačkoliv tato třída nevyžaduje žádnou implementaci, právě dědění z této třídy odlišuje neterminály a terminály.

Poslední třída, která bude v modulu použita, je třída `Rule`. Ta je postavena, podobně jako třída `Grammar`, na návrhu *Pipes and Filters*. Základní rozhraní bude obstarávat pouze definici pravidla a jeho rozhraní je na obrázku 3.5. Celé rozhraní je vytvořeno pomocí statických vlastností. Vlastnost, na rozdíl od atributu, je možné dopočítat až ve chvíli, kdy je o ni požádáno. Uživatel si zvolí, který ze zápisu použije a ostatní vlastnosti se automaticky vypočítají (viz. obrázek A.3).

Knihovna kromě toho obsahuje objekt reprezentující ϵ . Python zařídí, že hash tohoto objektu se nemění po celou dobu běhu programu. To nám poskytuje prostředek, jak ϵ identifikovat.

Výše uvedené rozhraní sloužilo pouze pro popis gramatiky. K dodání sémantiky a ke tvorbě AST využijeme již definované třídy.

AST bude tvořen **instancemi** definovaných tříd. Terminály budou reprezentovány instancí třídy `Terminal`, zatímco neterminály uživatelem definova-



Obrázek 3.6: Rozhraní pro tvorbu AST

nými třídami. AST přitom bude tvořen i instancemi pravidel, které uživatel nadefinuje. Každý neterminál (resp. terminál) bude mít odkaz na pravidlo, které se na něj přepsalo, stejně tak odkaz na pravidlo, na které se přepisuje (pokud takové pravidlo existuje). Podobně bude mít pravidlo pole symbolů, ze kterých se přepisuje, a pole symbolů, na které se přepisuje. Díky tomu lze reprezentovat syntaktický strom i pro neomezené gramatiky. Příklad takového stromu lze nalézt na obrázku A.4.

Protože instancujeme typy, které si uživatel vytvořil, přidáme-li ke třídám atributy nebo metody, budou tyto atributy či metody přístupné a použitelné nad AST. Tím můžeme pravidlům a eventuálně i neterminálům či terminálům přidat sémantický význam. Konkrétní použití bude demonstrováno v další kapitole.

Dříve, než popíšeme zbylé dva moduly, stručně shrneme výsledky dosavadního návrhu. Návrh celého modulu je poté zobrazen na obrázku A.5.

- Základní implementace gramatiky bude poskytovat funkcionalitu nezbytně nutnou k fungování knihovny.
- Rozšíření funkcionality bude probíhat zděděním ze základních tříd a využíváním jejich rozhraní.
- Terminálem může být libovolný objekt jazyka Python, který je hashovatelný. Ekvivalence se porovnává pomocí hashe.
- Neterminály se budou vytvářet poděděním ze třídy `Nonterminal`.
- Pravidla se budou vytvářet poděděním ze třídy `Rule`. Tvar pravidla bude určen statickými vlastnostmi.
- V AST budou uživatelem vytvořené třídy instanciovány.

3.4.3 Transformace gramatik

Nyní, když jsme gramatiku nadefinovali, můžeme navrhnout operace, které ji budou transformovat. Všechny transformace by se měly chovat jako tzv. pure function (čistá či pravá funkce) – funkce by měla záviset jen a pouze na svých parametrech a návratová hodnota funkce je pro stejné parametry vždy stejná.

Ideálním řešením tohoto problému by bylo funkcionální programování. Tento přístup je ovšem nevhodný v kontextu naší aplikace, protože by vedl ke

zvýšení nejen komplexnosti kódu, ale i zvýšení časové a paměťové náročnosti knihovny [13].

Po transformačních funkcích budeme alespoň požadovat standardizované rozhraní. Prvním parametrem bude vždy gramatika, následovaná zbylými parametry funkce. Každá transformační funkce musí navíc obsahovat nepovinný pojmenovaný atribut `transform_grammar`, který bude udávat, zda se má modifikovat gramatika z parametru, nebo zda se má vytvořit kopie, která se bude modifikovat. Výchozí hodnotou parametru bude hodnota `False`, tedy původní gramatika zůstane nezměněna. Návratovou hodnotou funkce je modifikovaná gramatika, a to i v případě, že se transformovala gramatika z parametru.

Funkce budou zastřešeny statickými metodami se stejnou deklarací, jako mají funkce samotné. Tyto statické metody budou rozděleny mezi třídy, které budou reprezentovat transformace z hlediska typu gramatiky. Jedná se pouze o logické rozdělení, které nemá z hlediska funkcionality žádný význam.

Parsovací operace budou v samostatném modulu. Definovat v jejich případě pevné rozhraní nedává smysl, protože různé parsovací algoritmy mohou vyžadovat různé vstupy. Podle aktuálních požadavků předpokládáme, že vstupem bude gramatika a vstupní slovo. Vzhledem k tomu, že zbytek knihovny není na tomto rozhraní závislý, je čistě na programátorovi parseru, jaké rozhraní zvolí.

Výstupem parsovací funkce by měl být AST, tedy instanciované terminály, neterminály, pravidla a jejich spojení do stromu, jako je demonstrováno na obrázku A.4.

V případě, že je vstup neplatný, je čistě na programátorovi parseru, jakým způsobem o této situaci informuje uživatele knihovny. Chybových stavů může být velký počet (vezneme-li navíc v potaz různé parsovací algoritmy) a jejich obsažení v knihovně není možné. V knihovně bude upřednostněno vyvolání výjimky s bližšími informacemi o problému.

Realizace

Následující kapitola se zabývá přípravou projektu, implementací, testováním a nakonec publikováním knihovny. Naší snahou v této části práce je popsat fungování knihovny a proces jejího vývoje. Algoritmy používané knihovnou jsou buď velmi jednoduché – a tedy postačí jejich textový popis, nebo jsou již z velké části popsány pseudokódem v dřívějších kapitolách. Z tohoto důvodu se budeme snažit minimalizovat ukázky zdrojového kódu za účelem přehlednosti a konzistence práce. Kompletní zdrojový kód lze nalézt na přiloženém médiu či online na portálu GitHub [17].

4.1 Příprava prostředí

Vývojové prostředí a metodika vývoje přímo ovlivňují, jakým způsobem software vzniká a jakým způsobem mezi sebou vývojáři sdílejí změny.

Prostředí jazyka Python bylo zvoleno poslední stabilní vydání, tedy verze 3.6.4. Pod tímto prostředím je knihovna vyvíjena a testována během vývoje, nicméně v rámci CI je knihovna otestována pod dalšími prostředími jazyka Python, a to až do verze 3.3, protože se jedná o poslední podporovanou verzi v Linuxové distribuci Fedora [14].

Během vývoje knihovny bude použit nástroj *Git*. Jedná se o nástroj široce používaný [42] a většině vývojářů známý, který zároveň vyřeší požadavek **P.s1**. V souladu s požadavky **P.u6** a **P.u4** budou zdrojové kódy umístěny veřejně na portálu GitHub [17], který je většině vývojářů taktéž známý.

K dodržení požadavku **P.r3** byl zvolen nástroj TravisCI [40]. Ten je pro otevřený software (open-source), kterým naše knihovna je, zcela zdarma. Konfigurace probíhá pouze jediným souborem „*.travis.yml*“. Tento nástroj spustí všechny testy nad knihovnou v námi definovaných prostředích a v případě chyby tuto skutečnost nahlásí prostřednictvím e-mailu. TravisCI je taktéž integrován s GitHubem, takže při libovolné změně zdrojového kódu jsou všechny testy spuštěny automaticky a jejich výstup vložen na GitHub. čímž je splněn požadavek **P.r4**.

V případě verzování knihovny bylo zvoleno sekvenční verzování se třemi pozicemi [35].

- Major – „*MAJOR version when you make incompatible API changes*“ [35] – major verze se mění při tzv. breaking changes, tedy při změnách, kvůli nimž nelze starý software, fungující se starou verzí knihovny, použít s její verzí novou. Obecně je snahou měnit major verzi co nejméně, pokud možno vůbec.
- Minor – „*MINOR version when you add functionality in a backwards-compatible manner*“ [35] – minor verze se mění se při vylepšení softwaru. Minor verze přináší uživatelům novou funkcionalitu, ale je dodrženo stejné rozhraní. Starý software tak může fungovat i s novější verzí knihovny, než pro který byl napsán.
- Patch – patche jsou změny knihovny za účelem opravy chyb. Na rozdíl od minor verze nepřináší novou funkcionalitu (eventuálně pouze minimální), ale spíše se soustřeďuje na opravu chyb resp. interní vylepšení knihovny.

Verze jsou psány za sebe a odděleny tečkou – `<major>.<minor>.<patch>` (například `1.2.0`).

Dříve, než se pustíme do samotné implementace, ještě musíme zvolit název knihovny resp. jednotlivých modulů. Název je důležitý pro konečné publikování aplikace do online repositáře, ze kterého si knihovnu mohou stáhnout uživatelé. Pro základní modul reprezentující gramatiku byl zvolen název *grammpy*, pro jejich transformace modul *grammpy-transforms*. Modul obsahující jednotlivé parsery bude pojmenován *pyparsers*. Zdrojové kódy budou umístěny online a to na portálu GitHub [17].

4.2 Reprezentace gramatik

Podoba modulu pro reprezentaci gramatik byla z velké části již dána návrhem knihovny. Základem modulu je třída `Grammar`, která gramatiku uchovává v interních strukturách a poskytuje ke gramatice rozhraní.

Třída `Grammar` byla implementována s ohledem na zvolenou architekturu *Pipes and Filters*. Logika je rozdělena do několika vrstev, kde každá vrstva je reprezentována třídou.

- `RawGrammar` – poskytuje základní rozhraní pro manipulaci s gramatikou. Poskytuje CRUD operace pro terminály, neterminály, pravidla a počáteční symbol. Provádí také validaci parametrů, tedy že pravidlo dědí ze třídy `Rule` a neterminál ze třídy `Nonterminal`.
- `StringGrammar` – zpracovává vstupní parametry v případě, kdy se jedná o text. Text je v Pythonu iterovatelný, tedy jde s ním zacházet jako

s polem. To je nežádoucí, protože u pole očekáváme jeho projití a přidání všech prvků v poli, zatímco text považujeme za dále nedělitelný. Třída `StringGrammar` se stará o to, že text nebude brán jako pole.

- `MultipleRulesGrammar` – z návrhu knihovny lze vytvořit třídu reprezentující několik pravidel současně. To je komplikace při vnitřní reprezentaci, protože následně můžeme chtít odstranit pouze jedno z pravidel. Třída `MultipleRulesGrammar` tyto pravidla rozdělí do samostatných tříd, které následně vloží do gramatiky. Díky této logice lze s pravidly manipulovat jednotlivě.
- `PrettyApiGrammar` – rozšiřuje rozhraní knihovny o některé metody, které jsou založeny nad standardním rozhráním. Jedná se především o zkrácení zápisu a eventuálně standardizaci parametrů.
- `RulesRemovingGrammar` – je-li z gramatiky odstraněn neterminál nebo terminál, očekávali bychom také odstranění všech pravidel, které tento terminál resp. neterminál obsahují. Třída `RulesRemovingGrammar` obstarává tuto logiku.
- `CopyableGrammar` – tato třída obstarává implementaci kopírování, tedy hluboké kopie (deep copy) a mělké kopie (shallow copy).
- `Grammar` – jedná se o zastřešující třídu, která skrývá vnitřní implementaci. Budou-li výše zmíněné třídy gramatiky změněny nebo přepsány, jejich skrytím za třídu `Grammar` je zajištěna kompatibilita se starými aplikacemi (za předpokladu, že se nezmění rozhraní).

Stejný přístup byl zvolen i pro třídu `Rule` reprezentující pravidlo.

- `BaseRule` – tato třída obsahuje základ pravidel. Má nadefinované statické vlastnosti a logiku s dopočítáváním zbylých vlastností (viz kapitola 3.4). Třída dále obsahuje definici hashovací funkce a porovnávacího operátoru.
- `ValidationRule` – třída `ValidationRule` obsahuje logiku související s validací pravidel. Tato logika se volá ve chvíli, kdy je pravidlo vkládáno do gramatiky. Kontroluje se tak, že jsou pravidla nadefinována syntakticky správně, ale také že všechny symboly použité v pravidlu jsou v gramatice k dispozici.
- `InstantiableRule` – jak bylo řečeno v sekci návrhu, pravidla se budou instanciovat a budou sloužit jako entita v AST. Tato třída definuje potřebné atributy a metody potřebné k tomuto účelu.
- `Rule` – stejně jako třída `Grammar`, je třída `Rule` pouze abstrakcí, která odděluje vnitřní implementaci od jejího použití.

Modul kromě výše zmíněných tříd obsahuje ještě třídy další. Některé ze tříd jsou pomocné a jsou určeny pouze k interním účelům modulu. Tyto třídy nejsou exportovány ven z modulu a pro uživatele knihovny se jeví jako nedostupné.

- **RuleConnectable** – je rozhraní umožňující propojit entity pomocí pravidel.
- **Terminal** – je třída reprezentující terminál v AST, resp. u operací s terminály pracujících. Třída dědí ze třídy **RuleConnectable** a v aktuální implementaci poskytuje jen metodu `symbol`, která vrátí skutečný symbol do gramatiky vložený.
- **Nonterminal** – třída **Nonterminal** neposkytuje žádnou implementaci, pouze dědí ze třídy **RuleConnectable**. Tato třída slouží jako bazová třída pro ostatní neterminály, které jsou identifikovány právě dědičností z této třídy.
- **WeakList** – je interní třída představující pole se slabými (*weak*) odkazy. Python používá pro uvolnění paměti garbage collector (GC), který používá tzv. reference counting. Ten je náchylný na odkazovací smyčky, které se v programu vyskytují (pravidlo má odkaz na symbol a symbol má odkaz zpět na pravidlo). Použitím slabých odkazů se tento problém eliminuje. **WeakList** je použit interně v implementaci třídy **InstantiableRule**.
- **HashContainer** – tato třída slouží jako jednoduchá množina, založená na hash hodnotách. Na rozdíl od výchozí implementace množiny (která je také založená na hash hodnotách) má **HashContainer** změněné rozhraní. Použita je pro uchovávání terminálů, neterminálů a pravidel ve třídě **RawRule**.

Nakonec má modul nadefinovanou hierarchii výjimek. Všechny výjimky dědí ze třídy **GrammpyException**, která sama dědí ze třídy **Exception**. Kompletní hierarchie je zachycena na obrázku A.6.

- **GrammpyException** – základní, společná třída.
- **CannotConvertException** – konverze neproběhla v pořádku, dědí také z **ValueError**.
- **NotASingleSymbolException** – na místě, kde má být jeden symbol, je jich uvedeno více.
- **NotRuleException** – parametr není pravidlo, mimo **GrammpyException** dědí také z **TypeError**.

- `NotNonterminalException` – parametrem není neterminál, dědí také z `TypeError`.
- `RuleException` – pravidlo je nadefinováno syntakticky nesprávně nebo nevalidně, kromě `GrammpyException` dědí také ze třídy `ValueError`.
- `RuleSyntaxException` – špatná syntaxe pravidla.
- `TerminalDoesNotExistsException` – použitý terminál v gramatice neexistuje.
- `NonterminalDoesNotExistsException` – použitý neterminál v gramatice neexistuje.
- `UselessEpsilonException` – ϵ symbol je špatně použit.
- `CantCreateSingleRuleException` – rozdělení na jednotlivé pravidla skončilo chybou.
- `RuleNotDefinedException` – pravidlo není nadefinováno.
- `TreeDeletedException` – odkazovaný element AST byl již smazán.

Výsledek implementace je znázorněn v diagramu na obrázku A.7, resp. na obrázku A.6, kde je znázorněna hierarchie výjimek.

4.3 Transformace a parsování

Algoritmy pro transformaci jsou pouze implementací pseudokódů, které byly ukázány v kapitole o operacích s gramatikami. Z tohoto důvodu bude kód jednotlivých algoritmů vynechán. Kompletní implementace je k dispozici na příloženém mediu nebo online [17].

Každý z algoritmů zpravidla definuje vlastní neterminály resp. pravidla, kterými nahradí modifikovaný neterminál resp. pravidlo. Díky jejich typu lze poté provést zpětnou transformaci AST. Dále následuje popis algoritmů, které jsou v knihovně implementovány, zároveň s jejich typy.

- `remove_nongenerating_nonterminals` – odstraní negenerující neterminály z gramatiky. Algoritmus nedefinuje žádné vlastní třídy, protože se jedná pouze o optimalizační techniku.
- `remove_unreachable_symbols` – odstraní nedostupné symboly. Opět se jedná pouze o optimalizační techniku, a tak algoritmus nedefinuje vlastní třídy.

- `remove_rules_with_epsilon` – odstraní ε pravidel. Algoritmus definuje vlastní třídu pro pravidlo – `EpsilonRemovedRule`. V této třídě je uloženo, ve kterém pravidle byl symbol přepsán na ε , jeho pozici a dále sekvence pravidel, které k ε symbolu vedly.

Algoritmus pro zpětnou transformaci na místě, kde je pravidlo typu `EpsilonRemovedRule` použito, vytvoří původní pravidlo. Poté vytvoří sekvenci pravidel, která vedla k ε symbolu, a vloží ji na pozici určenou indexem.

- `remove_unit_rules` – odstraní jednoduché pravidla z gramatiky. Algoritmus definuje pravidlo `ReducedUnitRule`, které nahradí pravidla, na jejichž pravé straně se vyskytuje symbol vedoucí k jednoduchému pravidlu. Třída `ReducedUnitRule` si podobně jako `EpsilonRemovedRule` ukládá, ze kterého pravidla vznikla a sekvenci jednoduchých pravidel, které k transformaci vedly.

Algoritmus pro zpětnou transformaci probíhá taktéž obdobně – nejdříve je vytvořena sekvence jednoduchých pravidel, poté konečné pravidlo a tato struktura je vložena na patřičné místo v AST.

- `transform_to_chomsky_normal_form` – transformuje gramatiku do CNF. Tento algoritmus definuje několik tříd, některé z nich pro interní potřeby. Neterminály definuje tři:
 - `ChomskyNonterminal` – bazová třída pro zbylé dva neterminály,
 - `ChomskyGroupNonterminal` – představuje neterminál reprezentující více neterminálů,
 - `ChomskyTermNonterminal` – představuje neterminál přímo derivovatelný na terminál.

K těmto neterminálům algoritmus definuje i patřičná pravidla.

- `ChomskyRule` – bazová třída pro zbylé pravidla.
- `ChomskySplitRule` – pravidlo vzniklé rozdělením pravidla do dvou neterminálů. Druhým symbolem pravé strany pravidla je vždy pravidlo `ChomskyGroupNonterminal`, který reprezentuje všechny symboly pravé strany s výjimkou prvního.
- `ChomskyRestRule` – reprezentuje zbylou část pravidla neterminálu `ChomskyGroupNonterminal`. Jedná se pouze o dočasný objekt, protože v další iteraci je i toto pravidlo rozděleno na dva neterminály – vznikne z něj tedy opět pravidlo typu `ChomskySplitRule`.
- `ChomskyTerminalReplaceRule` – slouží jako náhrada pravidla na jehož pravé straně jsou dva symboly, ale jeden z nich je terminál. Tento symbol je nahrazen `ChomskyTermNonterminal` a dále je přidáno pravidlo typu `ChomskyTermRule`.

- `ChomskyTermRule` – je pravidlo, které přímo derivuje neterminál na terminál.

Každý z definovaných neterminálů resp. terminálů definují atributy, na jejichž základě lze transformovat AST.

Modul kromě toho definuje pomocné třídy `Manipulations` a `Traversing`, které zjednodušují práci s AST. Třída `Manipulations` poskytuje rozhraní pro nahrazení pravidla v AST jiným pravidlem, resp. nahrazení terminálu či neterminálu jiným symbolem. Třída `Traversing` dále definuje metody pro procházení AST. V základu implementuje procházení metodou `pre-order` a `post-order`, ale také abstraktnější verzi pro procházení na základě callbacku.

Kompletní návrh modulu lze nalézt na obrázku A.8. Použitá pravidla jsou poté na obrázku A.9 a neterminály na obrázku A.10.

Modul `pyparsers` se skládá pouze z jedné funkce – `cyk`. Tato funkce provede CYK algoritmus a navrátí AST. Modul obsahuje další pomocné třídy, které jsou interně použity při algoritmu.

- `Point` – reprezentuje bod o souřadnicích x a y . Interně se nejedná o třídu, ale o tzv. `namedtuple` [34].
- `Field` – reprezentuje pyramidovou strukturu, nad kterou operuje CYK algoritmus.
- `PlaceItem` – zastřešuje strukturu uchováající použité pravidlo a jeho potomky.

Algoritmus si nejprve vytvoří instanci typu `Field`. Poté si vytvoří dva překladové slovníky – jeden pro pravidla, která se přímo derivují na neterminály, a druhý, jehož klíčem jsou dva neterminály a hodnotou je vždy pravidlo na tyto neterminály přímo derivovatelné. Na základě prvního slovníku jsou poté nalezeny neterminály, které se přímo derivují na vstupní terminály, a tyto neterminály jsou vyplněny do posledního řádku tabulky. Algoritmus dále probíhá stejně, jak je to uvedeno v jeho pseudokódu.

Po dokončení algoritmus zkontroluje, zda je počáteční symbol v prvním řádku pole (viz obrázek 2.3) a pokud ne, vyvolá výjimku. V případě, kdy se počáteční symbol v prvním řádku tabulky vyskytuje, algoritmus pokračuje s tvorbou AST. Díky struktuře `PlaceItem`, ve které si algoritmus ukládá použitá pravidla, lze AST vytvořit bez dalšího procházení vstupního slova nebo gramatiky. Zdrojové kódy jsou opět na příloženém mediu a online.

Vzhledem k tomu, že rozhraní modulu je pouze jedna funkce, není zde k tomuto modulu zobrazen diagram.

4.4 Testování

Pro otestování knihovny byl zvolen framework `unittest`. Tento framework je dostupný jako Python modul přímo ve výchozí instalaci, není tedy třeba instalovat knihovny třetích stran. Jedná se o testovací framework určený především k jednotkovým testům (unit testům). Vzhledem k faktu, že implementujeme pouze knihovnu a nikoliv celou aplikaci, je jednotkové (resp. integrační) testování dostatečným ověřením jejího správného fungování. Systémové či uživatelské testy vzhledem k povaze problému ani nelze provádět. Knihovna je otestována více než 450 jednotkovými testy.

Pokrytí kódu (anglicky code coverage) [4] je analytický přístup ověřující, že všechny kód, ze kterého se software skládá, byl alespoň jednou spuštěn. Při vysokém pokrytí kódu máme jistotu, že téměř celá aplikace byla spuštěna a v rámci tohoto běhu funguje jak má. Vysoké pokrytí nám nicméně nezaručuje, že software funguje správně, protože mohou nastat situace, se kterými není v kódu počítáno. Nástroje určující pokrytí kódu detekují, že byl kód spuštěn, ale samotnou chybu neobjeví. Software, jenž obsahuje grafické rozhraní, je zpravidla pokryt hůře, protože testování grafického prostředí je náročné a vyžaduje speciální nástroje. V našem případě můžeme zvolit požadované pokrytí větší (viz požadavek **P.r2**), protože se jedná pouze o knihovnu, kterou lze v běžných případech otestovat téměř celou [4]. Během vývoje se stav pokrytí neustále mění v závislosti na fázi vývoje a vydání. Knihovna se nicméně řídí požadavkem **P.r2** a její pokrytí neklesá pod 90 %. Publikované verze knihovny mají zpravidla pokrytí kódu větší, tradičně 100 %.

Jako nástroj měřící pokrytí kódu byl zvolen framework (resp. modul) `Coverage.py` [10], který je napsán v jazyce Python, a pro své fungování vyžaduje pouze instalaci tohoto modulu. Pokrytí kódu je automaticky generováno na základě jednotkových testů. Data jsou poté přenesena do nástroje `Coveralls` [11], který podle dodaných dat generuje přehledné reporty. Ty obsahují celkové pokrytí kódu, pokrytí jednotlivých tříd resp. souborů a samozřejmě také změnu pokrytí mezi jednotlivými verzemi softwaru. Nástroj je opět pro otevřený software zdarma a díky integraci s nástroji `GitHub` a `TravisCI` jsou reporty generovány pro každou verzi softwaru automaticky.

Demonstrace knihovny na příkladech

V následující kapitole je knihovna použita pro parsování aritmetických výrazů, regulárních výrazů a lambda kalkulu. Základ aplikace a postupy zůstávají stále podobné prvnímu příkladu. Z tohoto důvodu budou pro jednotlivé příklady ukázány pouze klíčové části programu. Kompletní zdrojové kódy jsou na přiloženém mediu.

5.1 Parser aritmetických výrazů

Nyní již máme všechny prostředky potřebné pro tvorbu parseru. V této sekci budeme demonstrovat fungování programu na parseru aritmetických výrazů, tedy výrazů podporujících operace $+$, $-$, $*$, $/$ a použití závorek.

Budeme vycházet z gramatiky $G = (N, T, S, R)$ kde

$$\begin{aligned}
 N &= \{Number, MultipleDivide, PlusMinus\} \\
 T &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, (,)\} \\
 S &= PlusMinus \\
 R &= \{Number \rightarrow 0, Number \rightarrow 1, \dots, Number \rightarrow 9, \\
 & \quad Number \rightarrow Number0, \dots, Number \rightarrow Number9, \\
 & \quad PlusMinus \rightarrow PlusMinus + MultipleDivide, \\
 & \quad PlusMinus \rightarrow PlusMinus - MultipleDivide, \\
 & \quad PlusMinus \rightarrow MultipleDivide, \\
 & \quad MultipleDivide \rightarrow MultipleDivide * Number, \\
 & \quad MultipleDivide \rightarrow MultipleDivide / Number, \\
 & \quad MultipleDivide \rightarrow Number, \\
 & \quad Number \rightarrow (PlusMinus)\}
 \end{aligned}$$

Nejdříve nadefinujeme neterminál, který bude mít jeden atribut – vytvoříme tak atributovanou gramatiku (kód 5.1). Z toho neterminálu budou dědit zbývající neterminály.

```
class Common(Nonterminal):
    def __init__(self):
        self._attribute = None

    @property
    def attribute(self):
        if self._attribute is None:
            self.to_rule.compute()
        return self._attribute

    @attribute.setter
    def attribute(self, value):
        self._attribute = value
```

Kód 5.1: Definice atributovaného neterminálu

Jak je v kódu vidět, atribut se počítá až ve chvíli, kdy je o něj požádáno (tzv. lazy evaluation). Přitom se zavolá metoda `compute` na pravidlu, které neterminál přepisuje. V tomto případě jsou totiž všechny atributy syntetizované. Nyní nadefinujeme pravidla a jejich sémantiku. Vzhledem k množství pravidel je nebudeme vypisovat všechny, ale jako příklad uvedeme pouze $MultipleDivide \rightarrow MultipleDivide/Number$.

```
class DivideApplied(Rule):
    fromSymbol = MultipleDivide
    right = [MultipleDivide, DivideOperator, Number]

    def compute(self):
        parent = self.from_symbols[0]
        left_child = self.to_symbols[0]
        right_child = self.to_symbols[2]
        result = left_child.attribute / right_child.attribute
        parent.attribute = result
```

Kód 5.2: Pravidlo $MultipleDivide \rightarrow MultipleDivide/Number$

Na základě terminálů, neterminálů a pravidel poté můžeme nadefinovat gramatiku (algoritmus 5.3).

Nyní použijeme gramatiku ke zpracování výrazu $10+(5*2+4)/2*4$. V rámci tohoto příkladu není vytvořen lexikální analyzátor, proto nebude vstupní slovo parsováno z příkazové řádky, ale bude zapsáno přímo v programu. Před samotným parsování ještě musíme provést transformaci gramatiky do CNF a po parsování provést zpětnou transformaci AST (algoritmus 5.4).

```

g = Grammar(terminals=[0, 1, 2, 3, 4, 5, 6, 7, 8,
↳ 9, PlusOperator, MinusOperator,
↳ MultipleOperator, DivideOperator, LeftBracket, RightBracket],
nonterminals=[Number, MultipleDivide, PlusMinus],
rules=[NumberDirect, NumberCompute, DivideApplied,
↳ MultipleApplied, NoDivideMultiple, MinusApplied,
↳ PlusApplied, NoPlusMinus, BracketsApplied],
start_symbol=PlusMinus)

```

Kód 5.3: Definice aritmetické gramatiky

```

ContextFree.remove_useless_symbols(g, transform_grammar=True)
ContextFree.remove_rules_with_epsilon(g, transform_grammar=True)
ContextFree.remove_unit_rules(g, transform_grammar=True)
ContextFree.remove_useless_symbols(g, transform_grammar=True)
ContextFree.transform_to_chomsky_normal_form(g,
↳ transform_grammar=True)

parsed = cyk(g, [1, 0, PlusOperator, LeftBracket, 5,
↳ MultipleOperator, 2, PlusOperator, 4, RightBracket,
↳ DivideOperator, 2, MultipleOperator, 4])

parsed =
↳ InverseContextFree.transform_from_chomsky_normal_form(parsed)
parsed = InverseContextFree.unit_rules_restore(parsed)
parsed = InverseContextFree.epsilon_rules_restore(parsed)
parsed = InverseCommon.splitted_rules(parsed)
print(parsed.attribute)

```

Kód 5.4: Proces parsování aritmetických výrazů

5.2 Parser regulárních výrazů

Obecný parser regulárních výrazů má široké uplatnění. Asi nejevidentnější je převod regulárního výrazu do stavového automatu. Stavovým automatem lze poté testovat, zda vstupní slovo vyhovuje nebo nevyhovuje regulárnímu výrazu.

V tomto příkladě budeme nicméně demonstrovat přesně opačné použití – vypsání všech slov (s jistými omezujícími podmínkami), které zadanému regulárnímu výrazu vyhovují. Takový program lze využít například pro prolamování hesel, která mají předem daný formát.

Při formátu regulárních výrazů vycházíme ze zdroje [2]. Program je založen

na gramatice $G = (N, T, S, R)$.

$$\begin{aligned} N &= \{Symb, Concat, Iterate, Or\} \\ T &= \{a, b, c, \dots, y, z, +, *, (,)\} \\ S &= Or \\ R &= \{Or \rightarrow Or + Or, Or \rightarrow Concat, \\ &\quad Concat \rightarrow IterateConcat, Concat \rightarrow Iterate, \\ &\quad Iterate \rightarrow Symb^*, Iterate \rightarrow Symb \\ &\quad Symb \rightarrow (Or), \\ &\quad Symb \rightarrow a, Symb \rightarrow b, \dots, Symb \rightarrow z \} \end{aligned}$$

Na rozdíl od předchozího příkladu nyní ukážeme přístup, kdy nebude sémantika aplikace uložena v pravidlech, ale přímo v neterminálech. Každý z neterminálů bude mít metodu `get`, která vrátí seznam slov, které danému regulárnímu podvýrazu vyhovují. Například pro neterminál *Symb* který by měl pouze pravidlo $Symb \rightarrow a$ by byla metoda implementována pouze jako navrácení hodnoty terminálu `return [self.to_rule.to_symbols[0].s]`. Pro představu ukážeme metodu `get` neterminálu *Concat* (algoritmus 5.5).

```
class Concat(Nonterminal):
    def get(self, i, f):
        if len(self.to_rule.to_symbols) == 1:
            return self.to_rule.to_symbols[0].get(i, f)
        new = []
        left = self.to_rule.to_symbols[0].get(i, f)
        right = self.to_rule.to_symbols[1].get(i, f)
        for l in left:
            for r in right:
                new.append(l + r)
        return new
```

Kód 5.5: Implementace metody `get` neterminálu *Concat*

Parametry *i* a *f* udávají maximální počet iterací, resp. znaky vyplnění na místě, kde má iterace skončit. Tyto parametry jsou relevantní pouze pro neterminál *Iterate*. Jsou využity k omezení délky slova, například pro regulární výraz „ab*(def+xy*z)“ se jako nejdelší slovo vygeneruje „abb...bbxyy...yyz“, je-li $i = 3$ a $f = \dots$.

Běh aplikace je ukázán na obrázku 5.1.

5.3 Interpret lambda kalkulu

V tomto příkladě budeme tvořit interpret lambda kalkulu. Při definici lambda kalkulu vycházíme ze zdroje [20]. Na rozdíl od předchozích příkladů, které se zaměřovaly čistě na parsování, musíme u interpretu lambda kalkulu řešit implementaci interpretu.


```
$ python3 run.py -i 2 -f "___" "(a+b)*c"  
c  
ac  
aac  
a___ac  
c  
bc  
bbc  
b___bc
```

Obrázek 5.1: Ukázka běhu parseru regulárních výrazů

Jedním z problémů je zjednodušení vstupního textu. Ačkoliv by bylo možné parsovat vstup v jeho textové podobě, velmi by se tím komplikovala gramatika. Pro příklad uveďme bílé znaky, které nemají ze sémantického hlediska žádný význam. Také identifikátory mohou nabývat téměř nekonečného počtu hodnot, ale tyto hodnoty jsou z hlediska parsovacího procesu nepodstatné. V aplikaci proto před samotným parsováním použijeme lexikální analyzátor, který převede vstupní text do série tokenů, které teprve budou zpracovány samotným parserem. Budeme tím demonstrovat použití lexikálního analyzátoru ve spojení s vytvořenou knihovnou

Dalším z problémů je samotná interpretace. Aplikace musí nejenom narsovat vstup, ale i správně provádět operace lambda kalkulu (konkrétně α redukci, β redukci a další). Jelikož zmíněné operace nijak nesouvisejí s procesem parsování ani s touto prací, bude použita existující knihovna.

Samotná aplikace bude rozdělena do několika částí.

- Syntaktický analyzátor – jedná se o podprogram převádějící vstupní text na sérii tokenů. Gramatika je nadefinována nad těmito tokeny a nikoliv nad samotným vstupním textem.
- Parser – hlavní část programu z hlediska této práce. Parser převede vstupní sérii tokenů na AST.
- Reprezentace – AST je převeden do struktur vyžadovaných knihovnou pro interpretaci lambda kalkulu.
- Interpretace – reprezentace je interpretována a výstup je zobrazen uživateli.

Sémantická analýza je vyřešena knihovnou *PLY* (Python Lex–Yacc) [32]. Jak název napovídá, knihovna je velmi silně ovlivněna nástroji *Lex* a *Yacc*. Jejím účelem je spojení těchto nástrojů a jejich implementace v jazyce Python. Z knihovny byl použit pouze modul `lex`, tedy modul zprostředkávající lexikální analýzu. Syntaxe je velmi podobná syntaxi klasického nástroje

Lex, pouze s tím rozdílem, že je celý proces definován v nativním Pythonu. Pro ukázkou uvedme vytvoření tokenu představující proměnnou (algoritmus 5.6), která může nabývat libovolné kombinace velkých písmen, malých písmen a jednoduchých uvozovek.

```
def t_VARIABLE(t):
    r'[a-zA-Z\']+'
    t.value = Variable(t.value)
    return t
```

Kód 5.6: Generování tokenu symbolizující proměnnou

Parsování probíhá podobně, jako je tomu v předchozích případech, až na rozdíl identifikace terminálů. Jak bylo řečeno, knihovna terminály porovnává podle jejich hash hodnoty. Budou-li dva objekty mít stejnou hash, budou interpretovány jako ekvivalentní. V našem případě bude hash instance vracet hash třídy – díky tomu lze instanci s libovolnou hodnotou interpretovat jako terminál určité třídy. Ukážeme to na příkladu terminálu reprezentujícím číslo (algoritmus 5.7). Ačkoliv bude na vstupu několika terminálů, každý s jiným atributem `value`, parsovací algoritmus je bude považovat za ekvivalentní a použije adekvátní pravidla.

```
class Number:
    def __init__(self, value):
        self.value = value
    def __hash__(self):
        return hash(Number)
```

Kód 5.7: Terminál reprezentující číslo

Samotná gramatika $G = (N, T, S, R)$ popisující lambda kalkul je k prohlédnutí níže.

$$\begin{aligned}
N &= \{ \text{Nobraceexpression}, \text{Expression}, \text{Expressionbody}, \\
&\quad \text{Lambda}, \text{Parameters} \} \\
T &= \{ \text{Lambdakeyword}, \text{Dot}, \text{Leftbracket}, \text{Rightbracket}, \\
&\quad \text{Number}, \text{Variable}, \text{Parameter} \} \\
S &= \text{Nobraceexpression} \\
R &= \{ \text{Nobraceexpression} \rightarrow \text{Expressionbody}, \\
&\quad \text{Expressionbody} \rightarrow \text{VariableExpressionbody}, \\
&\quad \text{Expressionbody} \rightarrow \text{Variable}, \\
&\quad \text{Expressionbody} \rightarrow \text{NumberExpressionbody}, \\
&\quad \text{Expressionbody} \rightarrow \text{Number}, \\
&\quad \text{Expressionbody} \rightarrow \text{LambdaExpressionbody}, \\
&\quad \text{Expressionbody} \rightarrow \text{Lambda}, \\
&\quad \text{Expressionbody} \rightarrow \text{ExpressionExpressionbody}, \\
&\quad \text{Expressionbody} \rightarrow \text{Expression}, \\
&\quad \text{Expression} \rightarrow \text{LeftbracketNobraceexpressionRightbracket} \\
&\quad \text{Lambda} \rightarrow \text{LeftbracketLambdakeywordParameters}, \\
&\quad \text{DotNobraceexpressionRightbracket}, \\
&\quad \text{Parameters} \rightarrow \varepsilon, \\
&\quad \text{Parameters} \rightarrow \text{ParameterParameters}, \\
&\quad \text{Parameters} \rightarrow \text{Parameter} \}
\end{aligned}$$

Sémantická část gramatiky je tentokrát smíšená – některá sémantická pravidla jsou definována na pravidlech, některá na neterminálech. Protože se jedná pouze o demonstraci knihovny, nebude implementace detailněji popsána. V případě zájmu jsou zdrojové kódy na příloženém mediu.

Pro samotnou reprezentaci a interpretaci knihovny byla zvolena knihovna *lambda_interpreter* [26]. Tato knihovna vznikla jako semestrální práce v předmětu „Programovací paradigmatá“ (BI-PPA) studentkou Simonou Kurňavovou na FIT CVUT. Tato knihovna definuje vlastní struktury pro definici výrazů v lambda kalkulu. Sémantická část gramatiky spočívá právě v převodu AST do struktur vyžadovaných knihovnou *lambda_interpreter*.

Ná závěr je na reprezentaci opakovaně aplikována β redukce a mezivýsledky jsou zobrazeny uživateli. Běh programu lze vidět na obrázku 5.2.

5. DEMONSTRACE KNIHOVNY NA PŘÍKLADECH

```
>>>((lambda x. ((lambda y. (y x)) x y)) z)
((lambda x. ((lambda y. (y x)) x y)) z)
((lambda y. (y z)) z y)
((z z) y)
>>>((lambda var y. ((lambda z. ((lambda f. (z f)) z)) var)) 3 7)
((lambda var y. ((lambda z. ((lambda f. (z f)) z)) var)) 3 7)
((lambda y. ((lambda z. ((lambda f. (z f)) z)) 3)) 7)
((lambda z. ((lambda f. (z f)) z)) 3)
((lambda f. (3 f)) 3)
(3 3)
>>>(lambda x (y). x y)
Invalid input
>>>
```

Obrázek 5.2: Ukázka běhu interpretu lambda kalkulu

Závěr

Cílem práce bylo navrhnout, implementovat a otestovat knihovnu, která je schopná na základě dodané gramatiky parsovat vstup.

Funkcionalita byla patřičně navržena, implementována, otestována a publikována v knihovně *grammpy* [17]. Knihovna přitom splňuje všechny požadavky, které na ni byly kladeny. Správné fungování knihovny je ověřeno více než 450 testy, které pokrývají téměř 100 % kódu knihovny. V práci bylo fungování knihovny demonstrováno na třech příkladech – parser aritmetických výrazů, parser regulárních výrazů a interpret lambda kalkulu.

Knihovna v aktuální implementaci poskytuje rozhraní nutné ke správnému fungování. Jedná se o základní verzi, sloužící primárně k ověření správného návrhu a sběru zpětné vazby pro budoucí vývoj. Knihovna má stále několik oblastí, které lze dále rozvíjet – implementace dalších parsovacích algoritmů nebo rozšíření rozhraní. Je v plánu knihovnu dále rozvíjet i po odevzdání této práce.

Celkově hodnotíme výsledek práce pozitivně. Knihovna splňuje všechny požadavky, které od ní byly očekávány. Během práce na tomto projektu již bylo nad knihovnou postaveno několik aplikací, které byly zmíněny v poslední kapitole. Tyto aplikace jednak demonstrují fungování knihovny, ale také mají jasný účel a dokážeme si představit jejich použití i mimo kontext této práce.

Literatura

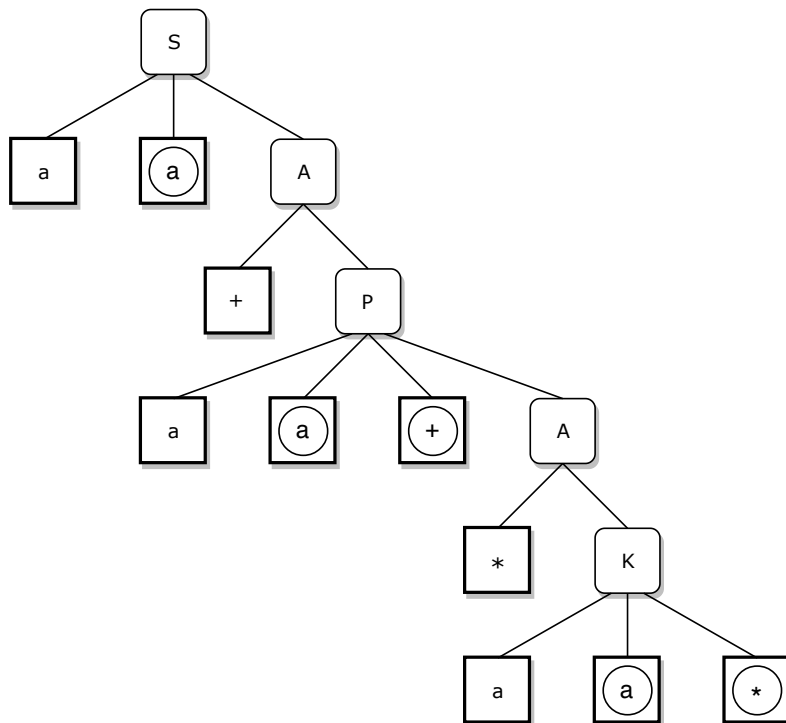
- [1] Abrahams, P. W.: A Final Solution to the Dangling else of ALGOL 60 and Related Languages. *Commun. ACM*, ročník 9, č. 9, Zář 1966: s. 679–682, ISSN 0001-0782, doi:10.1145/365813.365821. Dostupné z WWW: <<http://doi.acm.org/10.1145/365813.365821>>
- [2] Aho, A. V.; Sethi, R.; Ullman, J. D.: *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986, ISBN 0-201-10088-6.
- [3] Aho, A. V.; Ullman, J. D.: Translations on a Context Free Grammar. In *Proceedings of the First Annual ACM Symposium on Theory of Computing, STOC '69*, New York, NY, USA: ACM, 1969, s. 93–112, doi:10.1145/800169.805425. Dostupné z WWW: <<http://doi.acm.org/10.1145/800169.805425>>
- [4] Ammann, P.; Offutt, J.: *Introduction to Software Testing*. New York, NY, USA: Cambridge University Press, první vydání, 2008, ISBN 0521880386, 9780521880381.
- [5] Behkamal, B.; Kahani, M.; Akbari, M. K.: Customizing ISO 9126 quality model for evaluation of B2B applications. *Information and Software Technology*, ročník 51, č. 3, 2009: s. 599 – 609, ISSN 0950-5849, doi:<https://doi.org/10.1016/j.infsof.2008.08.001>. Dostupné z WWW: <<http://www.sciencedirect.com/science/article/pii/S0950584908001109>>
- [6] Blythe, S. A.; James, M. C.; Rodger, S. H.: LLparse and LRparse: Visual and Interactive Tools for Parsing. *SIGCSE Bull.*, ročník 26, č. 1, Břez 1994: s. 208–212, ISSN 0097-8418, doi:10.1145/191033.191121. Dostupné z WWW: <<http://doi.acm.org/10.1145/191033.191121>>
- [7] Chapman, N. P.: *LR PARSING Theory and Practice*. New York, NY, USA: Cambridge University Press, 1987, ISBN 0-521-30413-X.

- [8] Chomsky, N.: On Certain Formal Properties of Grammars. *Information and Control*, ročník 2, č. 2, June 1959: s. 137–167. Dostupné z WWW: <<http://www.diku.dk/hjemmesider/ansatte/henglein/papers/chomsky1959.pdf>>
- [9] Comparison of parser generators. [cit. 2018-09-30]. Dostupné z WWW: <https://en.wikipedia.org/wiki/Comparison_of_parser_generators>
- [10] Coverage.py. [cit. 2018-04-08]. Dostupné z WWW: <<https://coverage.readthedocs.io/en/coverage-4.5.1/>>
- [11] Coveralls. [cit. 2018-04-08]. Dostupné z WWW: <<https://coveralls.io/>>
- [12] Deremer, F. L.: PRACTICAL TRANSLATORS FOR LR(K) LANGUAGES. Technická zpráva, Massachusetts Institute of Technology. Dept. of Electrical Engineering, Cambridge, MA, USA, 1969.
- [13] Disadvantages of purely functional programming. [cit. 2018-04-11]. Dostupné z WWW: <<https://jaxenter.com/disadvantages-of-purely-functional-programming-126776.html>>
- [14] Fedora Multiple Python interpreters. [cit. 2018-04-08]. Dostupné z WWW: <<https://developer.fedoraproject.org/tech/languages/python/multiple-pythons.html>>
- [15] GCC 4.1 release notes. [cit. 2018-04-11]. Dostupné z WWW: <<https://gcc.gnu.org/gcc-4.1/changes.html>>
- [16] GNU bison. [cit. 2018-03-14]. Dostupné z WWW: <<https://www.gnu.org/software/bison/>>
- [17] Grammpy project. [cit. 2018-04-11]. Dostupné z WWW: <<https://github.com/PatrikValkovic/grammpy>>
- [18] H. Younger, D.: Recognition and Parsing of Context Free Languages in Time n³. ročník 10, 02 1967: s. 189–208.
- [19] Havill, J.: *Discovering Computer Science: Interdisciplinary Problems, Principles, and Python Programming*. Chapman & Hall/CRC, 2015, ISBN 148225414X, 9781482254143.
- [20] Hindley, J.; Seldin, J.: *Introduction to Combinators and the Lambda-Calculus*. London Mathematical Society student texts, Cambridge University Press, 1986. Dostupné z WWW: <<https://books.google.cz/books?id=N1RsnQAACAAJ>>

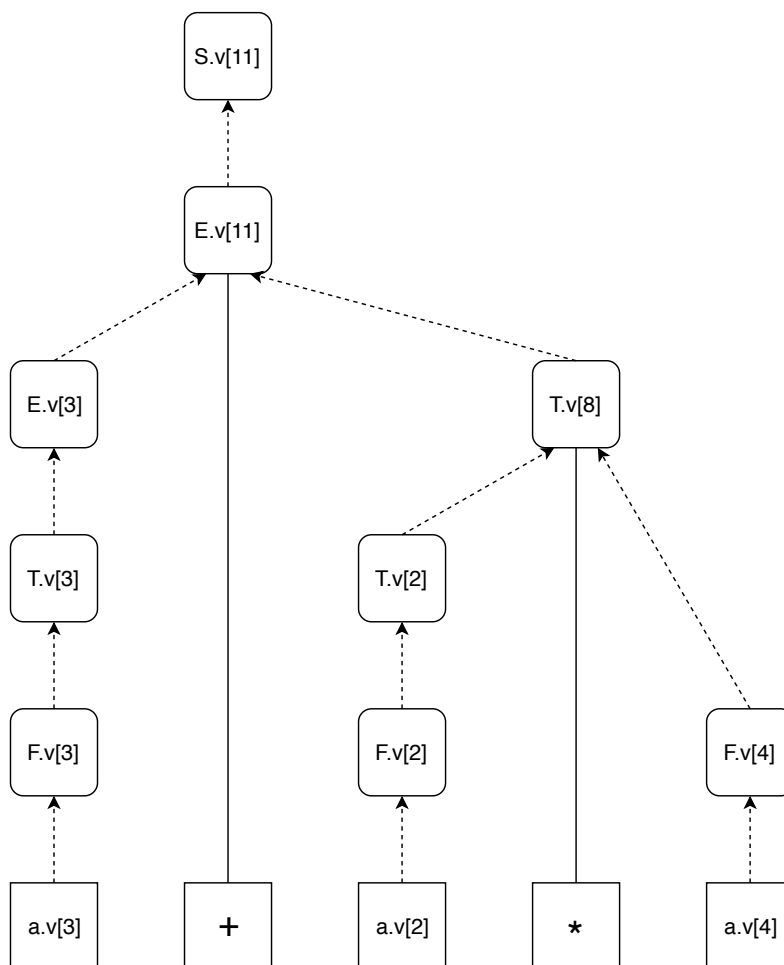
-
- [21] Irony — .NET Language Implementation Kit. [cit. 2018-04-11]. Dostupné z WWW: <<https://github.com/IronyProject/Irony>>
- [22] ISO/IEC JTC 1/SC 22: Information technology – Syntactic metalanguage – Extended BNF. Technická Zpráva ISO/IEC 14977:1996, The International Electrotechnical Commission, 3, rue de Varembe, Case postale 131, CH-1211 Genève 20, Switzerland, 1996.
- [23] Jager, G.; Rogers, J.: Formal language theory: refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences*, ročník 367, č. 1598, 2012: s. 1956–1970, ISSN 0962-8436, doi:10.1098/rstb.2012.0077. Dostupné z WWW: <<http://rstb.royalsocietypublishing.org/cgi/doi/10.1098/rstb.2012.0077>>
- [24] Kočíčka, M.: *Automata library – LR parsing construction*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2016.
- [25] Lai, A.: Declare-Before-Use Is Non-Context-Free. 2 2008, [cit. 2018-02-12]. Dostupné z WWW: <<https://www.vex.net/~trebla/weblog/declare-before-use.xhtml>>
- [26] lambda_interpreter library. [cit. 2018-04-11]. Dostupné z WWW: <<https://bitbucket.org/simonasya/lambda-interpreter>>
- [27] Laurent, N.; Mens, K.: Taming Context-Sensitive Languages with Principled Stateful Parsing. *CoRR*, ročník abs/1609.05365, 2016, <1609.05365>. Dostupné z WWW: <<http://arxiv.org/abs/1609.05365>>
- [28] Meduna, A.: *Formal Languages and Computation: Models and Their Applications*. Boston, MA, USA: Auerbach Publications, první vydání, 2014, ISBN 1466513454, 9781466513457.
- [29] Meunier, R.: Pattern Languages of Program Design. In *Pattern Languages of Program Design*, editace J. O. Coplien; D. C. Schmidt, kapitola The Pipes and Filters Architecture, New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995, ISBN 0-201-60734-4, s. 427–440. Dostupné z WWW: <<http://dl.acm.org/citation.cfm?id=218662.218694>>
- [30] MSDN – Intermediate representations supported by MSVC. [cit. 2018-04-08]. Dostupné z WWW: <<https://social.msdn.microsoft.com/Forums/vstudio/en-US/1bc5a9cb-0659-45c6-9390-f2c35f9c9af6/intermediate-representations-supported-by-msvc?forum=parallelcppnative>>

- [31] Pettorossi, A.; Proietti, M.: Regularity of non context-free languages over a singleton terminal alphabet. *CoRR*, ročník abs/1705.09695, 2017, <1705.09695>. Dostupné z WWW: <<http://arxiv.org/abs/1705.09695>>
- [32] PLY (Python Lex-Yacc). [cit. 2018-04-11]. Dostupné z WWW: <<http://www.dabeaz.com/ply/index.html>>
- [33] Pyrser knihovna. [cit. 2018-03-15]. Dostupné z WWW: <<http://pythonhosted.org/pyrser/>>
- [34] Python namedtuple. [cit. 2018-04-08]. Dostupné z WWW: <<https://docs.python.org/3/library/collections.html#collections.namedtuple>>
- [35] Semantic Versioning 2.0.0. [cit. 2018.04.08]. Dostupné z WWW: <<https://semver.org/>>
- [36] Šestáková, E.: *Automaty a gramatiky. Sběrka řešených příkladů*. ČVUT Praha, první vydání, 2017.
- [37] Sikkel, K.; Nijholt, A.: *Parsing of Context-Free Languages*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, ISBN 978-3-662-07675-0, s. 61–100, doi:10.1007/978-3-662-07675-0_2. Dostupné z WWW: <https://doi.org/10.1007/978-3-662-07675-0_2>
- [38] Stack Overflow Developer Survey Results 2018. 2018, [cit. 2018-04-07]. Dostupné z WWW: <<https://insights.stackoverflow.com/survey/2018/#most-popular-technologies>>
- [39] TIOBE index. [cit. 2018-04-07]. Dostupné z WWW: <<https://www.tiobe.com/tiobe-index/>>
- [40] TraviCI. [cit. 2018-04-08]. Dostupné z WWW: <<https://travisci.org/>>
- [41] Tsai, W. H.; Fu, K. S.: Attributed Grammar-A Tool for Combining Syntactic and Statistical Approaches to Pattern Recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, ročník 10, č. 12, Dec 1980: s. 873–885, ISSN 0018-9472, doi:10.1109/TSMC.1980.4308414.
- [42] Version Control Systems Popularity in 2016. [cit. 2018-04-09]. Dostupné z WWW: <<https://rhodecode.com/insights/version-control-systems-2016>>

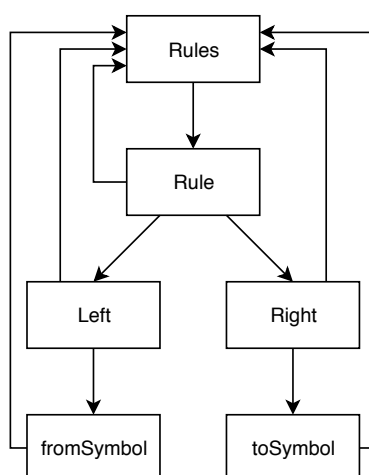
Obrazová příloha



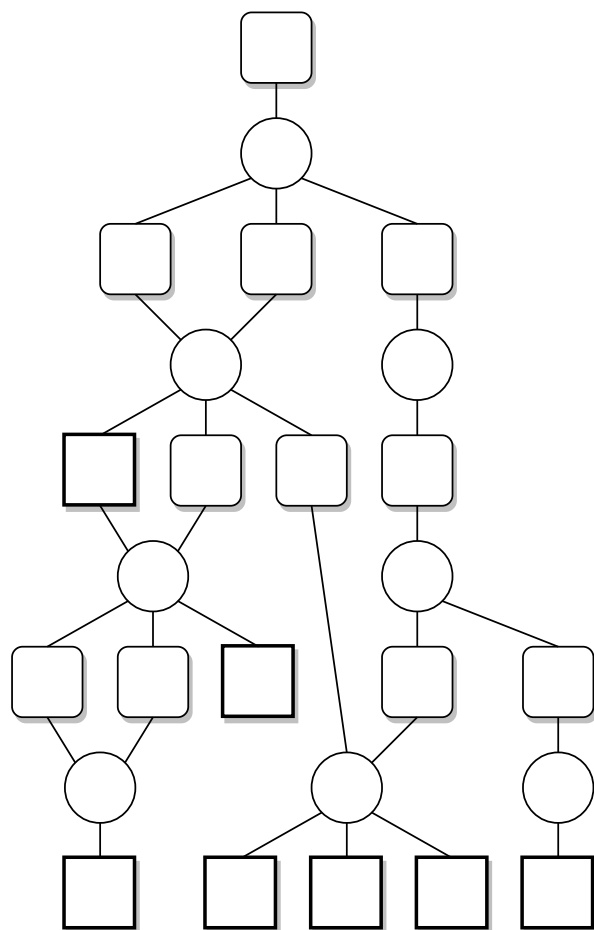
Obrázek A.1: Ukázka překládové gramatiky






Obrázek A.2: Vyhodnocení atributů u atributované gramatiky



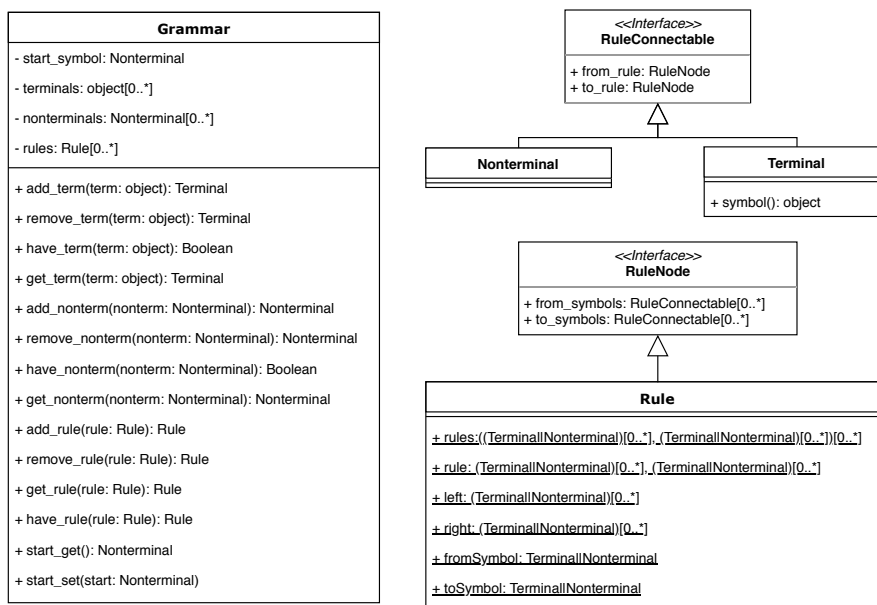
Obrázek A.3: Výpočet vlastností pro třídu Rule



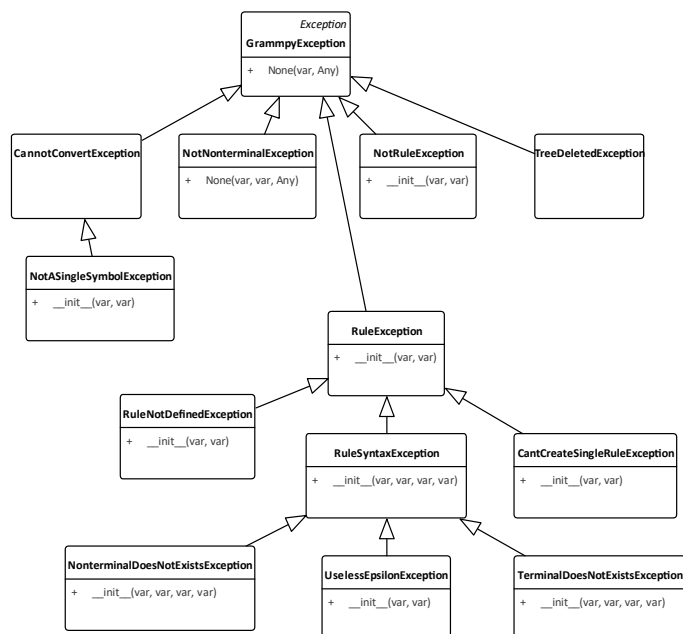
-  Instance terminálu
-  Instance neterminálu
-  Instance pravidla

Obrázek A.4: Repräsentace syntaktického stromu pro neomezené gramatiky

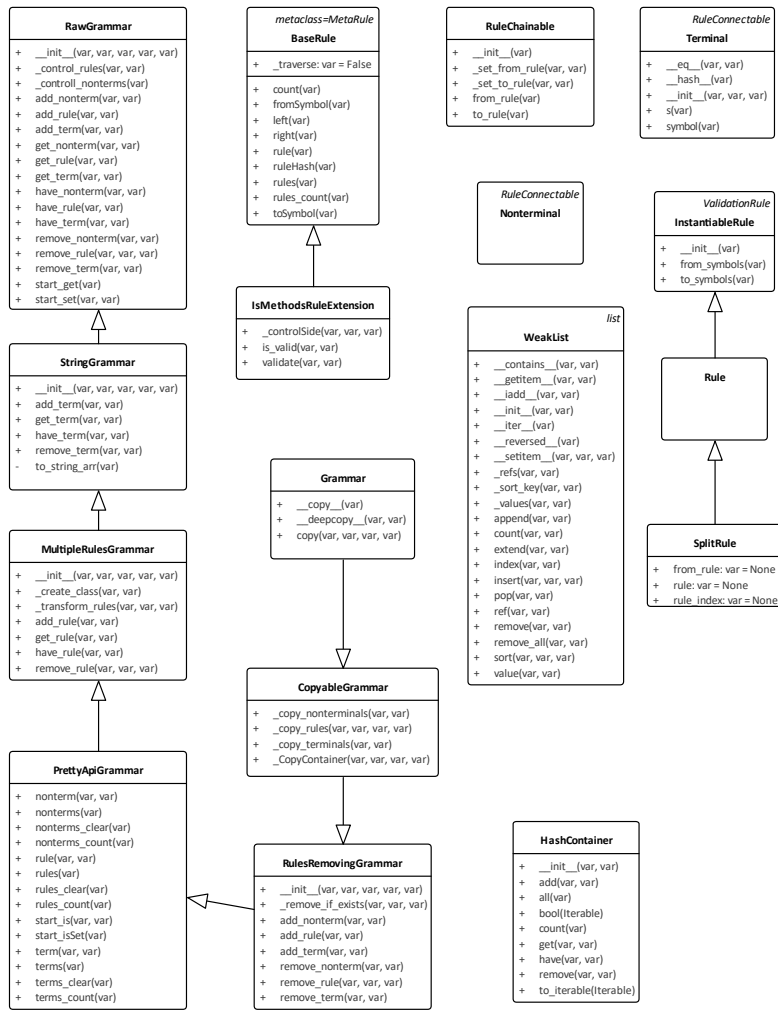
A. OBRAZOVÁ PŘÍLOHA



Obrázek A.5: Kompletní návrh reprezentující gramatiku

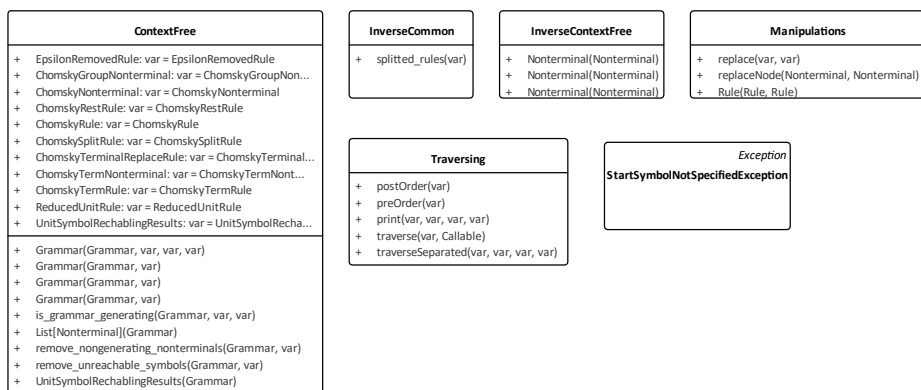


Obrázek A.6: Hierarchie vyjímek v modulu grammpy

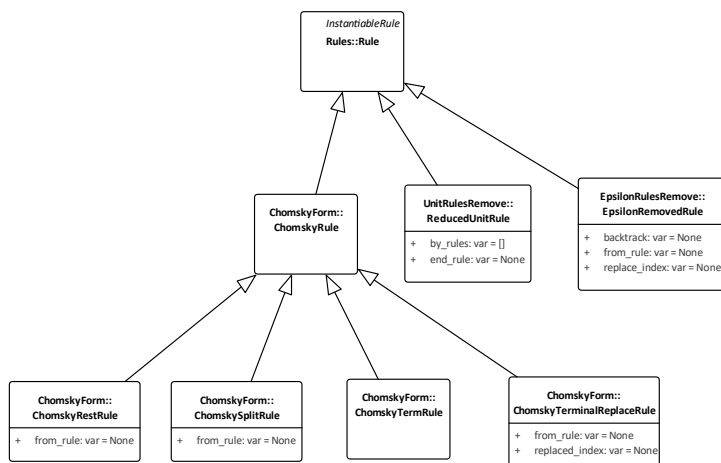


Obrázek A.7: Class diagram implementace modulu grammopy

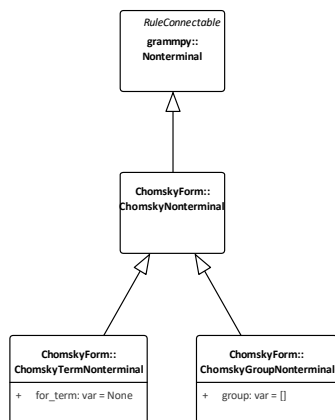
A. OBRAZOVÁ PŘÍLOHA



Obrázek A.8: Class diagram implementace modulu grammopy-transforms



Obrázek A.9: Class diagram pravidel v modulu grammopy-transforms



Obrázek A.10: Class diagram neterminálů v modulu grammpy-transforms

Seznam použitých zkratk

AST	Abstract syntax tree – abstraktní syntaktický strom.
CI	Continuous Integration – průběžná integrace.
CNF	Chomského normální forma.
CRUD	Create Read Update Delete – Vytváření, dotazování na stav, aktualizace a mazání.
CVUT	České vysoké učení technické v Praze.
CYK	Cocke-Younger-Kasami algoritmus.
EBNF	Extended Backus–Naur Form – Rozvinutá Backusova–Naurova forma.
FIT	Fakulta informačních technologií.
FURPS	Functionality, Usability, Reliability, Performance, Supportability – funkčnost, použitelnost, spolehlivost, výkon, udržovatelnost.
GC	Garbage Collector.
GCC	GNU Compiler Collection.
JSON	JavaScript Object Notation.
LALR	Look-Ahead LR parser.
LLVM	Low Level Virtual Machine.
MSVC	Microsoft Visual C++.
PEG	Parsing expression grammar.

SEZNAM POUŽITÝCH ZKRATEK

SLR	Simple LR parser.
XML	Extensible Markup Language.

Obsah přiloženého CD

readme.txt.....	stručný popis obsahu CD
src	
├─ grammpy.....	zdrojové kódy modulu grammpy
├─ grammpy-transforms.....	zdrojové kódy modulu grammp-transforms
├─ pyparsers	zdrojové kódy modulu pyparsers
examples	zdrojové kódy ukázek
├─ lambda-cli.....	lambda kalkulus interpret
├─ regex_inverse	parser regulárních výrazů
├─ calc.....	parser matematických výrazů
BP_Valkovic_Patrik_2018.pdf	text práce ve formátu PDF
thesis	zdrojové soubory práce ve formátu L ^A T _E X