UNIVERZITA KARLOVA V PRAZE

matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Matěj Outlý

Editor s funkcemi řízenými syntaxí jazyka

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Ondřej Zajíček

Studijní program: Informatika, oboru Programování

2008

Rád bych poděkoval především vedoucímu práce, Ondřeji Zajíčkovi, za svědomitou kontrolu, kritiku a směrování vznikající práce, včasné odpovědi, rady a dobré nápady, které pomohly k dosažení vytyčeného cíle. Dále si poděkování zaslouží Roman Barták a Jakub Yaghob, jejichž srozumitelné přednášky podaly dostatečný teoretický základ pro tuto práci.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne Matěj Outlý

Obsah

Název práce: Editor s funkcemi řízenými syntaxí jazyka
Autor: Matěj Outlý
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí bakalářské práce: Mgr. Ondřej Zajíček
E-mail vedoucího: Ondrej.Zajicek@mff.cuni.cz

Abstrakt: Návrh a implementace jádra programu, který využívá syntaxe obecného formálního jazyka, rozpoznává text a využívá získaných informací k další aplikaci, kterou může být například formátování nebo zvýrazňování zdrojového kódu. Cílem je popsat způsob počítačového zpracování jazyka daného gramatikou a navrhnout aplikaci smysluplně využívající těchto principů.

Klíčová slova: syntaxe, formální jazyk, gramatika, formátování, zvýrazňování zdrojového kódu

Title: Syntax directed editor
Author: Matěj Outlý
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: Mgr. Ondřej Zajíček
Supervisor's e-mail address: Ondrej.Zajicek@mff.cuni.cz

Abstract: Design and implementation of the program core, which uses syntax of the formal language, parses the input text and aplies the results in indentation and source code highlighting. The goal is to describe the way of the language processing given by formal grammar and to design the application, which builds on this principles.

Keywords: syntax, formal language, grammar, indentation, highlighting

Kapitola 1

Úvod

1.1 Základní definice

Definujme nejprve základní pojmy, které naleznou bohaté uplatnění v celém následujícím textu. Často budeme hovořit o různých abecedách, slovech, jazycích či dokonce třídách jazyků. Přirozeně asi každý tuší, že slovo bude nějakým způsobem složeno z písmen, tedy symbolů v abecedě a jazyk by pak mohl být nějaký soubor srozumitelných slov. Je však nezbytné tyto pojmy formálně zavést a náležitě vysvětlit.

Definice 1.1. *Abecedou* rozumíme libovolnou konečnou neprázdnou množinu symbolů.

Důležitá je zde podmínka konečnosti. Bez ní bychom jen těžko mohli konstruovat konečný automat či ho později reprezentovat v počítači.

Definice 1.2. Mějme abecedu X. *Slovem* w nazýváme nějakou konečnou posloupnost symbolů patřících do abecedy X.

$$w = x_1 x_2 \dots x_n,\ \forall i : x_i \in X$$

Prázdnou posloupnost speciálně označíme jako λ.
Délku slova budeme zapisovat do svislých čar, tedy $|w| = n$ ($|\lambda| = 0$).

Definice 1.3. Mějme slovo $w = x_1 \dots x_n,\ \forall i : x_i \in X$. *Prefixem* p slova w označíme konečnou posloupnost symbolů $p = x_1 \dots x_m$ kde $m \leq n$. Píšeme $p \trianglelefteq w$.

Úmluva. Symbolem X^* budeme dále označovat množinu všech slov na abecedě X. Konstrukce této množiny je zcela přirozená. Bude obsahovat všechny možné variace (s opakováním) symbolů z abecedy X. Je tedy zřejmé, že tato množina již není konečná. Můžeme ji chápat jako jakési universum, nad kterým budeme nadále pracovat. Podobně bude symbol X^+ značit množinu všech neprázdných slov na abecedě X (Zřejmě platí $X^+ = X^* - \{\lambda\}$).

Použití hvězdičky a znaku plus v označení těchto množim bude vysvětleno a formálně zavedeno později v části o operacích nad jazyky (2.3).

Nyní už máme potřebný aparát k zadefinování formálního jazyka tak, jak ho budeme v tomto textu potřebovat.

Definice 1.4. Mějme abecedu X nějakých symbolů. *Jazykem* L budeme rozumět libovolnou podmnožinu množiny X^*.

$$L \subseteq X^*$$

Jazyk tedy stejně jako zmíněné universum nemusí být konečná množina. Obecně tedy není možné definovat konkrétní jazyk jako výčet jeho slov. Naštěstí jsme schopni popsat určité (dalo by se říci i speciální) jazyky jinými způsoby. Právě způsob zápisu, jeho použití a zpětné rozhodnutí, zda dané slovo patří do definovaného jazyka je předmětem a jádrem této práce.

1.2 Chomského hierarchie

Pro přehlednější práci uspořádejme jazyky do určitých tříd, tedy do skupin, které vymezují určité podobné chování a vlastnosti jazyků. Nejzákladnějším takovým uspořádáním je tzv. Chomského hierarchie:

- Rekurzivně spočetné jazyky ($\mathcal{L}_0$)
- Kontextové jazyky ($\mathcal{L}_1$)
- Bezkontextové jazyky ($\mathcal{L}_2$)
- Regulární jazyky ($\mathcal{L}_3$)

Noam Chomský (*1928) takto uspořádal jazyky podle jejich vyjadřovací síly – každá následující třída jazyků je nadmnožinou třídy ležící v hierarchii pod ní. Tedy například všechny regulární jazyky jsou zároveň bezkontextové, ale zdaleka ne každý bezkontextový jazyk je také jazykem regulárním. Jazyky z nižších tříchse tedy dají modelovat způsobem, který funguje pro jazyky z vyšších tříd.

Pro nás budou důležité zejména jazyky z třídy regulárních a bezkontextových jazyků, těmi se budeme zabývat podrobněji v následujících kapitolách.

Chomského hierarchie často přesněji popisuje požadavky na gramatiku jazyků z jednotlivých tříd, či dokonce uspořádává samotné gramatiky. K tomuto rozšíření se vrátíme později, po zadefinování pojmu gramatika.

Kapitola 2

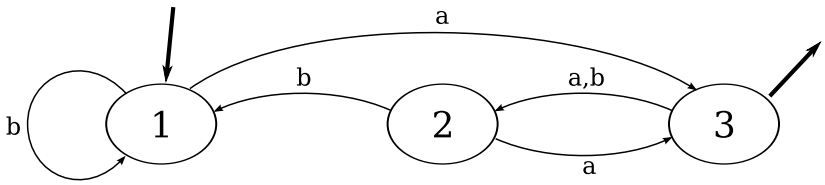
Regulární jazyky

2.1 Konečné automaty

Definice 2.1. *Konečným automatem* nazýváme pětici

$$A = (Q, X, \delta, q_0, F)$$

- Q je konečná neprázdná množina stavů, které budeme nadále říkat *stavový prostor*.
- X je vstupní abeceda.
- δ je zobrazení, které definuje přechod (přes symbol z abecedy) z jednoho stavu do jiného stavu, tedy $\delta : \quad Q \times X \rightarrow Q$. Toto zobrazení budeme nazývat *přechodovou funkcí*.
- q_0 je *počáteční stav*, $q_0 \in Q$.
- F je množina *koncových stavů*, $F \subseteq Q$.



Obrázek 2.1: Jednoduchý příklad konečného automatu vyjádřeného pomocí stavového diagramu

Tato definice je sice maximálně formální a přesná, nicméně moc dobře nepopisuje skutečnou funkci a význam konečného automatu. Jednen z nejběžnějších a také nejpřehlednějších způsobů jak popsat konečný automat je pomocí stavového diagramu (Obr. 2.1), kde uzly vyjadřují jednotlivé stavy, a šipky mezi nimi pak přechody

definované přechodovou funkcí. Počáteční stav je označen tlustou šipkou směřující do stavu, koncové stavy jsou pak označeny tlustou šipkou směřující od stavu. Všimněme si také faktu, že přechodová funkce je definována pro všechny stavy a všechny symboly (uvažujeme abecedu $X = \{a, b\}$), nemůže tedy nastat situace, kdy si automat „nebude vědět rady“, jakou cestou se má vydat.

Automat pak funguje následovně. Na počátku se nachází v počátečním stavu (v našem příkladu tedy ve stavu číslo 1). Automat postupně načítá symboly z imaginární vstupní pásky a podle načteného symbolu a aktuálního stavu použije přechodovou funkci ke změně stavu. Procedura se pak opakuje dokud je možné číst symboly ze vstupní pásky popř. dokud se automat nedostane do koncového stavu. Symboly načtené ze vstupní pásky po úspěšném přechodu do koncového stavu nám pak tvoří slovo, které je přijímáno konečným automatem. Všechna taková slova pak tvoří jazyk, přijímaný konečným automatem.

Zadefinujme toto však ještě jednou a pořádně.

Definice 2.2. Mějme přechdovou funkci $\delta : \ Q \times X \to Q$. *Rozšířenou přechodovou funkcí* pak rozumíme zobrazení $\delta^* : \ Q \times X^* \to Q$, definované následovně

$$\delta^*(q, \lambda) = q$$
$$\delta^*(q, wx) = \delta(\delta^*(q, w), x), \quad x \in X, \ w \in X^*$$

Jedná se tedy o tranzitivní uzávěr přechodové funkce.

Definice 2.3. *Jazyk rozpoznávaný konečným* automatem $A = (Q, X, \delta, q_0, F)$ je takový jazyk $L(A)$, pro který platí.

$$L(A) = \{w \mid w \in X^* \wedge \delta^*(q_0, w) \in F\}$$

Definice 2.4. Slovo w je *přijímáno* automatem A, právě když $w \in L(A)$.

Definice 2.5. Jazyk L je *rozpoznatelný konečným automatem*, právě když existuje konečný automat A takový, že $L = L(A)$. Třídu takových jazyků označíme symbolem $\mathcal{F}$.

2.2 Nedeterministické konečné automaty

Udělejme nyní malou odbočku, jejíž smysl vyjde najevo až v následující podkapitole. V některých situacích je jednodušší pracovat sice s konečným, avšak s automatem, ve kterém nejsou exaktně definovány přechody právě do jednoho stavu. Automat se pak v průběhu výpočtu nemůže deterministicky rozhodnout, jaký konkrétní přechod (z více možných) má zvolit a místo toho musí počítat se všemi variantami. Automat se tedy jakoby nachází ve více stavech najednou.

Zadefinujme v rychlosti také formálně.

Definice 2.6. *Nedeterministickým konečným automatem* nazýváme pětici

$$A = (Q, X, \delta, S, F)$$

- Q je konečná neprázdná množina stavů (*stavový prostor*).
- X je vstupní abeceda.
- δ je zobrazení, které definuje přechod (přes symbol z abecedy) z jednoho stavu do množiny stavů, tedy $\delta : \ Q \times X \to P(Q)$ (*přechodová funkce*).
- S je množina *počátečních stavů*, $S \subseteq Q$.
- F je množina *koncových stavů*, $F \subseteq Q$.

Definice 2.7. Slovo w je *přijímáno* nedeterministickým automatem A, jestliže existuje posloupnost stavů $q_1, \ldots, q_{n+1}$ taková, že

$$q_1 \in S$$
$$q_{i+1} \in \delta(q_i, x_i), \quad i = 1 \ldots n$$
$$q_{n+1} \in F$$

Pro nás bude podstatné následující tvrzení, ze kterého plyne, že nedeterministické konečné automaty zpracovávají stejnou třídu jazyků jako deterministické.

Věta 2.8. *Je-li A nedeterministický konečný automat, potom existuje konečný automat B takový, že $L(A) = L(B)$*

Důkaz. Provedeme konstruktivní důkaz spočívající ve vytvoření automatu B, jehož stavy budou nějaké podmnožiny stavového prostoru automatu A (Těch je konečně mnoho). Tím dokážeme nasimulovat přítomnost ve více stavech najednou.

$$B := (P(Q), X, \delta', S, F')$$

Koncové stavy budou takové podmnožiny, ve kterých je nějaký koncový stav z A

$$F' := \{R \mid R \in P(Q), \ R \cap F \neq \emptyset\}$$

a přechodová funkce bude sjednocení přechodových funkcí na A

$$\delta'(R, x) = \bigcup_{q \in R} \delta(q, x)$$

Vzhledem k tomu, že stavy automatu B jsou množiny stavů A, jedná se o přechodovou funkci pro deterministický automat. Stejně tak odpovídají všechny ostatní podmínky, B je tedy deterministický konečný automat.

Stačí dokázat, že skutečně přijímají stejný jazyk.

- $\lambda \in L(A) \Leftrightarrow S \cap F \neq \emptyset \Leftrightarrow S \in F' \Leftrightarrow \lambda \in L(B)$

  Počáteční stav B je také koncový, tedy S (počáteční stav B, ale také množina počátečních stavů A) je prvkem F' právě tehdy, když mají S a F neprázdný průnik (podle definice F'), tedy právě tehdy, když existuje stav automatu A který je počáteční a zároveň koncový.

- $L(A) \subseteq L(B)$

  Pro nějaké slovo z jazyka $L(A)$

$$w = x_1 \dots x_n \in L(A) \Leftrightarrow \exists q_1, \dots, q_{n+1} \in Q \quad (2.7)$$
$$q_1 \in S, q_{i+1} \in \delta(q_i, x_i), q_{n+1} \in F$$

  Položíme posloupnost stavů $R_1, \dots, R_{n+1}$ následovně

$$R_1 = S$$
$$R_{i+1} = \delta'(R_i, x_i)$$

  $(i+1)$-ní stav je tedy vždy sjednocení výsledků přechodové funkce ze všech stavů automatu A obsažených v i-tém stavu automatu B. Tedy $q_1 \in R_1$ a $q_{i+1} \in R_{i+1}$ (podle definice δ'). Potom ale také $R_{n+1} \in F'$, tedy slovo $w = x_1 \dots x_n$ je přijímáno také automatem B.

- $L(B) \subseteq L(A)$

  Postupujeme opačně něz v předchozím bodě. Pro nějaké slovo z jazyka $L(B)$

$$w = x_1 \dots x_n \in L(B) \Leftrightarrow \exists R_1, \dots, R_{n+1} \in P(Q) \quad (2.2, 2.3, 2.4)$$
$$R_1 = S,\ R_{i+1} = \delta'(R_i, x_i),\ R_{n+1} \in F'$$

  Zvolíme takové $q_{n+1} \in F \cap R_{n+1}$ (nějaké takové existuje podle definice F') a dále taková $q_i \in R_i$, pro která platí, že

$$q_{i+1} \in \delta(q_i, x_i)\ \ (\subseteq R_{i+1}), \quad \forall i = 1, \dots, n$$

  Potom i $q_1 \in R_1 = S$, tedy jsou splněny všechny podmínky pro přijetí nedeterministickým konečným automatem, proto $w \in L(A)$.

□

Nyní ještě rozšíříme konečné automaty o takzvané λ-přechody. Jedná se o zvlaštní druh přechodu, který říká, že automat může v tomto místě změnit stav (přejít po λ-přechodu) aniž by načetl nějaký symbol ze vstupní pásky.

Úmluva. V následujícím textu rozumíme symbolem ϵ speciální zástupný znak pro situaci, kdy není přečten žádný znak ze vstupní pásky.

Definice 2.9. *Konečným automatem s λ-přechody* rozumíme pětici

$$A = (Q, X \cup \{\epsilon\}, \delta, q_0, F)$$
$$\delta : Q \times (X \cup \{\epsilon\}) \to Q$$

pro kterou platí všechny další podmínky definované v 2.1. Přechodovou funkci definovanou pro libovolný stav $q \in Q$ a znak ϵ nazveme *λ-přechodem.*

Pro stavy, které nemají žádný λ-přechod lze definovat přechdovou funkci „do smyčky“, tedy $\delta(q, \epsilon) = q$.

Jak vidno, opět se jedná o formu nedeterminismu v konečných automatech.

Definice 2.10. Nechť A je konečný automat s λ-přechody, $q \in Q$ jeho libovolný stav a $R \subseteq Q$ libovolná množina stavů.

(i) Množinu $E(q) = \{p \mid p \in Q,\ p = \delta^*(q, \epsilon^*)\}$ kde ϵ^* je libovolně dlouhá posloupnost symbolů ϵ nazveme *λ-uzávěrem stavu q*.

(ii) Množinu $E(R) = \bigcup_{r \in R} E(r)$ nazveme *λ-uzávěrem množiny stavů R*.

Nyní už chybí jen jediný krůček. Předvedeme, že konečný automat s λ-přechody lze převést na nedeterministický konečný automat a tedy podle předchozí věty také na konečný automat.

Věta 2.11. *Je-li A konečný automat s λ-přechody, potom existuje nedeterministický konečný automat B takový, že $L(A) = L(B)$*

Důkaz.

$$A = (Q, X \cup \{\epsilon\}, \delta, q_0, F)$$

Pak konstruujeme automat B následovně

$$B = (Q, X, \delta', S, F)$$

kde

$$S := E(q_0)$$

$$\delta'(q, x) := E(\delta(q, x)), \quad q \in Q,\ x \in X$$

Takový automat jistě přijímá stejný jazyk jako původní automat A, protože každé přijímané slovo $w = x_1 \dots x_n \in L(B)$ je ekvivalentní libovolnému slovu $w' = \epsilon^* x_1 \epsilon^* \dots \epsilon^* x_n \epsilon^* \in L(A)$ a můžeme tedy najít příslušnou posloupnost stavů $q_1, \dots, q_{n+1}$, která je stejná v A i B. □

2.3 Množinové operace s jazyky

Jelikož máme jazyky zadefinovány jako nějaké množiny slov, můžeme s nimi samozřejmě provádět obvyklé množinové operace jako je sjednocení nebo průnik. Tyto operace můžeme jednoduše zadefinovat následovně.

Definice 2.12. Mějme libovolné jazyky L, L_1 a L_2. Pak jejich

(i) *sjednocením* rozumíme množinu $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$,

(ii) *průnikem* rozumíme množinu $L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$,

(iii) *rozdílem* budeme nazývat množinu $L_1 - L_2 = \{w \mid w \in L_1 \wedge w \notin L_2\}$

(iv) a konečně *doplňkem* jazyka L myslíme množinu $-L = \{w \mid w \in X^* \wedge w \notin L\}$.

Podívejme se, jak se tyto operace chovají na třídě jazyků rozpoznatelných konečným automatem.

Věta 2.13. *Nechť L, L_1 a L_2 jsou jazyky rozpoznatelné konečným automatem. Potom $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 - L_2$ a $-L$ jsou také jazyky rozpoznatelné konečným automatem (Třída $\mathcal{F}$ je uzavřená na operace sjednocení, průnik, rozdíl a doplněk).*

Důkaz. Důkaz provedeme konstruktivní.

(i) *sjednocení, průnik, rozdíl* jazyků L_1 a L_2

Nasimulujeme paralelní běh dvou automatů

$$A_1 = (Q_1, X, \delta_1, q_{10}, F_1),\ A_2 = (Q_2, X, \delta_2, q_{20}, F_2)$$

pro něž platí $L(A_1) = L_1$ a $L(A_2) = L_2$, tedy automatů, které přijímají dané jazyky.

Definujeme automat $A = (Q, X, \delta, q_0, F)$ následovně

$$Q := Q_1 \times Q_2,\ q_0 := (q_{10}, q_{20})$$
$$\delta((p_1, p_2), x) := (\delta_1(p_1, x), \delta_2(p_2, x)), \quad p_1 \in Q_1,\ p_2 \in Q_2,\ x \in X$$

a následně pro

- sjednocení: $F := (F_1 \times Q_2) \cup (F_2 \times Q_1)$
  pak automat A přijímá taková slova, která přijímá buď A_1 nebo A_2, tedy jazyk odpovídající sjednocení jazyků L_1 a L_2.
- průnik: $F := F_1 \times F_2$
  pak automat A přijímá taková slova, která přijímá A_1 a zároveň A_2, tedy jazyk odpovídající průniku jazyků L_1 a L_2.
- rozdíl: $F := F_1 \times (Q_2 - F_2)$
  pak automat A přijímá taková slova, která přijímá A_1, ale nepřijímá A_2, tedy jazyk odpovídající rozdílu jazyků L_1 a L_2.

(ii) *doplněk*

Zde je situace jednodušší, stačí pouze prohodit koncové a nekoncové stavy automatu A, $L(A) = L$.

□

Tento poznatek je jednoduchý, nicméně, stejně jako ten následující, poměrně zásadní při konstrukci regulárních výrazů. Nejprve ale zadefinujme nějaké další operace, které se dají na jazycích provádět.

Definice 2.14. Mějme libovolné jazyky L, L_1 a L_2.

(i) *Zřetězením* jazyků L_1 a L_2 rozumíme množinu $L_1.L_2 = \{uv \mid u \in L_1 \wedge v \in L_2\}$.

(ii) *Mocninou* jazyka L budeme nazývat množinu L^n iteračně definovou

$$L^0 = \{\lambda\}$$

$$L^{i+1} = L^i.L$$

(iii) *Pozitivní iterací* jazyka L myslíme množinu $L^+ = L^1 \cup L^2 ... = \bigcup_{i \geq 1} L^i$.

(iv) *Obecnou iterací* jazyka L rozumíme množinu $L^* = L^0 \cup L^1 \cup L^2 ... = \bigcup_{i \geq 0} L^i$.

Existuje samozřejmě celá řada dalších operací, které se navíc na zkoumané třídě jazyků $\mathcal{F}$ chovají rozumně (tím je myšlena zejména uzavřenost), my je však pro další povídání nebudeme potřebovat. I tak jsme si mohli odpustit například definici rozdílu nebo doplňku, kterou již dále nevyužijeme. Bližší informace naleznete v [1].

Stejně jako dříve dokažme uzavřenost těchto operací na třídě jazyků rozpoznatelných konečným automatem.

Věta 2.15. *Nechť L_1 a L_2 jsou jazyky rozpoznatelné konečným automatem. Potom i jazyk $L_1.L_2$ je rozpoznatelný konečným automatem.*

$$L_1, L_2 \in \mathcal{F} \Rightarrow L_1.L_2 \in \mathcal{F}$$

Důkaz. Důkaz provedeme opět konstruktivní. Tentokrát budeme simulovat situaci, kdy nejdříve počítá automat $A_1 = (Q_1, X, \delta_1, q_{10}, F_1)$ rozpoznávající jazyk L_1 a ve chvíli, kdy skončí, začne počítat automat $A_2 = (Q_2, X, \delta_2, q_{20}, F_2)$ rozpoznávající jazyk L_2.

Je jisté, že takovou operací zaneseme do výsledného automatu nedeterminismus, jelikož při přechodu prvního automatu do koncového stavu není jasné, zda má výpočet probíhat nadále v automatu A_1, nebo zda má přepnout do automatu A_2. Musí se tedy předpokládat obě varianty a tím se automat dostává do více stavů najednou. K realizaci takového automatu můžeme zcela přímo a jednoduše využít λ-přechody, pro které jsme si připravili teorii v předchozí podkapitole. Postačí, když λ-přechodem spojíme koncové stavy prvního automatu s počátečním stavem automatu druhého.

Výsledný automat tedy sestavíme následujícím způsobem.

$$B = (Q, X \cup \{\epsilon\}, \delta, q_{01}, F)$$

$$Q := Q_1 \cup Q_2$$

$$F := F_2$$

$$\delta(q, x) := \delta_1(q, x), \quad q \in Q_1$$

$$\delta(q, x) := \delta_2(q, x), \quad q \in Q_2$$

$$\delta(q, \epsilon) := q_{02}, \quad q \in F_1$$
$$\delta(q, \epsilon) := q, \quad q \notin F_1 \ (q \in Q_1 \vee q \in Q_2)$$

Stačí nahlédnout, že takový automat opravdu rozpoznává jazyk odpovídající zřetězení L_1 a L_2. Automat B navíc odpovídá definici konečného automatu s λ-přechody, a tedy podle 2.11 a 2.8 lze převést na konečný automat. □

Věta 2.16. *Nechť L je jazyk rozpoznatelný konečným automatem. Potom i jazyk L^* je rozpoznatelný konečným automatem.*

$$L \in \mathcal{F} \Rightarrow L^* \in \mathcal{F}$$

Důkaz. Mějme automat $A = (Q, X, \delta, q_0, F)$ rozpoznávající jazyk L. Podobně jako v předchozím tvrzení budeme konstruovat výsledný „iterační“ automat za pomocí λ-přechodů. Tentokrát však budeme spojovat koncové stavy s počátečním stavem toho samého automatu. Je nutné si však uvědomit jednu maličkost a tou je prázdné slovo λ. To je prvkem jazyka L^* i v případě, že ho původní jazyk L neobsahoval. Je tedy nutné λ ošetřit speciálně novým stavem s. Výsledný automat bude tedy definován následujícím způsobem.

$$B = (Q', X, \delta', q_0, F')$$

$$Q' := Q \cup \{s\}$$
$$F' := F \cup \{s\}$$
$$\delta'(q, x) := \delta(q, x), \quad q \in Q$$
$$\delta(q, \epsilon) := q_0, \quad q \in F_1, \ q \neq q_0$$
$$\delta(q, \epsilon) := q, \quad q \notin F_1, \ q \neq q_0$$
$$\delta(q_0, \epsilon) := s$$

A stejně jako v předchozím tvrzení převedeme automat B podle 2.11 a 2.8 na konečný automat. □

Úmluva. Jazyk, který neobsahuje žádná slova, tedy *prázdný jazyk*, označme symbolem $\emptyset$.

Nyní již konečně můžeme přistoupit k bodu, ke kterému jsme celou kapitolu směle směřovali. Tím je zadefinování regulárních jazyků jako takových. Dosud jsme se bavili jen o jakési třídě $\mathcal{F}$, nebylo však zcela jasné, jakou má vlastně spojitost s regulárními jazyky. Toto poznání nás čeká hned v následujícím podkapitole, která na naše povídání plynule navazuje.

Definice 2.17. *Třída regulárních jazyků $RL(X)$* nad abecedou X je nejmenší třída jazyků, pro kterou platí:

- Obsahuje prázdný jazyk $\emptyset$.

- Pro každý symbol x z abecedy X obsahuje jazyk $\{x\}$.
- Je uzavřená na operaci sjednocení, tedy $A, B \in RL(X) \Rightarrow A \cup B \in RL(X)$.
- Je uzavřená na operaci zřetězení, tedy $A, B \in RL(X) \Rightarrow A.B \in RL(X)$.
- Je uzavřená na operaci obecné iterace, tedy $A \in RL(X) \Rightarrow A^* \in RL(X)$.

Jedná se tedy vlastně o algebru se zmíněnými třemi operacemi.

2.4 Kleeneova věta

Věta 2.18. *Libovolný jazyk je regulární, právě když je rozpoznatelný konečným automatem.*

Důkaz.
„$\Longrightarrow$“

Dokazujeme, že regulární jazyky jsou rozpoznatelné konečným automatem.

Stačí si uvědomit, že pro triviální jazyky (tedy jazyky tvaru $\{x\}$ pro každý symbol x z abecedy X) jsme schopni zkonstruovat jednoduchý konečný automat (přesný tvar tohoto automatu bude uveden později v kapitole o implementaci regulárních výrazů).

Z poznatků 2.13, 2.15, 2.16 jsme schopni pro všechny operace na $RL(X)$ (tedy sjednocení, zřetězení a obecná iterace) nalézt odpovídající postup konstrukce konečného automatu. Iterací těchto postupů jsme tedy schopni nalézt konečný automat pro všechny regulární jazyky.

„$\Longleftarrow$“

Dokazujeme, že jazyky rozpoznatelné konečnými automaty jsou regulární, tedy podle definice 2.17 se dají složit z elementárních jazyků aplikováním operací sjednocení, zřetězení a iterace.

Mějme konečný automat $A = (Q, X, \delta, q_1, F)$ a uvažujme na množině Q uspořádání, které nám očísluje jednotlivé stavy indexy $1, \ldots, n,\ |Q| = n$. Položme

$$R_{ij} := \{w \in X^* \mid \delta^*(q_i, w) = q_j\}$$

tedy množinu slov, které převádějí stav q_i na stav q_j. Jistě platí

$$L(A) = \bigcup_{q_i \in F} R_{1i}$$

Položme dále R_{ij}^k jako množinu slov, které převádějí stav q_i na stav q_j bez toho, aby rozšířená přechodová funkce prošla nějakým stavem q_m, kde $m > k$ (procházíme pouze stavy s menším nebo rovným indexem). Jistě tedy platí

$$R_{ij} = R_{ij}^n$$

Nyní dokažme, že R_{ij}^k jsou regulární pro $\forall i, j, k$. Budeme postupovat indukcí.

- $\forall i, j : R_{ij}^0$ jsou jistě regulární, neboť připadají v úvahu maximálně jednopísmenná slova (neprocházíme přes žádný mezistav).
- $R_{ij}^{k+1} = R_{ij}^k \cup R_{i(k+1)}^k . (R_{(k+1)(k+1)}^k)^* . R_{(k+1)j}^k$

  Přidávaná slova procházející přes stav q_{k+1} zkonstruujeme tak, že zřetězíme slova jdoucí z q_i do q_{k+1} se slovy jdoucí z q_{k+1} do q_j a mezi ně zřetězíme všechny možné smyčky z q_{k+1} zpět do q_{k+1}. Všechny tyto části neprochází stavy s větším indexem než k.

  Z indukčního předpokladu víme, že jazyky R_{ij}^k, $R_{i(k+1)}^k$, $R_{(k+1)(k+1)}^k$ a $R_{(k+1)j}^k$ jsou regulární a operace sjednocení, zřetězení a iterace zachovávají regulárnost. R_{ij}^{k+1} je tedy také regulární.

Tedy $\forall i, j$ je R_{ij} regulární a vzhledem k tomu, že sjednocení zachovává regulárnost, je i $L(A)$ regulární. □

2.5 Regulární výrazy

Nyní již tedy máme určitou představu o tom, jak vypadají regulární jazyky. Dokonce bychom na základě předešlé teorie byli do uřité míry schopni navrhnout a implementovat aplikaci, schopnou reprezentovat nějaký regulární jazyk (tedy rozhodnout, zda dané slovo patří do implementovaného jazyka). Čeká na nás však ještě jeden podstatný úkol.

Tím úkolem je navrhnout nějaký jednoduchý způsob definování konkrétního regulárního jazyka. Tím způsobem by samozřejmě mohlo být definování přímo pomocí odpovídajícího konečného automatu, dalším způsobem by mohlo být například popsání přirozeným jazykem (např. „jazyk nad abecedou $\{0, 1\}$, který obsahuje všechna slova, která odpovídají binárnímu zápisu čísla dělitelného pěti“). Oba způsoby jsou však z nějakého důvodu nevyhovující.

Reprezentace konečným automatem sice přesně definuje daný jazyk, na první pohled však nevypovídá příliš dobře o tom, jak vlastně daná slova v jazyce vypadají. V praxi se pak používají jazyky příliš složité na to, aby mohl být tento způsob zápisu pro člověka srozumitelný.

Druhý navrhovaný způsob dává celkem jasnou představu o významu daného jazyka, ovšem jako formální definice pro počítač je pochopitelně zcela nevyhovující. Navíc nikde není jasně dáno, že jazyk, který jsme právě popsali, je obravdu regulární.

Z těchto důvodů definujeme třetí způsob, který bude vhodný jak pro počítač, tak poměrně srozumitelný pro člověka.

Definice 2.19. *Množina regulárních výrazů* $RE(X)$ nad abecedou $X = \{x_1, ..., x_n\}$ je nejmenší množina slov v abecedě $\{x_1, \ldots, x_n, \emptyset, \lambda, |, *, (,)\}$, pro kterou platí:

- Obsahuje výraz $\emptyset$ a výraz λ, $\emptyset \in RE(X)$, $\lambda \in RE(X)$.
- Pro každý symbol x v abecedě X obsahuje výraz x, $\forall x \in X : x \in RE(X)$.
- $\forall \alpha, \beta \in RE(X) \Rightarrow (\alpha|\beta) \in RE(X)$.

- $\forall \alpha, \beta \in RE(X) \Rightarrow (\alpha\beta) \in RE(X)$.
- $\forall \alpha \in RE(X) \Rightarrow \alpha^* \in RE(X)$.

Trochu v předstihu bychom mohli podotknout, že samotný jazyk regularních výrazů je sice bezkontextový, avšak my se nyní budeme zajímat špíše o hodnotu regulárního výrazu, kterou bude právě nějaký regulární jazyk. Ale neprotahujme a uveďme rovnou formální definici.

Definice 2.20. *Hodnotou regulárního výrazu* $\alpha \in RE(X)$ je množina slov $[\alpha]$, tedy jazyk, definovaný následovně. Pro $\forall \alpha, \beta \in RE(X)$ na nějaké abecedě X platí

- $[\emptyset] = \emptyset, [\lambda] = \{\lambda\}, [x] = \{x\}, \forall x \in X$,
- $[(\alpha|\beta)] = [\alpha] \cup [\beta]$,
- $[(\alpha\beta)] = [\alpha].[\beta]$,
- $[\alpha^*] = [\alpha]^*$.

Navzdory definici by se ještě slušelo ujasnit, jakou funkci mají v regulárních výrazech symboly $\emptyset$ a λ. $\emptyset$ vyjadřuje prázdný regulární výraz. Podle definice tento regulární výraz generuje prázdný jazyk, tedy jazyk, ve kterém není žádné slovo (ani prázdné). Oproti tomu λ (ve smyslu regulárních výrazů) je regulární výraz odpovídající právě jednomu slovu v universu X^* a to prázdnému slovu λ. Tento regulární výraz tedy generuje jazyk, který obsahuje jen prázdné slovo, tedy jazyk X^0.

Z definic 2.17 a 2.20 jsou zcela zřejmá následující fakta.

Věta 2.21. *Hodnotou regulárního výrazu je regulární jazyk.*

Věta 2.22. *Každý regulární jazyk lze reprezentovat pomocí regulárního výrazu (jehož hodnou je onen jazyk).*

V regulárních výrazech tedy můžeme vidět jakýsi mezikrok mezi přirozeným vyjádřením regulárního jazyka a zapsáním regulárního jazyka pomocí konečného automatu.

přirozený jazyk → regulární výraz → konečný automat

Převod mezi prvními dvěma pochopitelně nelze algoritmicky popsat, to je zcela na lidské invenci, avšak převod mezi regulárním výrazem a konečným automatem se dá popsat dokonce nekolika různými algoritmy. Jedna zcela zřejmá metoda vychází z definic a vět, popsaných výše. Nazveme ji *metodou inkrementální* a sestává ze dvou kroků.

(i) Všechny elementární (jednopísmenné) regulární výrazy převedeme na konečný automat.

(ii) Složitější automaty skládáme dohromady způsobem definovaným v 2.20.

Tato metoda bude blíže popsána v kapitole 4.3 o implementaci regulárních výrazů. Další metody nebudeme dále potřebovat, v případě zájmu je naleznete v [1].

Kapitola 3

Bezkontextové jazyky

3.1 Formální gramatiky

Každý se již někdy setkal s pojmem gramatika a pravděpodobně i tuší jeho přibližný význam. Gramatika by se dala tradičně (tedy v lingvistickém slova smyslu) popsat jako soubor pravidel, podle kterých se řídí syntaxe a morfologie jazyka, tedy skladba vět a tvoření jednotlivých slov. Tato, pro matematiky trochu vágní, definice se samozřejmě dá primárně aplikovat na přirozené jazyky.

My ji však pro naší potřebu trochu zjednodušíme a hlavně zformalizujeme. Zejména odstraníme rozdíl mezi tvorbou jednotlivých slov a celých vět.

V první řadě nemáme pojem věta vůbec zaveden. Časem dopějeme k závěru, že tvorba slov a tvorba vět je z našeho hlediska dost podobná a tedy námi použitá analýza jazyka bude probíhat dvoufázově, kde každá jednotlivá fáze leží na stejném teoretickém základě. Omezme se tedy na dříve definovanou abecedu, symboly z abecedy a slova z nich tvořená. Gramatika v našem slova smyslu bude tedy soubor pravidel, podle kterých se vytvářejí všechny řetězce (slova) daného jazyka.

Definice 3.1. Mějme abecedu V. *Přepisovací pravidlo (produkce)* (nad abecedou V) je uspořádaná dvojice (u, v), kde $u, v \in V^*$. Zpravidla zapisujeme ve tvaru $u \to v$

Definice 3.2. *Přepisovacím (produkčním) systémem* nazýváme dvojici $R = (V, P)$, kde

- V je abeceda, nad kterou pracujeme.
- P je konečná množina přepisovacích pravidel nad abecedou V.

Definice 3.3. Mějme produkční systém $R = (V, P)$.

(i) Říkáme, že slovo w ($\in V^*$) se *přímo přepíše* na slovo z ($\in V^*$) (píšeme $w \Rightarrow z$), jestliže

$$\exists u, v, x, y \in V^* : w = xuy,\ z = xvy,\ (u \to v) \in P$$

(ii) Říkáme, že slovo w ($\in V^*$) se *přepíše* na slovo z ($\in V^*$) (píšeme $w \Rightarrow^* z$), jestliže

$$\exists u_1, \ldots, u_n \in V^* : w = u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_n = z$$

(iii) Posloupnost $u_1, \ldots, u_n$ nazýváme *derivací*.

(iv) Jestliže $\forall i \neq j : u_i \neq u_j$, potom hovoříme o *minimální derivaci*.

Nyní tedy umíme nějakým způsobem vyprodukovat slovo složené ze symbolů v abecedě V. To však stále nestačí. Máme sice dobrý aparát na tvorbu slov, ale například nevíme s čím začít a kde skončit. Rozšířením definice se konečně dobereme k dostačujícímu zavedení formální gramatiky.

Definice 3.4. *Formální gramatikou* myslíme čtveřici $G = (V_N, V_T, S, P)$, kde

- V_N je abeceda *neterminálních symbolů*.
- V_T je abeceda *terminálních symbolů*. Obě abecedy jsou neprázdné a disjunktní.
- $S \in V_N$ je *počáteční neterminální symbol*.
- P je systém produkcí $\alpha \rightarrow \beta$, kde $\alpha, \beta \in (V_N \cup V_T)^*$ a α obsahuje alespoň jeden neterminální symbol.

Definice 3.5. Jazyk $L(G)$ *generovaný gramatikou* G je takový jazyk, pro který platí

$$L(G) = \{w \mid w \in V_T^* \ \wedge \ S \Rightarrow^* w\}$$

Přepisování slov funguje stejně jaku u produkčního systému. Novinkou je počáteční neterminál, tedy symbol, od kterého s produkcí začínáme. Stojí tedy na začátku každé derivace. Terminální symboly jsou pak ty symboly, u kterých derivaci končíme, tedy symboly, které se již dále nepřepisují. Celý postup končí tím, že vyprodukujeme slovo složené jen z terminálních symbolů. To nám tedy i jasně ukazuje, že jazyk $L(G)$ je podmnožinou V_T^*.

3.2 Chomského hierarchie podruhé

Jak jsme již dříve předeslali, doplníme teď Chomského hierarchii za pomocí gramatik. Víme, že každé gramatice odpovídá nějaký jazyk. Stejně tak můžeme již zmíněné třídy jazyků definovat právě podle určitých požadavků, které klademe na jejich gramatiky.

- Gramatiky typu 0

  Mají přepisovací pravidla v obecné formě, tedy libovolná, odpovídající 3.4. Tyto gramatiky generují třídu rekurzivně spočetných jazyků $\mathcal{L}_0$.

- Gramatiky typu 1

  Jejich přepisovací pravidla jsou pouze ve tvaru $\alpha X\beta \to \alpha w\beta$, kde $X \in V_N$, $\alpha, \beta \in (V_N \cup V_T)^*$, $w \in (V_N \cup V_T)^+$. Takové gramatiky jsou tedy nevypouštějící (slova se během přepisování nezkracují). Jedinou výjimkou budiž pravidlo $S \to \lambda$ s tím, že se počáteční neterminál S nesmí objevit na pravé straně žádného pravidla. Touto výjimkou zajistíme, aby takové gramatiky byly schopny generovat jazyky, obsahující prázdné slovo. Gramatiky typu 1 generují právě třídu kontextových jazyků $\mathcal{L}_1$.

- Gramatiky typu 2

  Právě tyto gramatiky (resp. určitá jejich podmnožina) budou pro nás nejzajímavější. Gramatiky typu 2 mají přepisovací pravidla ve tvaru $X \to \alpha$, kde $X \in V_N$, $\alpha \in (V_N \cup V_T)^*$. Podrobnějšího popisu se dočkáme za chvíli.

- Gramatiky typu 3

  Tyto gramatiky mají pravidla pouze ve tvaru $X \to wY$ nebo $X \to w$, kde $X, Y \in V_N$, $w \in V_T^*$. Tyto gramatiky generují třídu jazyků $\mathcal{L}_3$. Ta je rovna třídě regulárních jazyků, kterou jsme definovali dříve. Formální důkaz tohoto tvrzení naleznete v [1], v kapitole 6.

Chomského hierarchie nám utvořila určitou bližší představu o uspořádání jazyků a jejich gramatik. Tato představa je však stále jen přibližná a nebude ji považovat za formální definici. Omezíme se pouze na třídu bezkontextových jazyků a spokojíme se s následujícím.

Definice 3.6. *Bezkontextová gramatika* (zkratka CFG) je taková formální gramatika, která obsahuje přepisovací pravidla pouze ve tvaru $X \to \alpha$, kde $X \in V_N$, $\alpha \in (V_N \cup V_T)^*$.

Definice 3.7. *Bezkontextovým jazykem* L budeme nazývat takový jazyk, pro který existuje nějaká bezkontextová gramatika G taková, že $L = L(G)$

3.3 Derivační stromy a jednoznačnost gramatik

Vzhledem k „pěknému“ tvaru pravidel, který nám definuje 3.6, můžeme výpočet (přepsání) bezkontextových gramatik zachytit o mnoho názornější strukturou, než jen sekvencí pravidel. Tímto způsobem zápisu je derivační strom.

Jedná se víceméně o pouhé přeuspořádání použitých přepisovacích pravidel do grafické podoby stromu s tím, že na sebe navážeme stejné neterminály, které se v jednom kroku vygenerovaly na pravé straně pravidla a v následujících krocích jsou dále přepisovány. Uveďme však přesnou definici následovanou názornějším příkladem.

Definice 3.8. Mějme bezkontextovou gramatiku $G = (V_N, V_T, S, P)$. *Derivační strom* je takový strom, kde:

- každý vrchol je ohodnocen prvkem z $V_N \cup V_T \cup \{\lambda\}$.
- kořen je ohodnocen počátečním neterminálem S.
- vnitřní vrcholy jsou ohodnoceny prvky z V_N.
- je-li A ohodnocením vrcholu a $u_1, u_2, \ldots, u_n$ ohodnocením jeho potomků (záleží na pořadí, bereme potomky zleva doprava), potom $(A \to u_1 u_2 \ldots u_n) \in P$.
- je-li vrchol ohodnocen λ, potom je to list a je jediným potomkem svého rodiče.

Definice 3.9. Říkáme, že derivační strom *dává slovo* w, pokud je slovo složeno z ohodnocení listů (zleva doprava).

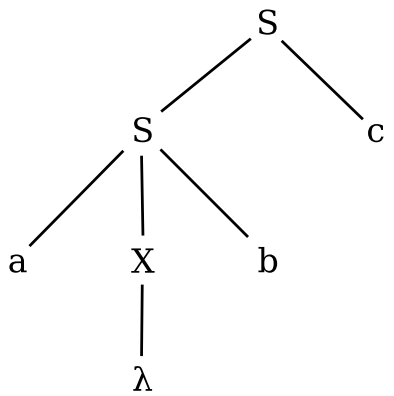
Příklad. Mějme gramatiku $G = (V_N, V_T, S, P)$

$$V_N = \{S, X\}$$
$$V_T = \{a, b, c\}$$
$$P : S \to aXb \mid Sc, \quad X \to c \mid \lambda$$

Jedna z možných derivací může být

$$S \Rightarrow Sc \Rightarrow aXbc \Rightarrow abc$$

K této derivaci pak můžeme sestrojit derivační strom, který bude mít následující podobu:



Obrázek 3.1: Jednoduchý příklad derivačního stromu

Tento strom podle definice 3.9 dává slovo *abc* a přehledně vypovídá o tom, jaká byla použita přepisovací pravidla.

Vyslovme teď jednu důležitou otázku týkající se právě derivací a derivačních stromů. Je možné získat pro jedno slovo dva různé derivační stromy? Tato, na pohled jednoduchá otázka, nám však svou odpovědí přináší poněkud znepokojující situaci.

Pravdou je, že v některých případech je to skutečně možné. Ujasněme si pojmy následujícími definicemi.

Zavedeme nejprve jakýsi jednotný způsob přepisování. Pokud máme přepisovací pravidlo, které má na pravé straně dva nebo více neterminálů, je pochopitelně možné přepsat například nejprve levý a pak pravý (a dále) nebo obráceně, pokud jich je tam větší počet, můžeme také postupovat v náhodném pořadí.

Dostáváme tedy několik různých derivací (myšleno různě uspořádaných posloupností z definice 3.3). Význam takových derivací je však stále stejný, odpovídající derivační strom má stále stejnou podobu. Pořadí přepisovaných neterminálů tedy nemá z hlediska jednoznačnosti význam, nicméně níže specifikovaná kanonická forma derivace nám umožní jednoznačnost zadefinovat.

Definice 3.10. Mějme gramatiku $G = (V_N, V_T, S, P)$ a slova $\alpha, \beta \in (V_N \cup V_T)^*$. O *levém přepsání* $\alpha \Rightarrow \beta$ mluvíme tehdy, přepisuje-li se nejlevější neterminál

$$\exists \gamma, \delta \in (V_N \cup V_T)^*,\ \exists w \in V_T^*,\ \exists X \in V_N : \alpha = wX\delta,\ \beta = w\gamma\delta,\ (X \to \gamma) \in P$$

Levá derivace vzniká použitím pouze levých přepsání.

Obdobně můžeme definovat i pravé přesání a pravou derivaci. Rozdíl je pouze v tom, že se vždy přepisuje nejpravější neterminál. Uveďme ještě větu, která nám říká, že se skutečně jedná o korektně definovanou derivaci.

Věta 3.11. *Pro bezkontextové gramatiky platí, $X \Rightarrow^* w$ právě tehdy, když existuje levá derivace w z X.*

Důkaz. Existuje-li daná levá derivace, zcela zřejmě se podle ní X přepíše na slovo w.

Stačí tedy dokázat, že existence derivace implikuje existenci levé derivace. Mějme nějaké přepsání $\alpha Y \beta \Rightarrow \alpha\gamma\beta$ použitím pravidla $Y \to \gamma$. Takové pravidlo neovlivní řetězce α a β ani jejich další přepisování, jednotlivá přepisování jsou na sobě nezávislá. Můžeme tedy preferovat aplikaci některých pravidel, tedy i levých přepsání. □

Nyní můžeme konečně zadefinovat jednoznačnost gramatiky a jazyka.

Definice 3.12. Bezkontextová gramatika G je *víceznačná*, jestiže $\exists w \in L(G)$, které má dvě různé levé derivace, tedy dva různé derivační stromy. V ostatních případech je gramatika *jednoznačná*.

Definice 3.13. Bezkontextový jazyk L je *jednoznačný*, jestiže existuje jednoznačná bezkontextová gramatika G tak, že $L = L(G)$.

Bezkontextový jazyk L je *nejednoznačný*, jestiže každá bezkontextová gramatika G taková, že $L = L(G)$, je nejednoznačná.

Pro účely zpracování jazyka počítačem je nejednoznačnost nepříjemný problém. Pokud po derivačním stromě požadujeme, aby něco skutečně reprezentoval, tedy jeho forma nesla určitý význam a ten význam se změnil s rozdílnou reprezentací, je jednoznačnost nepřípustná. Stroj by v takovém případě nemohl vytušit, jaká reprezentace je ta správná, a slovo by mělo více formálních významů.

Z toho důvodu je nutné položit první podmínku, kterou klademe na zpracovávaný jazyk (gramatiku). Zajištění jednoznačnosti tedy přechází na samotného uživatele.

3.4 Levá faktorizace

Někdy se v gramatice vyskytnou zdánlivě nejednoznačná, resp. pro stroj na první pohled nerozlišitelná, pravidla (z důvodu takového, že později popsaný syntaktický analyzátor většinou zkoumá pouze začátek pravidla, který je nějakým způsobem důležitý). Jsou jimi taková pravidla, která mají za prvé stejnou levou stranu, ale co je podstatnější, mají stejný začátek pravé strany (ať už jde o terminály nebo neterminály).

Existuje však algoritmus, kterým je možno takové případy eliminovat a gramatiku tak přepsat. Tento postup se nazývá *levá faktorizace*. Popišme tento postup jednoduchým tvrzením.

Věta 3.14. *Nechť $G = (V_N, V_T, S, P)$ je bezkontextová gramatika a*

$$X \to \alpha\beta_1$$
$$X \to \alpha\beta_2$$

nějaká její přepisovací pravidla, pro která platí $X \in V_N,\ \alpha, \beta_1, \beta_2 \in (V_N \cup V_T)^*$. Pak gramatika $G' = (V_N \cup \{X'\}, V_T, S, P')$ kde P' vznikne z P tak, že odebereme výše uvedená pravidla a místo nich přidáme pravidla

$$X \to \alpha X'$$
$$X' \to \beta_1$$
$$X' \to \beta_2$$

kde X' je nový neterminál, generuje stejný jazyk jako G, tedy $L(G) = L(G')$.

Důkaz. Jen krátce. Vezmeme slovo z jazyka $L(G)$ a najdeme jeho například levou derivaci. Pokud se v této derivaci využije přepsání podle pravidla $X \to \alpha\beta_1$ (resp. $X \to \alpha\beta_2$), pak jistě můžeme toto přepsání nahradit dvěma přepsáními podle pravidel $X \to \alpha X'$ a $X' \to \beta_1$ (resp. $X' \to \beta_2$). Pro zkoumané slovo tedy existuje levá derivace v gramatice G' a je tedy prvkem jazyka $L(G')$.

Opačnou impikaci postavíme symetricky. □

Jak vidno transformovaná gramatika již netrpí tím neduhem, že by měla nejednoznačné začátky některých pravidel. Této metody využijeme později v rámci implementace bezkontextového parseru.

3.5 Eliminace levé rekurze

Dalším podstatným problémem při zpracovávání gramatiky strojem je tzv. rekurze, ať už levá nebo pravá. První je problémem pro analyzátory stavějící výpočet odleva, druhá naopak pro analyzátory fungující zprava – o tom bude řeč v kapitole o syntaktické analýze. Pracujme nadále pouze s levou rekurzí. Nejdříve by se však slušelo tento pojem zadefiovat.

Definice 3.15. Mějme bezkontextovou gramatiku $G = (V_N, V_T, S, P)$. Gramatika je *levě rekurzivní*, pokud obsahuje pravidlo ve tvaru $(X \to X\alpha) \in P,\ X \in V_N, \alpha \in (V_N \cup V_T)^*$.

A nyní se levé rekurze zase zbavíme následujícím tvrzením.

Věta 3.16. *Nechť $G = (V_N, V_T, S, P)$ je bezkontextová gramatika, $X \in V_N$ nějaký její neterminál. Nechť*

$$X \to X\alpha_1, \ldots, X \to X\alpha_n, \quad \alpha_i \in (V_N \cup V_T)^*$$

jsou všechna levě rekurzivní pravidla pro neterminál X a

$$X \to \beta_1, \ldots, X \to \beta_m, \quad \beta_i \in (V_N \cup V_T)^*$$

jsou všechna ostatní pravidla pro neterminál X. Potom gramatika $G' = (V_N \cup \{X'\}, V_T, S, P')$ kde P' vznikne z P tak, že odebereme výše uvedená pravidla a místo nich přidáme pro $\forall i, \forall j$ pravidla

$$X \to \beta_i X'$$
$$X' \to \alpha_j X'$$
$$X' \to \lambda$$

kde X' je nový neterminál, generuje stejný jazyk jako G, tedy $L(G) = L(G')$.

Důkaz. Postupujeme velice podobně jako v důkazu 3.14. Tentokrát bereme taková slova, která obsahují nějaké podslovo $\beta_j\alpha_{i_1} \ldots \alpha_{i_k}$, byla tedy zderivována za použití pravidel (po řadě)

$$X \to X\alpha_{i_k}, \ldots, X \to X\alpha_{i_1}, X \to \beta_j$$

Místo nich však můžeme použít pravidla (po řadě)

$$X \to \beta_j X', X' \to \alpha_{i_1} X', \ldots, X' \to \alpha_{i_k} X', X' \to \lambda$$

Získáváme tak derivaci v gramatice G' a slovo je tedy prvkem jazyka $L(G')$.
Obdobně postupujeme i v případě opačné implikace. □

Jsme tedy schopni poměrně snadno transformovat levou rekurzi na pravou, čehož opět využijeme v rámci implementace parseru bezkontextové gramatiky.

Kapitola 4

Lexikální analýza

4.1 Dvoufázový model zpracování

V části o formálních gramatikách jsme krátce hovořili o podobnostech formálních a přirozených jazyků, zejména pak o rozdílu mezi tvořením jednotlivých slov a celých vět. Podotkli jsme, že v rámci našich definic nic jako věta zatím neexistuje. Otázkou je, zda by se tento dvouvrstvý model (slova, věty) dal prakticky využít i v našem případě. Rozeberme si z tohoto hlediska pověstnou gramatiku zpracování výrazů se sčítáním a násobením.

Příklad. Mějme gramatiku $G = (V_N, V_T, E, P)$, kde

$$V_N = \{E, T, F\}$$
$$V_T = \{+, *, (,), a\}$$

$$P:$$
$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid a$$

Gramatika je zcela v pořádku až do posledního přepisovacího pravidla $F \to a$. V reálném případě budeme samozřejmě chtít faktor (F) přepisovat nejenom na symbol a, nýbrž například na čísla nebo identifikátory proměnných. Gramatika by se tak musela rozšířit o definici čísel, identifikátorů apod. Přibyla by tedy například tato pravidla.

$$F \to N \mid I$$
$$I \to LI \mid L$$
$$L \to a \mid b \mid c \mid d \ \ldots \mid z$$
$$N \to UN \mid U$$
$$U \to 0 \mid 1 \ \ldots \mid 9$$

Gramatika se znatelně zesložitila a to jsme přidali pouze přirozená čísla a identifikátory složené pouze z malých písmen anglické abecedy.

Tento příklad nám ukázal, že takový způsob není příliš efektivní. Můžeme si navíc všimnout, že dodefinování čísel a identifikátorů nijak neovlivní původní synatktickou strukturu výrazu, ba co víc, z bezkontextového hlediska jejich gramatika není příliš nápaditá. Jazyk který jsme tímto kusem gramatiky definovali je totiž pouze regulární.

Tuto strukturu můžeme pozorovat u většiny umělých, především programovacích, jazyků. Nabízí se tedy myšlenka, zdali bychom toho měli využít. Podrobněji bychom mohli tento postup popsat následovně: „Dokud to jde, zpracovávej jazyk jako regulární, a až s výsledkem nakládej bezkontextově“. V praxi to pak vypadá tak, že je vstupní text „regulárně“ rozsekán na posloupnost tzv. lexikálních elementů (*lexémů*), které pak z větší části vstupují ve formě zástupných *tokenů* do druhé fáze jako terminály pro bezkontextovou gramatiku.

Příklad. V našem příkladu s gramatikou výrazů bychom tedy gramatiku zachovali víceméně nezměněnou. Nahradili bychom jen přepisovací pravidlo

$$F \to a \quad \text{pravidly} \quad F \to \mathbf{id} \mid \mathbf{number}$$

Pro názornost bychom mohli změnit terminály $+$, $*$, (,) na **plus**, **mul**, **left**, **right**. Množina terminálů by se tedy změnila na

$$V_T = \{\mathbf{plus}, \mathbf{mul}, \mathbf{left}, \mathbf{right}, \mathbf{id}, \mathbf{number}\}$$

S těmito terminály bychom v předchozí fázi počítali jako s výstupní abecedou. Definovali bychom sadu regulárních výrazů (použité regulární výrazy jsou oproti dřívější definici trochu rozšířené, jejich vysvětlení přijdě později, nicméně smysl je asi zřejmý)

- $[a - z]^+$ pro terminál **id**,
- $[0 - 1]^+$ pro terminál **number**,
- $+$ pro terminál **plus**,
- $*$ pro terminál **mul**,
- (pro terminál **left**,
- a konečně) pro terminál **right**.

Pomocí této sady bychom tedy například vstupní slovo tvaru $ab*(36+cd)$ transformovali na posloupnost tokenů **id mul left number plus id right**, které by bylo v druhé fázy zpracováno následující levou derivací:
$E \Rightarrow T \Rightarrow T$ **mul** $F \Rightarrow F$ **mul** $F \Rightarrow$ **id mul** $F \Rightarrow$ **id mul left** E **right** $\Rightarrow$
$\Rightarrow$ **id mul left** E **plus** T **right** $\Rightarrow^*$ **id mul left** F **plus** T **right** $\Rightarrow$
$\Rightarrow$ **id mul left number plus** T **right** $\Rightarrow^*$ **id mul left number plus** F **right** $\Rightarrow$
$\Rightarrow$ **id mul left number plus id right**

Úloha se tedy rozpadá na dvě části.

- Lexikální analýza
- Syntaktická analýza

Jednotlivé fáze byly naznačeny v příkladě a budeme se jimi nadále podrobně zabývat. Začneme, jak název kapitoly napovídá, fází první a to lexikální analýzou. Vraťme se tedy v našem povídání zpět do světa regulárních jazyků.

4.2 Reprezentace konečného automatu

Abychom mohli sestavit lexikální analýzu, budeme zajisté v první řadě potřebovat nějak rozpoznávat regulární jazyk. Již dříve jsme dokázali, že pro každý regulární jazyk existuje konečný automat, který tento jazyk rozpoznává. Dokonce jsme teoreticky ukázali způsob, jak takový automat získat z regulárního výrazu. Tyto dvě podkapitoly budou tedy z části opakování, nicméně probrané z implementačního hlediska více do hloubky.

Konečný automat je navíc struktura, která, jak za chvíli uvidíme, je počítačem poměrně snadno implementovatelná. Prvním krokem, který musíme zvládnout, je jeho datová reprezentace. Zatím jsme ukázali dvě – samotnou matematickou definici obsahující jakousi přechodovou funkci δ a grafickou reprezentaci pomocí diagramu. První zmiňovaná je velice učitečná ve formálním textu a důkazech, druhá se pro svou názornost hodí pro člověka. Z pochopitelných důvodů však obě zniňované reprezentace budeme jen težko implementovat v počítači. Uvedeme tedy ještě jednu, tentokrát vhodnou pro počítač.

Zmiňovaným způsobem je reprezentace pomocí dvourozměrné tabulky. Prvním rozměrem jsou stavy konečného automatu, druhým rozměrem všechny symboly abecedy. Obsahem tabulky je pak hodnota přechodové funkce při přechodu z daného stavu přes daný symbol.

Příklad. Tabulka pro automat naznačený na obrázku 2.1 by vypadala následovně

	a	b
→1	3	1
2	3	1
←3	2	2

Obrázek 4.1: Jednoduchý příklad konečného automatu vyjádřeného pomocí tabulky

Obsah tabulky nám tedy plně definuje přechodovou fukci δ a její rozměry definují množinu stavů automatu Q a použitou abecedu X, ovšem pouze pokud zvolíme vhodné uspořádání stavů a symbolů v abecedě. Je tedy rozumné přímo číslovat stavy

automatu a tyto hodnoty nadále používat jako indexy do tabulky. Při použití symbolů z klasické anglické abecedy pak můžeme využít jejího přirozeného uspořádání od A do Z.

Při přechodu se tedy pouze najde příslušný řádek, odpovídající stavu, ve kterém se automat aktuálně nachází, a sloupec, odpovídající symbolu, právě načtenému ze vstupní pásky. Je zcela zřejmé, že při této reprezentaci pracuje přechodová funkce s časovou složitostí $O(1)$. Samotná implementace dále vyžaduje nějakým způsobem popsat koncové stavy (například bitovou mapou) a pro běh je nutná již naznačená stavová proměnná (v jakém stavu se automat momentálně nachází).

Dalším krokem při sestavování konečného automatu by mohla být implementace některých operací, které se dají na automatech provádět. V podkapitole 2.3 o množinových operacích nad regulárními jazyky jsme si pár takových operací ukázali (vzhledem k jednoznačné korespondenci konečných automatů a regulárních jazyků můžeme tyto operace chápat i jako operace nad samotnými automaty) a dokonce jsme v 2.13, 2.15 a 2.16 dokazovali jejich uzavřenost pomocí konstrukce na konečných automatech. Tyto konstrukční důkazy (tak jak byly popsány) nyní s radostí využijeme.

Představme tedy funkce, které bude mít takový konečný automat jistě implementovány.

- `removeLambdaTransitions()` poskytuje odstranění veškerých λ-přechodů, převede tedy automat na nedeterministický konečný automat a následně na deterministický konečný automat, který je možno spustit a zpracovávat jím nějaký regulární jazyk. Přímo využívá poznatků dosažených v konstrukčních důkazech 2.11 a 2.8.

- `join()` poskytuje implementaci operátoru pro sjednocení regulárnách jazyků (konečných automatů), popsaný v důkazu 2.13(i).

- `concatenate()` je implementace operátoru žřetězení daný důkazem 2.15. Zde je však nutné podotknout, že při konstrukci zřetězeného automatu vznikají λ-přechody. Je tedy nutné použít algoritmus pro jejich odstranění popsaný v rámci funkce `removeLambdaTransitions()`.

- `iterate()` dává implementaci obecné iterace na konečných automatech popsanou v 2.16. Opět využívá algoritmu pro odstranění λ-přechodů.

- `positiveIterate()` přidává k obecné iteraci ještě iteraci pozitivní. Její konstrukci jsme explicitně neuváděli, nicméně jedná se o algoritmus téměř totožný, jako je použit v 2.16 s tím rozdílem, že nepřidáváme žádný nový stav s (protože jazyk L^+ neobsahuje prázdné slovo pokud ho neobsahuje původní jazyk L).

V implementaci může fungovat také jeden dodatečný mechanismus (nazveme ho mechanismem *accepted/working*), který do každého automatu přidává implicitní odpadní stav. To je takový stav, do kterého je možno spadnout, ovšem není možno z něj vyjít, všechny výchozí přechody jsou zacykleny zpět do tohoto stavu. Zajistí se

tak splnění definice konečného automatu v případě, kdy uživatel nedefinuje všechny přechody a navíc to přidává jednu důležitou funkčnost a tou je detekce chyb.

Mechanismus říká, že ve chvíli, kdy se automat dostal do odpadního stavu, již nedokáže přijmout žádné slovo a o výpočtu můžeme s klidem říci, že selhal. V opačném případě totiž nemáme jistotu, jestli se automat po nějakém počtu kroků do nějakého koncového stavu ještě nedostane. Rozšiřujeme tak definici konečného automatu o pojem „odmítá“, který se v normálním případě použít nedá.

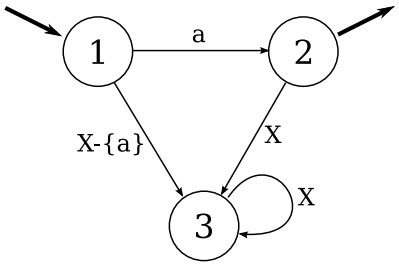
Vychází nám z toho tedy jednoduché pravidlo. Pokud je automat v koncovém stavu, pak *přijímá* (*accepted*), pokud je v chybovém stavu, pak *odmítá* (*rejected*) a ve všech ostatních případech automat stále *pracuje* (*working*).

4.3 Implementace regulárních výrazů

Jak jsme řekli v úvodu podkapitoly o regulárních výrazech, není pro člověka příliš vhodné zapisovat regulární jazyk přímo ve formě konečného automatu. Místo toho jsme našli rozumný kompromis ve formě regulárních výrazů. V 2.22 a 2.21 jsme ukázali, že každý regulární jazyk lze zapsat nějakým regulárním výrazem a definice 2.17, 2.20 a věta 2.18 nám naznačily, že se při převodu z regulárního výrazu na regulární jazyk a tedy konečný automat (tedy vyhodnocování regulárního výrazu) dá využít tří základních operací a to sjednocení, zřetězení a obecné iterace a jejich postupného iterování.

Předpokládejme, že již máme rozparsovaný regulární výraz (k tomu se vrátíme až později v rámci syntaktické analýzy) a jsme tedy schopni podle dané struktury výrazu provádět dané operace ve správném pořadí.

Jednotlivé operace již máme implementovány (předchozí podkapitola), jediné co tedy chybí, je implementace automatů, které zpracovávají elementární regulární výrazy, tedy takové, které jsou složeny pouze z jednoho symbolu abecedy (2.19, druhý bod). Konstrukce takového automatu je však velice jednoduchá. Následuje taková implementace pro příjem znaku $a \in X$.



Obrázek 4.2: Příklad elementárního konečného automatu přijímajícího právě jeden znak

Stav označený číslem 3 je tvz. odpadní stav. Zde je uveden pro úplnost, aby příklad odpovídal definici konečného automatu. Je asi zřejmé, že přechod přes celou množinu symbolů je zkratka pro přechody definované pro každý symbol v této

množině.

Je možné také implementovat malé vylepšení regulárních výrazů. Jako operátor zřetězení se většinou nepoužívá tečka, parser je uzpůsoben tak, aby se symboly řetězily bez použití jakéhokoli operátoru. Tečka místo toho zastupuje speciální symbol, který říká, že na tomto místě může být libovolný symbol z abecedy.

Dále je přidána zmiňovaná pozitivní iterace (která ve formální definici regulárních výrazů chyběla).

Další vylepšení je přidání symbolu výběru. Jedná se v podstatě o sjednocení na úrovni elementárních konečných automatů, které je oproti klasickému sjednocení značně optimalizováno (jelikož je implementováno na úrovni identifikátoru, tedy daný automat se vytvoří rovnou tak, aby zpracoval výběr). Zapisuje se tradičně jako hranaté závorky obsahující výčet symbolů, které má tento elementární automat rozpoznávat.

4.4 Lexikální analyzátor

Implementací regulárních výrazů jsme vyřešili podstatnou část ulohy, která se na straně lexikální analýzy provádí. Nyní je nutné tomu všemu dát nějakou formu. Nejprve si uvědomme, co je naším cílem a jaké jsou nutné kroky k jeho dosažení.

Od lexikální analýzy očekáváme, že vstupní text nějakým způsobem rozseká na lexémy a nahradí ho posloupností odpovídajících tokenů. Jejím vstupem tedy musí být množina nějakých vzorů, v našem případě množina regulárních výrazů, podle kterých se bude analýza rozhodovat. Bude tedy nutné zpracovávat několik regulárních jazyků najednou (pro každý regulární výraz).

Je zde sice možnost sjednotit všechny regulární výrazy (jazyky), přineslo by to však nutnost rozšířit definici konečného automatu o možnost rozlišovat různé druhy výstupních uzlů (pro každý zpracovávaný token jiný druh). Postupujme tedy raději jednodušší cestou a zpracovávejme několik regulárních jazyků paralelně, ale odděleně. V praxi to pak bude znamenat nějakou kolekci konečných automatů, kterým budeme podávat stejnou vstupní pásku.

Samotný výstup lexikální analýzy pak bude v rámci rozhraní reprezentován několika podstatnými funkcemi.

- `nextToken()` bude funkce, která vrátí další nalezený token. Opakovaným voláním tedy realizuje vytvoření posloupnosti tokenů.
- `tokenText()` bude funkce vracející hodnotu naposledy přečteného tokenu. Tou hodnotou se myslí právě lexém, tedy část původního textu odpovídající tokenu.

Příklad. Uveďme teď jeden krátký motivační příklad, který nám poodhalí pár problémů, které musíme v rámci lexikální analýzy zvládnout.

Mějme lexikální analýzu zpracovávající nějaký programovací jazyk, který obsahuje klíčová slova, identifikátory a bezznaménková celá čísla. Analyzátor tedy bude mimojiné obsahovat regulární výrazy tvaru

- *while* pro token **while**,

- $[a-zA-Z][a-zA-Z0-9]^*$ pro token **id**,
- a $[0-9]^+$ pro token **number**

Zpracovávejme nyní vstupní text ve tvaru *while*. Takový vstupní text jistě vyhovuje prvnímu vzoru, který generuje token **while** (což je ta správná varianta). Problém je však v tom, že tento vstupní text může vyhovovat také druhému vzoru, který generuje token **id**.

Nyní zpracujme vstupní text ve tvaru *whale*01. Situace se nám ještě více zkomplikovala, neboť takový text vyhovuje druhému vzoru (což je ta správná varianta), ovšem dokážeme ho také transformovat na posloupnost **id number** a to dokonce dvěma způsoby: $(whale(\mathbf{id})), (01(\mathbf{number}))$ a $(whale0(\mathbf{id})), (1(\mathbf{number}))$.

Jak tedy donutit analyzátor, aby takto nejasně definovanou množinu vzorů použil správně? Řešení se nabízí, vyjasníme definici.

Definice 4.1. *Lexikálním analyzátorem* myslíme pětici

$$LA = (X, \mathcal{R}, T, \delta, \omega)$$

- X značí vstupní abecedu,
- $\mathcal{R} \subset RL(X)$ je konečná neprázdná posloupnost délky $n \in \mathbb{N}$ regulárních jazyků $R_1, R_2, \ldots, R_n$ nad abecedou X reprezentovaných konečnými automaty

$$A_i = (Q_i, X, \delta_i, q_{i0}, F_i), \quad i = 1 \ldots n$$

- T je konečná neprázdná množina tokenů,
- $\delta : Q \times X \to Q$ je přechodová funkce, kde $Q = Q_1 \times \cdots \times Q_n$ je vektorový prostor dimenze n, kde i-tou složkou je stav $q_i \in Q_i$ z množiny stavů konečného automatu A_i. Funkce je definovaná následujícím způsobem

$$\delta((q_1, q_2, \ldots, q_n), x) = (\delta_1(q_1, x), \delta_2(q_2, x), \ldots, \delta_n(q_n, x)), \quad x \in X, q_i \in Q_i$$

- a $\omega : \mathbb{N} \to T \cup \{\epsilon\}$ je výstupní funkce, která každému indexu v posloupnosti $\mathcal{R}$ přiřadí výstupní token nebo speciální prázdný znak ϵ, pokud jazyk daný indexem nemá přidělen žádný výstupní token.

Definice 4.2. Mějme lexikální analyzátor LA. Jeho *rozšířenou přechodovou funkcí* $\delta^* : Q \times X^* \to Q$ myslíme funkci definovanou následovně

$$\delta^*((q_1, \ldots, q_n), \lambda) = (q_1, \ldots, q_n)$$
$$\delta^*((q_1, \ldots, q_n), wx) = \delta(\delta^*((q_1, \ldots, q_n), w), x), \quad x \in X,\ w \in X^*,\ q_i \in Q_i$$

Definice 4.3. Mějme lexikální analyzátor $LA = (X, \mathcal{R}, T, \delta, \omega)$, slovo $w \in X^*$ a nějaký jeho prefix $p \in X^*$. Řekneme, že prefix p je *přijímaný ve slově* w lexikálním analyzátorem LA pokud

- $\exists i : p$ je přijímáno automatem A_i.
- $\forall p' : p' \trianglelefteq w, (\exists i : p' \text{ je přijímáno automatem } A_i) \Rightarrow |p'| \leq |p|$.

  Neexistuje delší prefix, který by byl přijímaný nějakým automatem A_i.

Definice 4.4. Mějme lexikální analyzátor $LA = (X, \mathcal{R}, T, \delta, \omega)$, slovo $w \in X^*$, nějaký jeho prefix $p \in X^*$ přijímaný ve slově w lexikálním analyzátorem LA (4.3) a $n \in \mathbb{N}$. Řekneme, že lexikální analyzátor LA *přijímá prefix* p *n-tým vzorem*, pokud

- p je přijímáno automatem A_n.
- $\nexists i : p$ je přijímáno automatem $A_i, i < n$.

Věnujme pozornost především posledním dvěma definicím. Definice 4.3 především říká, že se snažíme získat vždy co nejdelší vyhovující lexém, to je tedy řešení druhého problému z příkladu (nejasnost mezi identifikátorem a dvojicí identifikátor, číslo). Definice 4.4 pak už jen přidává vzorům obsaženým v lexikálním analyzátoru určitou priorizaci, konkrétně říká, že máme použít vždy vzor s nejmenším indexem (např. dříve vloženým). Jak vidno, právě priorizace vzorů je řešením prvního problému (nejasnost, zda použít vzor pro klíčové slovo while nebo pro identifikátor).

Přímým důsledkem definic 4.3 a 4.4 jsou pak následující dvě tvrzení.

Věta 4.5. *Mějme lexikální analyzátor LA, slovo $w \in X^*$ a nějaký jeho prefix $p \in X^*$ přijímaný ve slově w lexikálním analyzátorem LA. Pak je prefix p určen jednoznaně.*

Věta 4.6. *Mějme lexikální analyzátor LA a slovo $w \in X^*$. Nechť LA přijímá nějaký prefix $p \in X^*$ slova w n-tým vzorem. Pak je číslo n určeno jednoznaně.*

Nyní již jen stačí upřesnit, co vlastně dělá funkce `nextToken()`. K dispozici má lexikální analyzátor a vstupní pásku, kterou pro splnění podmínek definice 4.3 položí jako slovo w. Podle téže definice najde nejdelší prefix přijímaný daným lexikálním analyzátorem a vzhledem k následující definici 4.4 najde příslušný index n, pod nímž se skrývá ten správný vzor. Na tento index dále použije výstupní funkci ω získá tak token $t \in T$, což je výsledek celé funkce `nextToken()`.

Ke konkrétní implementaci se dá využít mechanismus *accepted/working* který dokáže detekovat, kdy automat ještě počítá (je zde možnost na úspěšné namatchování) a kdy automat již selhal.

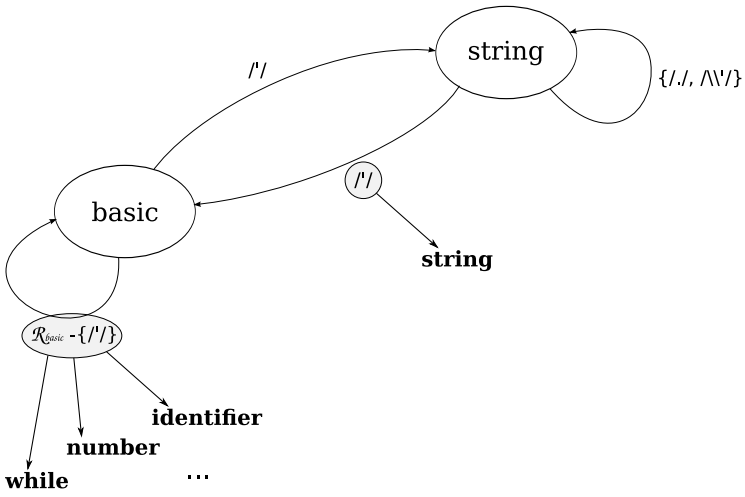
Samotný algoritmus je pak již jednoduchý. Zvětšujeme délku prefixu tak dlouho, dokud nějaký automat $A_1, \ldots A_n$ ještě pracuje nebo přijímá, tedy končíme ve chvíli, kdy všechny automaty selžou (nebo když narazíme na konec vstupní pásky). Následně vybereme poslední krok (4.3), kdy přijímal nějaký automat a zvolíme ten s nejmenším indexem (4.4).

Nalezený prefix uloží pro funkci `tokenText()`, která ho už jen vystaví. Jako poslední krok nesmíme opomenout zkrácení vstupní pásky o nalezený prefix.

Příklad. Představme si nyní situaci, kdy chceme pomocí lexikální analýzy rozpoznávat řetězcové konstanty, tedy například `'nějaký text'`. To samo o sobě nevypadá jako problém, ovšem uvědomme si, co se stane, pokud konstanta bude obsahovat apostrof (či jiný znak použitý pro uvození řetězcové konstanty).

Tradičně se tento problém řeší pomocí escape znaků (například zpětného lomítka), popř. zdvojení (tedy dva apostrofy za sebou znamenají jeden escapovaný apostrof). My však stojíme před úkolem napsat regulární výraz, který by byl schopný takový případ ošetřit. Takový regulární výraz by byl značně netriviální.

Pomožme si malým trikem. Rozdělme lexikální analyzátor (respektive posloupnost zpracovávaných regulárních jazyků) do více částí (tedy do více různých stavů) s tím, že nalezení nějakého vzoru (jedno provedení funkce `nextToken()`) dokáže převést analyzátor od příštího výpočtu do jiného stavu. Dostáváme tak jakýsi konečný automat postavený nad různými částmi původního analyzátoru, kde přechodová funkce nepracuje s abecedou, ale s indexy nalezených vzorů.



Obrázek 4.3: Stavy lexikálního analyzátoru při zpracování řetězconých konstant

Řešení problému s apostrofy se tak velice zjednodušuje. Nalezení apostrofu v základním stavu se analyzátor převede do stavu *string*. V tomto stavu jsou pak definovány vzory pro escapovaný apostrof a pro libovolný znak zvlášť. Navíc je zde vzor pro neescapovaný apostrof, který převede analyzátor zpět do základního stavu. Právě toto poslední pravidlo má definovanou výstupní funkci do tokenu **string** (4.3).

Kapitola 5

Syntaktická analýza

V jedné z předchozích kapitol jsme hovořili o dvoufázovém modelu zpracování vstupního slova. Slovo je v první fazi transformováno na posloupnost tokenů a následně ve druhé fázi rozpoznáno, říkáme také rozparsováno, nějakou bezkontextovou gramatikou. Již jsme probrali, jakým způsobem probíhá první lexikální fáze a nyní jsme tedy postaveni před situaci, kdy jako vstup dostáváme zmíněnou posloupnost tokenů a musíme provést fázi druhou, syntaktickou.

Hlavním cílem této fáze (syntaktické analýzy) je již tradiční rozhodnutí, zda slovo patří do daného (nyní již bezkontextového) jazyka či nikoliv. Takový výsledek je sice pěkný, avšak ne příliš užitečný. Přitom syntaktická analýza dokáže vytvořit mnohem lepší výsledek, než jen odpověď ano/ne. Takovým informačně bohatým zdrojem je derivační strom, který se dá pomerně jednoduše během analýzy zkonstruovat. U analýzy pomocí tzv. rekurzivního sestupu dokonce vzniká samovolně na zásobníku volání procedur. K tomu se však ještě dostaneme.

Stejně jako jsme u regulárních výrazů hovořili o různém způsobu jejich zápisu, tedy pomocí regulárních výrazů nebo přímo pomocí konečného automatu, budeme se i nyní zabývat způsobem zápisu a reprezentace bezkontextového jazyka v počítači.

Bezkontextový jazyk lze pochopitelně popsat slovně, ovšem tento způsob (stejně jako u regulárních jazyků) není možno zpracovat počítačem. Další fáze je vyjádření jazyka pomocí bezkontextové gramatiky. Toto vyjádření nám už dává určitý formální tvar, jazyk je jím exaktně definován a pro člověka je celkem dobře čitelný. Tento zápis bude tedy sloužit (stejně jako regulární výrazy) jako prostředník mezi uživatelem a programem.

Samotný počítač však nepracuje přímo s gramatikou jako takovou, nýbrž se zásobníkovým automatem, konkrétněji s parserem, jehož částí je nějaká forma zásobníkového automatu. Tato kapitola bude právě o různých typech parserů, jejich síle a zejména o převodu gramatiky do podoby takového parseru.

přirozený jazyk → bezkontextová gramatika → parser bezkontextového jazyka

5.1 Operátory FIRST a FOLLOW

Nejprve si zadefinujeme dva základní mechanismy, které velice zjednodušší celkový pohled na bezkontextové gramatiky a konstrukci jejich parserů. Jsou jimi operátory $FIRST$ a $FOLLOW$, v podstatě funkce, které nějakým způsobem pracují s danou gramatikou a odpovídají na dvě podstatné otázky – „Co může přijít jako první“ a „Co může následovat“.

Úmluva. V následující kapitole bude znakem \$ označován *konec vstupní pásky*, tedy jakýsi imaginární poslední znak, který značí, že již nenásleduje žádný vstup.

Operátor $FIRST$ definujeme primárně pro jednotlivé symboly (neterminály i terminály), sekundárně pak pro celé řetězce symbolů.

Definice 5.1. Mějme gramatiku $G = (V_N, V_T, S, P)$ a symbol $x \in V_N \cup V_T$, pak $FIRST(x) \subseteq (V_T \cup \{\lambda\})$ je množina terminálů, pro kterou platí

- $x \in V_T \Longrightarrow FIRST(x) = \{x\}$
- $x \to \lambda \in P \Longrightarrow \lambda \in FIRST(x)$
- $x \to y_1 y_2 \dots y_k \in P$
  $\exists a \in V_T \; \exists i \in \{1, \dots, k\}$ takové, že $a \in FIRST(y_i)$ a $\forall j < i, \lambda \in FIRST(y_j)$, potom $a \in FIRST(x)$
- $x \to y_1 y_2 \dots y_k \in P$
  $\forall j \in \{1, \dots, k\} : \lambda \in FIRST(y_j) \Longrightarrow \lambda \in FIRST(x)$

Operátor $FIRST$ pro symbol x je tedy množina terminálů, kterými začínají libovolné řetězce derivované z x. Pokud se x může zderivovat na λ, pak je λ také ve $FIRST(x)$

V praxi se však používá jeho rozšířená verze pro celé řetězce symbolů (speciálně tedy pro celé pravé strany pravidel gramatiky).

Definice 5.2. Mějme gramatiku $G = (V_N, V_T, S, P)$ a řetězec symbolů $x_1 x_2 \dots x_n$ kde $x_i \in V_N \cup V_T$, pak $FIRST(x_1 x_2 \dots x_n) \subseteq (V_T \cup \{\lambda\})$ je množina terminálů, pro kterou platí

- $\exists a \in V_T \; \exists i \in \{1, \dots, n\}$ takové, že $a \in FIRST(x_i)$ a $\forall j < i, \lambda \in FIRST(x_j)$, potom $a \in FIRST(x_1 x_2 \dots x_n)$
- $\forall j \in \{1, \dots, n\} : \lambda \in FIRST(x_j) \Longrightarrow \lambda \in FIRST(x_1 x_2 \dots x_n)$

Oproti tomu operátor $FOLLOW$ je definován jen pro jednotlivé neterminály.

Definice 5.3. Mějme gramatiku $G = (V_N, V_T, S, P)$ a neterminál $X \in V_N$, pak $FOLLOW(X) \subseteq (V_T \cup \{\$\})$ je množina terminálů, pro kterou platí

- $X = S \Longrightarrow \$ \in FOLLOW(X)$
- Pro libovolné pravidlo $A \to \gamma X \delta \in P$, pak $\forall a \in V_T : a \in FOLLOW(X) \Longleftrightarrow a \in FIRST(\delta)$.

  Přidáváme tedy všechny prvky z $FIRST(\delta)$, kde δ je nějaký řetězec gramatiky, který může následovat za zkoumaným neterminálem.
- Pro libovolné pravidlo $A \to \gamma X \in P$ nebo $A \to \gamma X \delta \in P$, kde $\lambda \in FIRST(\delta)$, pak $\forall a \in V_T : a \in FOLLOW(X) \Leftrightarrow a \in FOLLOW(A)$.

  Pro pravidla, která mají zkoumaný neterminál na konci, je nutné přidat všechny prvky $FOLLOW(A)$, kde A je levá strana daného pravidla.

Operátor $FOLLOW$ pro neterminál X je tedy množina terminálů, které se mohou vyskytovat za neterminálem X v nějakém řetězci, který vznikl derivací z počátečního neterminálu. Je-li navíc X nejpravější neterminál, obsahuje $FOLLOW(X)$ znak zastupující konec vstupního slova.

K čemu nám však takové funkce budou dobré? Představme si nyní, jakým způsobem by mohl počítač postupovat při analýze nějakého slova pomocí bezkontextové gramatiky. Buďme v situaci, kdy se snažíme přepsat nějaký neterminál, jsme tedy někde vprostřed výpočtu a někde uprostřed vstupní pásky. K dispozici máme nějakou sadu pravidel, které mají na levé straně onen neterminál a my stojíme před úkolem vybrat to správné.

Nabízí se myšlenka, že se rozhodneme podle následujícího symbolu (terminálu) na pásce. Pak už stačí jen zjistit kterými terminály mohou začínat nabízená pravidla a rozhodnout to správné. Právě k tomu se nám výborně hodí dříve nadefinovaný operátor $FIRST$ popř. $FOLLOW$ v případě, že narazíme na neterminál, jenž se dá přepsat na prázdné slovo λ.

Je zde však jeden podstatný problém. Co když se na pásce vyskytne terminál, který je možný začátek u dvou nebo více různých pravidel? Předpokládejme, že chceme postupovat deterministicky, potom není možné rozdělovat výpočet do více směrů. Sice ne dokonalým, ale alespoň malým řešením by mohla být změna kritéria pro rozhodování. Pokud bychom byli schopni se rozhodovat podle více terminálů na pásce než jen jednoho, měli bychom o něco silnější mechanismus pro rozlišování nabízených pravidel.

K takovému postupu však potřebujeme i silnější operátory $FIRST$ a $FOLLOW$ než ty, co jsme doposud zkonstruovali.

Definice 5.4. Mějme gramatiku $G = (V_N, V_T, S, P)$, řetězec symbolů $x_1 x_2 \dots x_n$ kde $x_i \in V_N \cup V_T$ a přirozené číslo $k \in \mathbb{N}$, pak $FIRST_k(x_1 x_2 \dots x_n) \subset V_T^*$ je množina slov složených z terminálů o délce nejvýše k, kterými začínají libovolné řetězce derivované z $x_1 x_2 \dots x_n$ (včetně prázdného slova λ).

Definice 5.5. Mějme gramatiku $G = (V_N, V_T, S, P)$, neterminál $X \in V_N$ a přirozené číslo $k \in \mathbb{N}$, pak $FOLLOW_k(X) \subset (V_T^* \cup \{\$\})$ je množina slov složených z terminálů o délce nejvýše k, které se mohou vyskytovat za neterminálem X v nějakém řetězci,

který vznikl derivací z počátečního neterminálu. Je-li navíc X nejpravější neterminál, obsahuje $FOLLOW_k(X)$ znak zastupující konec vstupního slova.

Nyní již stačí porovnávat pravidla pomocí zobecněných operátorů $FIRST_k$ a $FOLLOW_k$.

5.2 Zjemnění definice bezkontextové gramatiky

Ve smyslu předchozího povídání nyní zformulujme několik poznatků.

5.2.1 Analýza shora dolů

Naznačili jsme způsob analýzy, který staví na myšlence, že se ze známého neterminálu snažíme uhodnout výsledné slovo, a doufáme, že bude odpovídat vstupní pásce. Začínáme tedy počátečním neterminálem, pásku čteme zleva doprava a po každém přečtení terminálu z pásky provedeme (uhodneme) všechny derivace, které ve výsledném derivačním stromě stojí na cestě z kořene (počátečního neterminálu) do načteného terminálu.

Konstruujeme tedy derivační strom *odshora dolů* (od kořene k listům) a pokud si postup rozepíšeme na jednotlivé derivační kroky, zjistíme, že konstruujeme *levé* větné formy, tedy posloupnosti symbolů, které zleva začínají (načtenými) terminály, pak následuje neterminál, který budeme v následujícím kroku zpracovávat a pak následuje libovolná posloupnost terminálů a neterminálů, které se nějakým způsobem vygenerovaly v rámci předchozího výpočtu.

5.2.2 Analýza zdola nahoru

Toto však není jediný možný postup, jak zpracovávat bezkontextovou gramatiku. Odpoutejme se nyní od předchozí myšlenky a postupujme přesně opačně. Jako pevný bod si zvolíme slovo, které čteme z pásky. Naším cílem je teď přepsat toto slovo opačnou aplikací pravidel tak, abychom dospěli k počátečnímu neterminálu gramatiky.

Pásku opět čteme zleva doprava, a pro načtený terminál hledáme takové pravidlo, pomocí kterého mohl být tento terminál zderivován. Zjistíme tak, jaký neterminál je jeho otcem v derivačním stromě. Postupujeme tedy *zdola nahoru* – nejdříve známe potomky a až následně doplňujeme jejich rodiče, až v posledním kroku dostáváme kořen. Pokud náš postup zachytíme rozepsáním derivace, dostaneme v každém kroku *pravou* větnou formu, kde napravo stojí terminály, které ještě nejsou přečteny z pásky, před nimi stojí právě zjištěný neterminál a na začátku je posloupnost terminálů a neterminálů, které jsme vygenerovali předchozím výpočtem.

5.2.3 Názvosloví bezkontextových gramatik

Abychom měli v dalším povídání nějaký systém, zavedeme jednoznačné názvosloví dané šablonou $PDT(k)$, kde

- P značí prefix, který zajišťuje ještě jemnější dělení.
- D je směr čtení vstupní pásky.
- T je druh derivace, resp. větné formy generované v rámci derivace. Možné jsou tedy levá (L) a pravá (R).
- k značí počet terminálů používaných pro rozhodnutí a výběr správného pravidla. Toto číslo budeme nadále označovat jako *výhled* (*lookahead*).

5.2.4 Třídy bezkontextových gramatik

Všechny nově definované gramatiky jsou nějakým omezením třídy bezkontextových gramatik. Zformulujme nejprve třídu, kterou je možno použít při analýze shora dolů s obecným výhledem k. Zajišťuje to podmínka na prázdný průnik operátorů $FIRST$ u pravidel, které by mohly kolidovat.

Definice 5.6. Mějme přirozené číslo $k \in \mathbb{N}$, bezkontextovou gramatiku $G = (V_N, V_T, S, P)$ a libovolný neterminál X. Gramatika G je $LL(k)$ gramatika, pokud pro

- každá dvě pravidla $X \to \alpha, X \to \beta \in P$ kde $\alpha, \beta \in (V_N \cup V_T)^*, \alpha \neq \beta$
- a každé dvě levé větné formy $uX\gamma, vX\delta$ kde $u, v \in T^*,\ \gamma, \delta \in (V_N \cup V_T)^*$

platí $FIRST_k(\alpha\gamma) \cap FIRST_k(\beta\delta) = \emptyset$.
Speciálně pro $k = 1$ hovoříme o $LL(1)$ gramatice a můžeme použít jednoduchý operátor $FIRST$ (5.2).

A obdobně třídu gramatik, které jsou využitelné při analýze zdola nahoru.

Definice 5.7. Mějme přirozené číslo $k \in \mathbb{N}, k \geq 1$, bezkontextovou gramatiku $G = (V_N, V_T, S, P)$ a libovolný neterminál X. Gramatika G je $LR(k)$ gramatika, pokud pro

- každá dvě pravidla $X \to \alpha, X \to \beta \in P$ kde $\alpha, \beta \in (V_N \cup V_T)^*, \alpha \neq \beta$
- a každé dvě pravé větné formy $\gamma Xu, \delta Xv$ kde $u, v \in T^*,\ \gamma, \delta \in (V_N \cup V_T)^*$

platí $FIRST_k(u) \cap FIRST_k(v) = \emptyset$.

Definice 5.8. Třídou *deterministických bezkontextových jazyků* nazveme sjednocení všech tříd jazyků generovaných libovolnou $LR(k)$ gramatikou pro $\forall k \in \mathbb{N}, k \geq 1$.

Z faktu, že $LL(k)$ i $LR(k)$ gramatiky vznikají z bezkontextových gramatik pouhou restrikcí podmínek jasně plyne následující poznámka.

Věta 5.9. *Pro libovolné $k \in \mathbb{N}, k \geq 1$ platí, že*

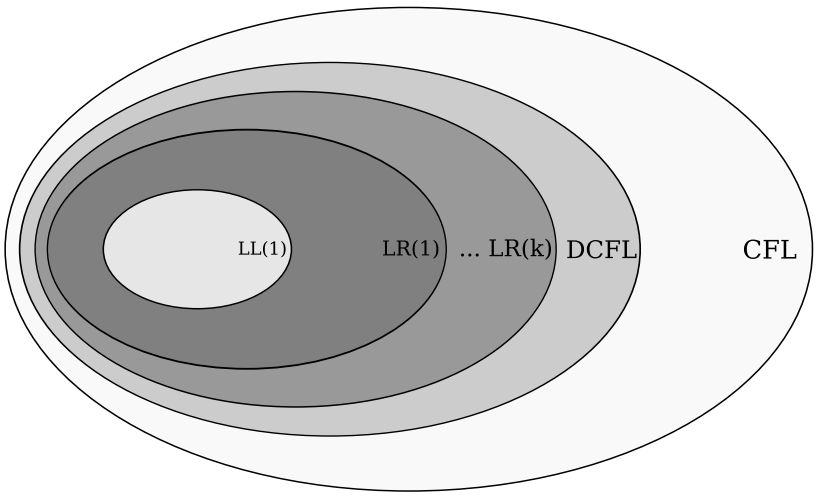
- *třída jazyků generovaných libovolnou $LL(k)$ gramatikou je podmnožinou třídy bezkontextových jazyků $\mathcal{L}_2$.*
- *třída jazyků generovaných libovolnou $LR(k)$ gramatikou je podmnožinou třídy bezkontextových jazyků $\mathcal{L}_2$.*

Také jsme naznačili, že větší výhled gramatiky podává větší možnost rozlišení pravidel, celkově tedy zvětšuje vyjadřovací sílu gramatiky. Platí tedy následující.

Věta 5.10. *Nechť $l, k \in \mathbb{N},\ k, l \geq 1,\ k \leq l$, potom*

- *třída jazyků generovaných libovolnou $LL(k)$ je podmnožinou třídy jazyků generovaných libovolnou $LL(l)$ gramatikou.*
- *třída jazyků generovaných libovolnou $LR(k)$ je podmnožinou třídy jazyků generovaných libovolnou $LR(l)$ gramatikou.*

Navíc platí, že třída jazyků generovaných libovolnou $LL(1)$ je vlastní podmnožinou třídy jazyků generovaných libovolnou $LR(1)$ gramatikou.



Obrázek 5.1: Množinové vztahy tříd bezkontextových jazyků

5.3 Rekurzivní sestup

Vraťme se k analýze shora dolů a představme jednu z jejích implementací. Sestavíme jednoduchý parser $LL(1)$ gramatiky založený na volání procedur a rekurzi. Myšlenka je jednoduchá. Každý neterminál je reprezentován jednou procedurou, jejíž obsah kopíruje pravé strany pravidel gramatiky s daným neterminálem na levé straně. Obsahuje-li pravá strana neterminál, obsahuje tělo volání procedury příslušného neterminálu. Obsahuje-li terminál, zavolá se procedura `match`, která zkontroluje daný terminál se vstupní páskou, konkrétně s výhledem.

V rámci popsané procedury se ještě na základě výhledu rozhoduje, které konkrétní pravidlo se má použít. Pravidlo s pravou stranou α se použije ve chvíli, kdy je výhled ve $FIRST(\alpha)$.

Definice 5.11. Mějme $LL(1)$ gramatiku $G = (V_N, V_T, S, P)$ kde v P jsou pravidla v obecném tvaru $X \to \alpha \mid \beta \mid \ldots,\ \alpha = a_1 a_2 \ldots a_n,\ a_i \in (V_T \cup V_N),\ \beta = b_1 b_2 \ldots b_m,\ b_i \in (V_T \cup V_N)$

Pak rekurzivní parser vypadá následnovně.

```
void X() {
   if (lookahead in FIRST(α)) {
      match(a1); // V případě, že a1 je terminál
      a2(); // V případě, že a2 je neterminál
      ...
   } else if (looahead in FIRST(β)) {
      b1(); // V případě, že b1 je neterminál
      match(b2); // V případě, že b2 je terminál
      ...
   ...
   } // Pokud existuje pravidlo X → λ, necháváme prázdné, jinak chyba
}

void a2() {
   ...
}

void b1() {
   ...
}

void match(token t) {
   if (lookahead == t) lookahead = nextToken(); // Posuneme vstupní pásku
   else error();
}
```

Výpočet spouštíme voláním procedury startovního neterminálu, tedy `S()`.

Všimněme si, že už samotné volání procedur vytváří derivační strom. Pokud vstupní slovo odpovídá gramatice, pak se výpočet vrátí z procedury `S()` a ve výhledu je znak $ zastupující konec vstupu.

Jak vidno, konstrukce takového parseru je velice jednoduchá, musí se však počítat s pár problémy. Obecně LL parserem nelze zpracovat gramatika obsahující levou rekurzi, parser by se totiž zacyklil. Tento popsaný parser by skončil v nekonečné rekurzi. Je tedy nutné gramatiku ručně zpracovat a levou rekurzi odstranit způsobem popsaným v 3.16.

Další podstatná nevýhoda je ta, že je v parseru napevno zakódována zpracovávaná gramatika, její případné úpravy nejsou triviální a například nastavení gra-

matiky až za běhu je zcela nemožné. Takový parser je tedy vhodný pro jednodušší napevno dané úlohy, např. pro parsování konfiguračního souboru.

5.3.1 Zpět k regulárním výrazům

V kapitole 4.3 jsme slíbili, že se ještě vrátíme k regulárním výrazům a osvětlíme způsob jejich rozparsování. Nyní již máme vše potřebné k tomuto kroku. Pro rozparsování použijeme právě rekurzivní sestup zkonstruovaný podle definice 5.11. Pro konstrukci použijeme následující $LL(1)$ gramatiku (zápis je již po odstranění levé rekurze).

$$G = (V_N, V_T, E, P)$$

$$V_N = \{E, E', T, T', F, F', G\}$$
$$V_T = \{\mathbf{join}, \mathbf{dot}, \mathbf{mul}, \mathbf{plus}, \mathbf{left}, \mathbf{right}, \mathbf{id}\}$$

P :
$E \to T\ E'$
$E' \to \mathbf{join}\ T\ E' \mid \lambda$
$T \to F\ T'$
$T' \to \mathbf{dot}\ F\ T' \mid \lambda$
$F \to G\ F'$
$F' \to \mathbf{mul}\ F' \mid \mathbf{plus}\ F' \mid \lambda$
$G \to \mathbf{left}\ E\ \mathbf{right} \mid \mathbf{id}$

5.4 Nerekurzivní analýza s predikcí

Rekurzivní sestup je sice jednoduchá, avšak velice neflexibilní implementace $LL(1)$ parseru. Podívejme se tedy ještě na jednu nyní již plnohodnotnou metodu, jak takového parseru dosáhnout.

Jádrem celého parseru bude struktura připomínající zásobníkový automat. Zásobníkové automaty (obdoba konečných automatů pro regulární jazyky) jsou teoreticko-informatické struktury, které jsou schopné rozpoznávat slovo pomocí bezkontextové gramatiky. Jejich formální definice však není příliš vhodná pro implementaci počítačem, neboť se takové automaty chovají nedeterministicky. Důležité je, že ke své práci požívají kromě vstupní pásky ještě zásobník, tedy jakousi paměť, na kterou jsou schopny během výpočtu ukládat informace a později je opět využívat. Více o zásobníkových automatech, včetně převodů mezi gramatikou a automatem najdete v [1].

Pro naše účely není nutné definovat zásobníkový automat v plně obecné formě, bude stačit, když představíme jeho jakousi deterministickou modifikaci šitou přímo na míru nerekurzivní analýze s predikcí.

Definice 5.12. *Zásobníkovým automatem* rozumíme čtveřici

$$A = (V_T \cup \{\$\}, V_N \cup V_T \cup \{\$\}, \mu, \$)$$

- V_T je množina terminálů (v našem případě odpovídá množině terminálů bezkontextové gramatiky). $V_T \cup \{\$\}$ je potom *vstupní abeceda*, tedy množina symbolů, které čteme ze vstupní pásky.
- V_N je množina neterminálů (opět odpovídá množině neterminálů nezkontextové gramatiky). $V_N \cup V_T \cup \{\$\}$ je *zásobníková abeceda*, tedy množina symbolů, které je možno umístit na zásobník.
- $\mu : V_N \times (V_T \cup \{\$\}) \to (V_T \cup V_N)^*$ je *parsovací tabulka*.
- a \$ je *počáteční zásobníkový symbol*, v našem případě pevně daný jako konec vstupu.

Na počátku výpočtu je na zásobník umístěn startovní neterminál gramatiky a vstupní páska posunuta na začátek. V každém kroku výpočtu se pak automat rozhoduje podle terminálního symbolu a na vstupní pásce (tedy podle jednosymbolového výhledu) a podle symbolu X na vrcholu zásobníku.

- Je-li $X = \$,\ a = \$$, vyčerpali jsme vstupní pásku a zásobník je prázdný, pak se automat s úspěchem zastaví.
- Je-li $X \in V_T,\ X = a$, tedy X je terminál odpovídající výhledu, pak se provede tzv. *redukce* – ze zásobníku se vyzvedne symbol X a posune se vstupní páska.
- Je-li $X \in V_N$, pak se provede tzv. *derivace* – ze zásobníku se vyzvedne symbol X a místo něho se přidá slovo $\mu(X, a)$. Slovo se na zásobník přidává od konce, aby na vrcholu zásobníku stál jeho první symbol. Použité přepisovací pravidlo je navíc vystaveno pro další použití, je jím konstruován odpovídající derivační strom.

Chybové situace nastavávají ve chvíli, kdy $X = \$ \ \&\ a \neq \$$, $X \neq \$ \ \&\ a = \$$ nebo není definován výsledek parsovací tabulky μ pro X a a.

Dá se vytušit, že právě parsovací tabulka je onen mechanismus pro rozhodování, jaké pravidlo má parser použít. Parametry funkce μ jsou tedy (vzhledem k předchozímu popisu analýzy shora dolů) přepisovaný (derivovaný) neterminál a výhled parseru. Výsledek je pravá strana pravidla, které se má použít (levá strana odpovídá přepisovanému neterminálu). Na zásobník si už jen ukládáme aktuální přepis, onu levou větnou formu, jejíž terminální začátek odebíráme operací reduce.

Tím jsme už také trochu naznačili, jakým způsobem budeme z gramatiky konstruovat parsovací tabulku. Opět se budeme držet popisu analýzy shora dolů a definice $LL(1)$ gramatiky. Pro neterminál X a výhled a potřebujeme, aby funkce μ dala pravou stranu pravidla, pro které platí, že na levé straně má neterminál X a a je ve $FIRST$ pravé strany. Pokud by takových pravidel existovalo více, nesplňovala by gramatika definici $LL(1)$ gramatiky. Dostáváme tak.

Definice 5.13. Mějme $LL(1)$ gramatiku $G = (V_N, V_T, S, P)$ a zásobníkový automat $A = (V_T \cup \{\$\}, V_N \cup V_T \cup \{\$\}, \mu, \$)$. Pro $\forall (X \to \alpha) \in P$ definujeme μ následovně.

- $\forall a \in FIRST(\alpha), a \neq \lambda : \mu(X, a) := \alpha$
- Pokud $\lambda \in FIRST(\alpha)$, pak $\forall b \in FOLLOW(X) : \mu(X, b) := \alpha$. To platí včetně symbolu $ pro konec vstupu.

Každé nedefinované políčko má význam chyby při parsování.

To nám opět dává poměrně jasný návod pro implementaci. Parsovací tabulku pochopitelně reprezentujeme dvourozměrnou tabulkou, jako u konečných automatů využijeme vhodně zvolené uspořádání na množinách terminálních a neterminálních symbolů. Je pouze nutné k terminálům dodefinovat znak ukončení vstupu. Uložení λ-pravidla reprezentujeme uložením prázdné posloupnosti symbolů jako obsah jednoho políčka tabulky.

Zásobník můžeme implementovat standardně jako dynamickou strukturu poskytující funkce `push` a `pop`.

Načtení gramatiky do parsovací tabulky se přímo řídí podle definice 5.13. Je tedy nutné implementovat i operátory $FIRST$ a $FOLLOW$. Tento postup je však jednoznačně popsán definicemi 5.2 a 5.3.

Před načtením gramatiky je možné algoritmicky aplikovat eliminaci levé rekurze a levou faktorizaci, což jsou postupy popsané v 3.16 a 3.14.

5.5 Syntaktický analyzátor

Syntaktický analyzátor je už jen obalující struktura, která obsahuje popsaný zásobníkový automat (parser) a lexikální analyzátor popsaný dříve. Její hlavní úkol je zajišťovat přechod dat od lexikální analýzy k bezkontextovému parseru (v podstatě realizuje vstupní pásku).

Volá tedy ve smyčce funkci `nextToken()` lexikálního analyzátoru a její výsledek podá parseru jako výhled. Zároveň může výsledek parseru (derivační strom) plnit dodatečnými daty, jako je například textová hodnota terminálů (funkce `tokenText()` lexikálního analyzátor), nebo pozice terminálu v textu.

Kapitola 6

Aplikace

Tím jsme se konečně dobrali k našemu cíli, zkonstruovali jsme úplný a implementovatelný parser daného bezkontextového (resp. $LL(1)$) jazyka včetně lexikální analýzy. Jsme tedy schopni vstupní text rozložit, zpracovat, rozhodnout, zda vyhovuje gramatice a zkonstruovat jeho (jednoznačný) derivační strom. Zbývá tedy doplnit, jak získané informace prakticky dále využít.

6.1 Využití při zpřehlednění zdrojového kódu

Aplikací se dá jistě vymyslet mnoho. V případě, že se jedná o kompilovaný (popř. interpretovaný) programovací jazyk, pak je základní způsob použití pochopitelně konstrukce překladače (interpreteru) daného jazyka. Právě lexikální a syntaktická analýza jsou základní prvky tzv. front-endu. Toto je však aplikace mnohonásobně přesahující rámec této práce.

Zde se budeme jen krátce zabývat jiným druhem a tím je metoda zpřehlednění zdrojového kódu.

Úmluva. Vzhledem k tomu, že budeme hovořit o možnostech ke zpřehlednění zdrojového kódu, budeme nadále pod pojmem „jazyk“ považovat libovolný formální programovací, skriptovací nebo jakýkoliv jiný jazyk, u kterého má smysl hovořit o zpřehledňování jeho zápisu.

6.1.1 Zvýraznění tokenů (highlighting)

Nejjednodušší aplikací předešlých poznatků je zvýraznění jednotlivých tokenů, respektive lexémů, které se na dané tokeny přepsaly v rámci lexikální analýzy. Využíváme tak pouze výsledků lexikální analýzy – kusy textu dané jako výstup lexikálního analyzátoru jsou v rámci vstupního textu zvýrazněny určitou barvou (popř. kurzívou, tučným písmem nebo jiným způsobem).

6.1.2 Zvýraznění neterminálů

Mějme nějaký neterminál v derivačním stromě, pak jistě všechny terminály vyskytující se v jeho podstromě tvoří po přepsání na lexémy souvislý blok textu (3.9). Neterminály se tedy v textu dají chápat jako souvislý (obvykle delší) blok nebo také jako sekvenci lexémů.

Pokud vezmeme nějakého potomka, pak jím generovaná sekvecne lexémů je opět souvislý blok a navíc je to jistě podmnožina sekvecne odpovídající původnímu neterminálu (protože podstrom tohoto potomka je podgrafem podstromu původního neterminálu). Platí tedy, že neterminál výše v derivačním stromě zaujímá rozsáhlejší blok – bloky odpovídající jeho i nepřímým potomkům jsou do tohoto bloku vnořeny. Kořenový neterminál pak objímá celý text.

Je tedy možné takové bloky v textu nějak zvýraznit. Dá se použít například zvýraznění pomocí barvy pozadí nebo pomocí orámování.

6.1.3 Formátování (indentace)

Při psaní zdrojového kódu je většinou žádoucí odsazení některých řádků o určitý počet prázdných znaků. Je to s highlightingem nejúčinější způsob zpřehlednění kódu. Indentaci lze také implementovat pomocí derivačního stromu a neterminálů.

Pokud nějaké neterminály označíme příznakem, který říká, že je chceme odsazovat, pak stačí pro začátek každé řádky spočítat, kolik takto označených neterminálů se vyskytuje v derivačním stromě na cestě od kořene k poslednímu terminálu předchozí řádky.

6.1.4 Párové tokeny

V rámci bezkontextové gramatiky jsou párové tokeny (například párová levá a pravá závorka) definovány v rámci jednoho pravidla, popř. v rámci více pravidel, které však nemají jinou možnost přepsání než takovou, že se ve výsledném textu budou párové tokeny vyskytovat vždy naráz nebo vůbec. Často se dá navíc gramatika přepsat tak, aby se tyto tokeny vyskytovaly právě na začátku a na konci nějakého neterminálu.

Stačí tedy nějak graficky vyznačit první a poslední lexém daného neterminálu.

Jako párové tokeny můžou kromě závorek sloužit např. **begin** a **end** v programovacím jazyce *Pascal*.

6.1.5 Skládání neterminálů

V některých editorech je možno skládat víceřádkové kusy textu do jedné řádky – dá se tak ušetřit místo na obrazovce a skrýt momentálně nepodstatné části textu. Takové skládání můžeme definovat i pro kusy textu korespondující (podle dříve naznačeného způsobu) s nějakým neterminálem v derivačním stromě.

6.1.6 Posouvání po tokenech

Stejně jako první popisovaná, je tato aplikace založena pouze na základě lexikální analýzy. Jedná se o pouhé posouvání (textového) kurzoru editoru po začátcích (prvních symbolech) jednotlivých lexémů.

6.1.7 Selekce řízená neterminály

Jak bylo řečeno, jednotlivé neterminály odpovídají souvislému bloku v textu. Je tedy jistě možné vybírat v textu právě tyto části. Jedná se v podstatě o pouhý posun (uzlového) kurzoru v derivačním stromě akorát vyznačený přímo v textu.

6.2 Editace derivačního stromu

Toto již není metoda zpřehlednění kódu, není to ani způsob využití výsledků parseru. Jedná se spíše o altenativní pohled na práci s daty (textem), kdy uživatel pouze doplňuje (nebo vytváří nanovo) derivační strom na základě načtené gramatiky. Výsledný text je pak obsažen v listech stromu (hodnoty terminálů zleva doprava (3.9)). Je zde tedy použit opačný postup než při parsování – z derivačního stromu se získává plochý text. To je pochopitelně úloha o mnoho jednodušší než dříve popsané parsování – zahrnuje pouhé procházení stromem a přepisování listů. [3]

Kapitola 7

Frey

Frey je textový editor, který implementuje výše popsané techniky a teorie. Kompletní distribuci projektu lze nalézt na přiloženém CD, včetně uživatelské dokumentace (instalace, konfigurace a používání aplikace) v podobě souborů `README` a `INSTALL`. Zde se budeme krátce zabývat strukturou projektu. Cílem je, aby čtenář snadno dohledal zmíněné algoritmy implementované ve zdrojovém kódu. Tyto algoritmy jsou v drtivé většině implementovány přesně tak, jak byly dříve popsány.

7.1 Struktura

Aplikace je členěna do několika jmenných prostorů, které odpovídají adresářové struktuře zdrojových kódů (popř. přímo zdrojovým souborům). Jsou jimi následující.

- `Core` (podadresář `src/core/`)

  Jmenný prostor obsahující programové jádro celé aplikace. Zahrnuje veškerou implementaci dané teorie až po kapitolu 5 včetně, pokrývá tedy implementaci konečných automatů, regulárních výrazů, lexikálního analyzátoru, parseru bezkontextové gramatiky, syntaktického analyzátoru a reprezentaci abeced nebo derivačního stromu.

  - `Alphabet` (podadresář `src/core/alphabet/`)

    Téměř celé jádro je psáno pomocí šablon tak, aby bylo možné parametrizovat naprogramované struktury různými použitými abecedami. Abeceda je nyní v našem slova smyslu nějaká třída a její instance jsou jednotlivé symboly.

    Právě v tomto jmenném prostoru se vyskytují definice používaných abeced, jsou jimi `CharAlphabet` – obyčejná znaková abeceda, `TokenAlphabet` – abeceda používaná pro výstup z lexikálního analyzátoru, v podstatě číselná abeceda, která reprezentuje jednotlivé tokeny (terminály) a `ExtensibleAlphabet` – číselná abeceda používaná pro neterminály, podporuje dynamické dodefinování nových neterminálů.

– `Regular` (podadresář `src/core/regular/`)

Část jádra pracující s regulárními jazyky, tedy implementace konečného automatu (`FiniteStateMachine`) včetně dříve popsaných operací (metod) `removeLambdaTransitions()`, `join()`, `concatenate()`, `iterate()` a `positiveIterate()` (kapitola 4.2) a mechanismu accepted/working.

V rámci jmenného podprostoru `RegularExpression` také najdeme parser regulárních výrazů (implementaci jejich převodu na konečný automat).

– `ContextFree` (podadresář `src/core/context_free/`)

Obsahuje části, které se týkají bezkontextových jazyků. Jedná se zejména o reprezentaci bezkontextové gramatiky (`ContextFreeGrammar`) včetně implementace levé faktorizace (3.14), odstranění levé rekurze (3.16) a operátorů $FIRST$ (5.1, 5.2) a $FOLLOW$ (5.3).

Další podstatnou částí tohoto prostoru je zkonstruovaný $LL(1)$ parser odpovídající parseru definovanému v rámci nerekurzivní analýzy s predikcí – `ContextFreeParser`.

Poslední je reprezentace derivačního stromu (`ParseTree`), coby výstupu z parseru.

Lexikální analyzátor (kapitola 4.4) je definován v rámci třídy `LexicalAnalyser` včetně rozšíření o stavy. Obdobně syntaktický analyzátor najdeme ve třídě `SyntaxAnalyser`, obojí v tomto jmenném prostoru.

Jak vidno, jediné, co z předešlé teorie zbylo pro ostatní jmenné prostory, jsou různé aplikace (kapitola 6). Následuje tedy letmý přehled jmenných prostorů, které na tomto základě staví samotný editor.

- `Engine` (podadresář `src/engine/`)

  Samotná aplikace pracuje ve dvou vláknech. První vlákno se stará o vnitřní chod aplikace, opakovaně provádí lexikální a syntaktickou analýzu editovaného textu, tedy o převod textu na derivační strom. To je implementováno v rámci tohoto prostoru ve třídě `Director` a funkci `textToTree()`. Kromě toho zde najdeme rutiny pro opačný postup, tedy linearizaci derivačního stromu zpět na text (funkce `treeToText()`).

  Ve druhém vlákně pak běží celé uživatelské prostředí, popsané v rámci dalšího bodu.

- `UI` (podadresář `src/ui/`)

  Uživatelské prostředí je postaveno na dvou základních pohledech. Jedná se o klasický textový pohled (v rámci `AbstractTextScreen`, `TextScreen`, `TextCursor`, ...), tedy editaci přímo zdrojového kódu. Zde jsou implementovány popsané aplikace jako například zvýrazňování tokenů, neterminálů, párové tokeny, skládání bloků či posun po tokenech. Ke všem aplikacím je však položen základ už textové paměti, o které bude řeč v následujícím bodě.

Jako druhý je pohled na editaci derivačního stromu (`AbstractTreeScreen`, `TreeScreen`, `TreeCursor`, `CacheTree`, ...). Editor vykreslí derivační strom získaný z editovaného textu a uživatel je schopen ho procházet nebo pozměnit, může mazat uzly, přidávat nové, měnit celou strukturu stromu, to vše v rámci pravidel daných gramatikou, nebo měnit textovou hodnotu jednotlivých tokenů. Tyto změny se pak reflektují zpět do textu (dříve popsaná funkcionalita v rámci jmenného prostoru `Engine`).

Vzájemné provázání těchto dvou pohledů je mimojiné realizováno i v podobě kurzorů editoru – kurzor v textu odpovídá po přepnutí do pohledu na derivační strom kurzoru ve stromě (ukazuje na tentýž token) a kurzor ve stromě se reflektuje i zpět do textového pohledu jako selekce bloků nebo pouhým přemístěním textového kurzoru.

- `Memory` (podadresář `src/memory/`)

  Obě popsaná vlákna pracují se společnými daty dvou druhů – textová paměť (textový soubor) a jeho reprezentace v podobě derivačního stromu. Takovými společnými strukturami jsou třídy `TextMemory` a `TreeMemory`. Ty jsou navíc „obaleny“ synchronizační třídou `Synchronizer` (a předkem `AbstractSynchronizer`), která zajišťuje sdílený přístup z více vláken (zamykání pomocí mutexů).

  Samotné paměti, které pracují na úrovni jednoduchých instrukcí (např. „načti znak na pozici x, y“ nebo „vrať délku řádky y“) mají jakési „nadstavby“ – třídy zajišťující pokročilejší funkce jako je zejména formátování textu (`TextMemoryIndenter`) a příprava neterminálních bloků pro uživatelské prostředí (`BlockProvider`).

  Dále v tomto jmenném prostoru najdeme symbolové tabulky (`SymbolTables`), které nesou dodatečné informace o gramatice, např. mapují zástupná čísla neterminálů na jejich pojmenování uvedené v definici gramatiky.

Mimo popsané jmenné prostory se vyskytují rutinní záležitosti procházející celým projektem, jako je např. hlášení chyb, lokalizace hlášek a zpráv nebo konfigurace.

7.2 Vývoj

Projekt prošel mnoha vývojovými fázemi, mnoho cest bylo slepých a i přes průběžné pročišťování a přepisování kódu zbylo mnoho situací, které by se daly vyřešit lépe a elegantněji.

V původním návrhu se počítalo se zpracováním jazyků o mnoho silnějších než jen $LL(1)$. Uvažovalo se dosažení úrovně až deterministických bezkontextových jazyků. To se však ukázalo jako nesnadno realizovatelné, ale co je podstatnější, nemělo by to přílišný praktický význam, neboť většina programovacích jazyků (cílová skupina) jsou jazyky slabší.

Také parametrizovatelnost struktur v jádře pomocí abeced se ukázala jako ne příliš praktická volba. Návrh by se bez ní jistě obešel a fungoval by i s použitím obyčejných ordinálních typů jako `integer` nebo `char`.

Zbylo také poměrně hodně kandidátů pro optimalizaci nebo vylepšení, ať už se jedná například o kompresi reprezentace konečného automatu, optimalizace některých operací na konečných automatech, silnější regulární výrazy nebo třeba použití parseru silnějšího bezkontextového jazyka (např. zamlčené $LALR$ nebo SLR parsery).

Do budoucna by se dalo jít ještě dál. Jednou z takových zajímavých cest by mohlo být navržení uživatelského prostředí pracujícího třídimenzionálně. Tím je myšleno jakési spojení textového a stromového pohledu do jednoho trojrozměrného modelu, který by ve dvou rozměrech pracoval s klasickou textovou obrazovkou a do třetího by stavěl derivační strom, se kterým by se jinak pracovalo stejně jako doposud. U takového grafického kabátku by sice hrozilo nebezpečí snížené praktické použitelnosti editoru, z hlediska studia struktury textu by to však mohlo být nečekaným vylepšením pohledu na věc.

Závěr

Nakonec jsou všechny popsané techniky a teorie v praxi opravdu realizovatelné a implementovatelné při udržení relativně nízké režie a vytížení výpočetních prostředků.

Technika založená na poctivém rozpoznávání vstupního textu však rozhodně nemusí být jedinou možnou a tou správnou cestou k dosažení cíle. Kupříkladu ke zvýrazňování lexémů syntaktickou analýzu vůbec nepotřebujeme, je tedy možné postavit plnohodnotnou aplikaci fungující jen s technikami popsanými v rámci lexikální analýzy. Odsazování řádků (indentace) může být jistě také implementováno pouze na základě znalosti struktury konkrétního jazyka, kupříkladu na základě premisy pro programovací jazyk *C*, že indentovaný blok začíná za levou složenou závorkou a končí před pravou složenou závorkou.

Mohlo by se tedy zdát, že jsme se pustili do věci až příliš obecně a složitě. To by platilo v případě, že by naším cílem bylo skončit a spokojit se s funkcemi, které se dají „nějak odhadnout“ bez použití formální definice jazyka pomocí gramatiky. My jsme však schopni jít mnohem dál.

Literatura

[1] Doc. RNDr. Roman Barták, Ph.D.: *Automaty a Gramatiky*, prezentace k přednášce.

[2] RNDr. Jakub Yaghob, Ph.D.: *Principy překladačů*, prezentace k přednášce.

[3] Marvin V. Zelkowitz: *A Small Contribution to Editing with a Syntax Directed Editor*, ACM SIGPLAN proceedings; ISSN 0362-1340; 1984.