Týden 5

Přednáška

Ukázali jsme jednoduchý převod konečného automatu na bezkontextovou gramatiku, čímž jsme prokázali, že *každý regulární jazyk je bezkontextovým jazykem (ale ne naopak).* (K automatu $A = (Q, \Sigma, \delta, q_0, F)$ jsme sestrojili [tzv. regulární] gramatiku $G = (\Pi, \Sigma, S, P)$, kde $\Pi = Q$, $S = q_0$, $P = \{q \longrightarrow aq' \mid \delta(q, a) = q'\} \cup \{q \longrightarrow \varepsilon \mid q \in F\}$.)

'Překladač' sestrojující k regulárnímu výrazu ekvivalentní konečný automat

Připomeňme si jednoznačnou gramatiku G pro jazyk $RV(\{a, b\})$

$$R \longrightarrow T + R \mid T \;\; ; \;\; T \longrightarrow FT \mid F \;\; ; \;\; F \longrightarrow F^* \mid (R) \mid C \;\; ; \;\; C \longrightarrow a \mid b$$

v níž jsme pro zjednodušení vynechali symboly $\emptyset$ a ϵ. Naše G tedy generuje jazyk v abecedě $\Sigma = \{\, a, b, +, ^*, (,) \,\}$, jehož prvky jsou příslušné regulární výrazy.

Zamyslíme se nad implementací algoritmu, který k zadanému regulárnímu výrazu sestrojí ekvivalentní konečný automat. Uvědomíme si, že v prvé řadě potřebujeme implementovat syntaktický rozbor zadaného regulárního výrazu, tedy konstrukci jeho syntaktického stromu. Víme, že tento strom se snadno odvodí např. z derivačního stromu podle uvedené gramatiky. Jde nám tedy o to, jak příslušný derivační strom (či jeho reprezentaci např. v podobě konkrétního odvození reg. výrazu v gramatice) sestrojit. Ukáže se, že se velmi přirozeně nabízí využití dynamické datové struktury *zásobník*; později budeme diskutovat ekvivalenci tzv. zásobníkových automatů (tj. konečných automatů, které navíc mohou využívat zásobník) s bezkontextovými gramatikami.

Teď ukážeme algoritmický přístup zvaný *rekurzivní sestup*, kde se zásobník neobjevuje explicitně, ale je skrytý v použití rekurzivního volání procedur.

Jako příklad řetězce z $L(G)$ vezměme

$$(aa + b)^* .$$

K němu by náš kýžený algoritmus měl sestrojit např. (jedinou existující) levou derivaci

$$R \Rightarrow T \Rightarrow F \Rightarrow F^* \Rightarrow (R)^* \Rightarrow (T + R)^* \Rightarrow (FT + R)^* \Rightarrow (CT + R)^* \Rightarrow (aT + R)^* \Rightarrow$$
$$\Rightarrow (aF + R)^* \Rightarrow (aC + R)^* \Rightarrow (aa + R)^* \Rightarrow (aa + T)^* \Rightarrow (aa + F)^* \Rightarrow (aa + C)^* \Rightarrow (aa + b)^*$$

Taková derivace je jednoznačně dána posloupností použitých přepisovacích pravidel. Když pravidla např. očíslujeme

$$1 : R \longrightarrow T + R, 2 : R \longrightarrow T, 3 : T \longrightarrow FT, \ldots, 8 : C \longrightarrow a, 9 : C \longrightarrow b,$$

lze uvedenou levou derivaci reprezentovat posloupností $2, 4, 5, 6, 1, 3, 7, 8, 4, 7, 8, 2, 4, 7, 9$.

Na přednášce jsme prodiskutovali návrh následujícího algoritmu (zapsaného ve formě spustitelného programu v Pythonu [**s jistými chybami**, viz cvičení]):

```
#funkce syntAnalRV vypise pro zadany rv (reg. vyraz) pravou derivaci pozpatku
#podle gramatiky s pravidly uvedenymi nize v poznamkach,
#pokud je retezec rv skutecne reg. vyrazem (nad abecedou {a,b})

def syntAnalRV(rv):

    idx = {'val' : 0} #aktualni index pri cteni rv zleva doprava

# pravidla s R na leve strane: R --> T+R | T, tedy 1: R --> T+R, 2: R --> T
    def zpR(): # zpracuj neterm. R, az v prubehu se zjisti, zda jde o 1 ci 2
        zpT()  # mj. precte max. prefix useku rv zacinajiciho na akt.indexu,
               # ktery je "termem" (odvozenym z T)
               # po zpracovani index ukazuje za onen prefix
        if  idx['val'] >= len(rv):
            print 2 #  2: R --> T
        elif rv[idx['val']] == '+':
            idx['val'] += 1
            zpR()
            print 1 #  1: R --> T+R

# T --> FT | F ... 3: T --> FT, 4: T --> F
    def zpT():
        zpF()
        if (idx['val'] < len(rv) and rv[idx['val']] in ('a', 'b', '(')):
            zpT()
            print 3 #  3: T --> FT
        else:
            print 4 #  4: T --> F

# F --> F* | (R) | C
# nahrazeno  F --> BH , B --> (R) | C , H --> *H | epsilon
# odstraneni leve rekurze F --> F*

# 5: F --> BH
    def zpF():
        zpB()
        zpH()
        print 5 # 5: F --> BH
```

```
# B --> (R) | C ... 6: B --> (R), 7: B --> C
    def zpB():
        if (rv[idx['val']] in ('a', 'b')):
            zpC()
            print 7 # 7: B --> C
        elif (rv[idx['val']] == '('):
            idx['val'] += 1
            zpR()
            if (rv[idx['val']] != ')'):
                raise Exception("ERROR 1")
            else:
                print 6 # 6: B --> (R)
                idx['val'] += 1
        else:
            raise Exception("ERROR 2")

# H --> *H | epsilon ... 8: H --> *H , 9: H --> epsilon
    def zpH():
        if (idx['val'] < len(rv) and rv[idx['val']] == '*'):
            print 8 # 8: H --> *H
            idx['val'] += 1
            zpH()
        else:
            print 9 #  9: H --> epsilon

# C --> a | b ... 10: C --> a, 11: C --> b
    def zpC():
        if (rv[idx['val']] == 'a'):
            print 10 #  10: C --> a
        elif(rv[idx['val']] == 'b'):
            print 11 #  11: C --> b
        else:
            raise Exception("ERROR 3")
        idx['val'] += 1

    zpR()
    print "OK"
# end of syntAnalRV(rv)

syntAnalRV("(aa+b)*")
```

Pozn. Šlo nám především o demonstraci algoritmu, nikoli o ideální program v Pythonu.

Naše upravená gramatika (s 11 pravidly v poznámkách v programu) je nejen jednoznačná, ale umožňuje i uvedený postup rekurzivním sestupem. Dala by se upravit na tzv. LL(1)-gramatiku, ale toto téma teď nebudeme blíže rozebírat.

Stručně dotáhneme myšlenku našeho malého překladače. Po přímočaré úpravě vydá výše uvedený program derivační strom pro daný regulární výraz (který mj. obsahuje informaci o příslušném syntaktickém stromu). Kořen zkonstruovaného stromu pak můžeme předložit funkci NFA, která sestrojí odpovídající konečný automat – stačí nám nedeterministický s případnými ε-šipkami. Níže uvedený pseudokód snad již nevyžaduje bližší komentář (ale prověření korektnosti ano).

automat NFA(node v)
(* procedura vracející k zadanému vrcholu v derivačního stromu automat, např. ve formě tabulky, který odpovídá podvýrazu určenému podstromem s kořenem v *)
{

if ($v.symb =$ 'a') return

	a	b	ε
$\rightarrow q_1$	q_2	-	-
((q_2))	-	-	-

;

if ($v.symb =$ 'b') return

	a	b	ε
$\rightarrow q_1$	-	q_2	-
((q_2))	-	-	-

;

if ($v.symb =$ 'R' and $v.rule =$ "$R \longrightarrow R + T$")
return UNION(NFA($v.succ_1$), NFA($v.succ_3$));
if ($v.symb =$ 'T' and $v.rule =$ "$T \longrightarrow FT$") return CONC(NFA($v.succ_1$), NFA($v.succ_2$));
if ($v.symb =$ 'F' [and $v.rule =$ "$F \longrightarrow BH$"]):
if $v.succ_2.succ_1.symb =$ '$*$' return ITER(NFA($v.succ_1$)) else return NFA($v.succ_1$);
if ($v.symb =$ 'B' and $v.rule =$ "$B \longrightarrow (R)$") return NFA($v.succ_2$));
if ($v.symb =$ 'H') return () # nevraci zadny automat;
otherwise return NFA($v.succ_1$)
}

Cvičení

Příklad 5.1

Proveďte „ručně“ běh algoritmu (či aspoň podstatnou část běhu algoritmu) syntAnalRV z přednášky např. pro regulární výraz $(ab)^* + a$. Znázorněte si rekurzivní volání zásobníkovou strukturou (třeba dnem nahoru jako na přednášce) a konstruujte přitom odpovídajícím způsobem derivační strom.

Příklad 5.2

Doplňte (neúplné) argumenty v poznámkách u programu, z nichž by měla vyplynout správnost, tedy, že výpočet končí s OK přesně pro (správně utvořené) regulární výrazy nad abecedou $\{a, b\}$. Přitom byste měli odhalit, že tomu tak není! (Jsou tam alespoň dvě různé chyby.)

Příklad 5.3

Uvažujme jazyk sestávající ze všech booleovských formulí s proměnnými $x1, x2, x3, \ldots$ a logickými spojkami $\neg, \wedge, \vee$; mohou se v nich používat závorky $(,)$, ale není nutné plně závorkovat. Každá taková formule je tedy řetězcem v abecedě

$$\Sigma = \{\, x, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \neg, \wedge, \vee, (,) \,\};$$

jako příklad může sloužit řetězec $(\neg x15 \vee x2 \wedge x5) \wedge \neg x21 \vee \neg(x2 \vee x5)$, který do jazyka patří. (Samozřejmě zde můžeme preferovat přehlednější zápis $(\neg x_{15} \vee x_2 \wedge x_5) \wedge \neg x_{21} \vee \neg(x_2 \vee x_5)$, ale to není podstatné.)

Navrhněte co nejjednodušší bezkontextovou gramatiku generující uvedený jazyk.

Takto navržená (jednoduchá) gramatika asi není jednoznačná; ověřte.

Zkonstruujte pak pro stejný jazyk jednoznačnou gramatiku, u níž derivační stromy přirozeně odpovídají obvyklé prioritě operátorů: negace váže silněji než konjunkce a konjunkce váže silněji než disjunkce. (Zkuste postupovat podobně jako při konstrukci bezkontextové gramatiky generující regulární výrazy, tedy přidáváním vhodných neterminálů.)

Příklad 5.4

Navrhněte bezkontextové gramatiky generující následující jazyky:

- $L_1 = \{\, w \in \{a, b\}^* \mid w \text{ obsahuje podslovo } baab \,\}$
- $L_2 = \{\, w \in \{a, b\}^* \mid |w|_b \bmod 3 = 0 \,\}$
- $L_3 = \{\, ww^R \mid w \in \{a, b\}^* \,\}$
- $L_4 = \{\, 0^n 1^m 0^n \mid m, n \geq 0 \,\}$
- $L_5 = \{\, 0^n 1^m \mid 1 \leq n \leq m \leq 2n \,\}$
- $L_6 = \{w \in \{a, b\}^* \mid w = w^R\}$,
- L_7 je jazyk posloupností v abecedě $\{\, (,), [,] \,\}$, které odpovídají správnému uzávorkování.

Příklad 5.5

Snažte se co nejvýstižněji charakterizovat jazyk generovaný gramatikou $S \longrightarrow bSS \,|\, a$. (Možná vám pomůže chápat a jako "atomický výraz" a b jako "binární operátor".)

Příklad 5.6

Navrhněte bezkontextovou gramatiku G tak, že $L(G) = L_1 \cdot L_2$ kde
$L_1 = \{w \in \{a,b\}^* \mid w$ obsahuje podřetězec $bab\}$
$L_2 = \{a^n u \mid u \in \{a,b\}^*$ a $1 \leq n \leq$ délka$(u) \leq 2n\ \}$
Přitom použijte S jako počáteční neterminál, a pokud možno jen jedno pravidlo s S na levé straně. (Snažte se o přehledný návrh využívající co nejméně pravidel.)
Nejprve navrhněte gramatiky G_1, G_2 pro jazyky L_1, L_2 a z nich jsme pak jednoduše získejte gramatiku pro $L_1 \cdot L_2$ (jak je naznačeno v části 5.2., s. 166). Uvědomte si, proč je obecně důležité zajistit, aby množiny neterminálů gramatik G_1 a G_2 byly disjunktní.

Příklad 5.7

Zjistěte, zda pro následující gramatiku G je $L(G) \neq \emptyset$, čili zda lze z neterminálu S vygenerovat alespoň jedno terminální slovo.
$S \longrightarrow aS \,|\, AB \,|\, CD$
$A \longrightarrow aDb \,|\, AD \,|\, BC$
$B \longrightarrow bSb \,|\, BB$
$C \longrightarrow BA \,|\, ASb$
$D \longrightarrow ABCD \,|\, \varepsilon$
Zobecněte svůj postup, tedy navrhněte algoritmus, který toto zjišťuje pro jakoukoli zadanou bezkontextovou gramatiku.
Přitom postupujte obecněji tak, že algoritmus pro danou $G = (\Pi, \Sigma, S, P)$ sestrojí množinu $\mathcal{T}_G = \{X \in \Pi \mid \exists u \in \Sigma^* : X \Rightarrow^* u\}$. Množinu $\mathcal{T}_G$ nejdříve zadejte jednoduchou induktivní definicí:

- $(X \to v) \in P, v \in \Sigma^* \Longrightarrow X \in \mathcal{T}_G$,
-

Příklad 5.8

(Nepovinně.)
Uvažujme jazyk $L = \{\, w \in \{a,b\}^*; |w| \geq 1$ a $|w|_a = |w|_b\}$.
Charakterizujte slova z $L^2 = L \cdot L$. Je pravda, že $L = L^2$? Platí případně alespoň jedna z inkluzí $L \subseteq L^2$, $L^2 \subseteq L$?
Charakterizujte slova z $L - L^2$.
Na základě předešlých úvah navrhněte bezkontextovou gramatiku generující L.

Příklad 5.9

(Nepovinně.)
Navrhněte bezkontextovou gramatiku generující jazyk všech palindromů v abecedě $\{a, b\}$, jejichž délka je násobkem tří.
(Jedná se tedy o jazyk $L = \{\, w \in \{a, b\}^* \mid w = w^R \text{ a } (|w| \bmod 3) = 0 \,\}$.)
Zkuste nejprve najít gramatiku používající jediný neterminál. Poté popřemýšlejte, jestli použitím více neterminálů dokážete počet potřebných pravidel zmenšit.
(Nakonec porovnejte s řešením Cvičení 4.13. na s. 137.)