GRAMATIKY A JAZYKY

URČENO PRO VZDĚLÁVÁNÍ V AKREDITOVANÝCH STUDIJNÍCH PROGRAMECH

HASHIM HABIBALLA

ČÍSLO OPERAČNÍHO PROGRAMU: CZ.1.07
NÁZEV OPERAČNÍHO PROGRAMU:
VZDĚLÁVÁNÍ PRO KONKURENCESCHOPNOST
OPATŘENÍ: 7.2
ČÍSLO OBLASTI PODPORY: 7.2.2

INOVACE VÝUKY INFORMATICKÝCH PŘEDMĚTŮ VE STUDIJNÍCH PROGRAMECH OSTRAVSKÉ UNIVERZITY

REGISTRAČNÍ ČÍSLO PROJEKTU: CZ.1.07/2.2.00/28.0245

OSTRAVA 2020

Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky

Recenzent: Doc. RNDr. PaedDr. Eva Volná, PhD.

Název: Gramatiky a jazyky
Autor: Doc. RNDr. PaedDr. Hashim Habiballa, PhD., Ph.D.
Vydání: druhé, 2020
Počet stran: 174

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

OBSAH

1 Konečný automat

V této kapitole se dozvíte:

- Čím se zabývá teorie formálních jazyků.
- Základní informace o hierarchii jazyků a problému determinismu a nedeterminismu.
- Princip konečného automatu.
- Vztah mezi automatem, jazykem a gramatikou.
- Základní pojmy teorie jazyků – abeceda, slovo, zřetězení, uzávěra.

Po jejím prostudování byste měli být schopni:

- Navrhnout konečný automat pro jednoduchý jazyk.
- Definovat konečný automat, včetně jeho výpočtu.
- Definovat jazyk rozpoznávaný konečným automatem.
- Transformovat konečný automat nedeterministický (NKA) na deterministickou variantu (DKA).
- Dokázat vztah mezi jazyky rozpoznatelnými NKA a DKA.

Klíčová slova této kapitoly:

Teorie formálních jazyků, abeceda, slovo, zřetězení, uzávěra, konečný automat.

Průvodce studiem

Studium této kapitoly je poměrně náročné zejména pro ty z Vás, kteří dosud nemají žádné zkušenosti s matematickou formalizací pojmů. Na druhou stranu se zde snažíme vyložit nejprve intuitivní náhled na problematiku. Určitě tomuto úvodnímu seznámení věnujte velkou pozornost, neboť pochopením této části se Vám usnadní studium následujících kapitol.
Na studium této části si vyhraďte alespoň 6 hodin. Doporučujeme studovat s přestávkami vždy po pochopení jednotlivých podkapitol. Po celkovém prostudování a vyřešení všech příkladů doporučujeme dát si pauzu, třeba 1 den, a pak se pusťte do vypracování korespondenčních úkolů.

Teorie formálních jazyků je velmi důležitou součástí teoretické informatiky. Základy novodobé historie této disciplíny položil v roce 1956 americký matematik Noam Chomsky, který v souvislosti se studiem přirozených jazyků vytvořil matematický model gramatiky jazyka.
I když původní motivace – vyvinout model pro přirozené jazyky jako je angličtina – zdaleka nebyla tak úspěšná, teorie formálních jazyků stojí za dynamickým rozvojem informačních technologií, tak je známe dnes. Její výsledky umožnili tvorbu a automatické zpracování jazyků pro pokročilé programování počítačů, pro databázovou technologii a její aplikace lze vidět prakticky denně.

Začala se prosazovat teorie jazyků, která nyní obsahuje bohaté výsledky v podobě matematicky dokázaných tvrzení teorémů. Tato teorie pracuje se dvěma duálními matematickými entitami, s gramatikou a s automatem, představující abstraktní matematický stroj. Zatímco gramatika umožňuje popsat strukturu vět formálního jazyka, automat dovede tuto strukturu identifikovat.
Jak už jsme uvedli, původní dosti optimistická představa formalizace procesů zpracování přirozených jazyků nebyla příliš úspěšná (přirozené jazyky jsou prostě příliš složité). Přesto již v 60. letech 20. století Chomského definici gramatiky formálního jazyka a klasifikaci formálních jazyků (Chomského hierarchie jazyků) použili Backus a Nauer pro definici syntaxe programovacího jazyka Algol 60 (ve tvaru formalismu, jež se nazývá Backus-Nauerova forma). Další vývoj pak přímočaře vedl k aplikacím teorie jazyků v oblasti překladačů programovacích jazyků. Stanovení principů syntaxí řízeného překladu a generátorů překladačů (programovacích systémů, které na základě formálního popisu syntaxe a sémantiky programovacího jazyka vytvoří jeho překladač) představuje kvalitativní skok při konstrukci překladačů umožňující automatizovat náročnou programátorskou práci spojenou s implementací programovacích jazyků.

1.1 Základní pojmy teorie jazyků

Abychom mohli ve studiu TFJA pokračovat, zavedeme nejprve základní definice, ze kterých budeme vycházet.

Abeceda

Abeceda

Definice 1: **<u>Abeceda</u> Σ je konečná množina symbolů.**

Řetězec (slovo)

Řetězec (slovo)

Definice 2: **<u>Řetězec</u> (<u>slovo</u>) α prvků z konečné množiny Σ je libovolná konečná posloupnost prvků této množiny. Řetězce zpravidla označujeme řeckými písmeny. Počet prvků v řetězci udává jeho délku a označujeme ji $|\alpha|$. Řetězec, který neobsahuje žádný prvek, nazýváme prázdný řetězec a označujeme ho ε nebo e (nedojde-li k záměně). Jeho délka je 0.**

$\alpha = a_1a_2...a_{k-1}a_k$ $a_i \in \Sigma$, $i = 1,2,...,k$; $|\alpha| = k$

Pozn. Velmi dobře znáte řetězce z běžného života – např. Vaše jméno je řetězec složený z písmen české abecedy.

Řešený příklad 1:
Mějme množinu Σ={0,1}. Řetězce prvků této množiny jsou binární čísla bez znaménka. Například
α = 001011 |α| = 6
β = 0000 |β| = 4

1.1.1.1.1.1.1 Konkatenace (zřetězení), uzávěry množiny

Definice 3: **Zřetězení (konkatenace) řetězců $\alpha=a_1a_2\ldots a_r$ a $\beta=b_1b_2\ldots b_s$ je řetězec $\alpha\beta=a_1a_2\ldots a_rb_1b_2\ldots b_s$.**

Zřetězení (konkatenace)

Pozn. Pro délku konkatenace αβ dvou řetězců α a β platí |αβ| = |α| + |β|. Dále pro konkatenaci libovolného řetězce α s prázdným řetězcem ε platí α·ε = ε·α = α

Konkatenace může být provedena také několikanásobně, potom používáme následující značení: α^n, přičemž označuje konkatenaci řetězce α provedenou n-krát. Například $(01)^3$ je řetězec 010101.

Pozn. Zřetězení opět není nic složitějšího než spojení dvou řetězců k sobě.

Uzávěry

Definice 4: **Pozitivním uzávěrem Σ^+ konečné množiny prvků Σ nazveme množinu všech řetězců sestavených z prvků množiny Σ bez prázdného řetězce. (Uzávěr Σ^+ je spočetná množina.)**

Definice 5: **Uzávěr Σ^* množiny Σ navíc obsahuje prázdný řetězec ε, tj. je definován $\Sigma^* = \{ \varepsilon \} \cup \Sigma^+$**

Pozn. Tyto uzávěry jsou tedy všechny možné kombinace, jak nějakou množinu můžeme prokombinovat spojením jejich řetězců libovolně-krát za sebou (viz příklad).

Řešený příklad 2:
Nechť množina prvků je Σ = { a,b,c } Její pozitivní uzávěr a uzávěr je (pouze některé prvky – uvědomte si, že je nekonečný!)
Σ^+ = { a,b,c,aa,ab,ac,ba,bb,bc,ca,cb,cc,aaa,aab,… }
Σ^* = { ε,a,b,c,aa,ab,… }

1.1.1.1.1.1.2 Jazyk

Formální jazyk

Definice 6: **Nechť je dána abeceda Σ, pak libovolná podmnožina uzávěru Σ^* je formálním jazykem nad abecedou Σ. Tedy formální jazyk L je definován $L \subseteq \Sigma^*$ (pro jednoduchost budeme mluvit o jazyce).**

Specifické případy:

- Je-li L prázdná množina (L = ∅), je to prázdný jazyk.
- Je-li L konečná podmnožina, je to konečný jazyk.
- Je-li L je nekonečná podmnožina, je to nekonečný jazyk.

Jazyk je tedy výběrem slov v určité abecedě ze všech možných kombinací symbolů. Může samozřejmě obsahovat všechna slova, žádné nebo některé. Je jasné, že jazyk může být různě složitý (jak jsme to již probírali).

Řešený příklad 3:

1.1.1.1.1.2 Nechť abeceda je

Σ = { +,-,0,1,2,…,9 }
Pak množina celých čísel je jazykem nad Σ.

Uvedená definice formálního jazyka je pro praktické použití příliš obecná. Pouze definuje, co jazyk je, ale neposkytuje žádné prostředky pro popis struktury jazyka nebo zjištění, zda určitý řetězec do jazyka patří.

Pojmy, které jsme si právě zavedli, vás budou provázet celým studiem. Pokuste se je nyní znovu projít a zkuste si pro každý z nich představit konkrétní příklad, podobně jako jste je viděli v předchozích úlohách. Uvidíte, že pokud si hned od začátku studia budete za každým matematizovaným pojmem představovat konkrétní příklady ze života, bude pro Vás mnohem jednodušší pracovat s nimi ve složitějších kapitolách.

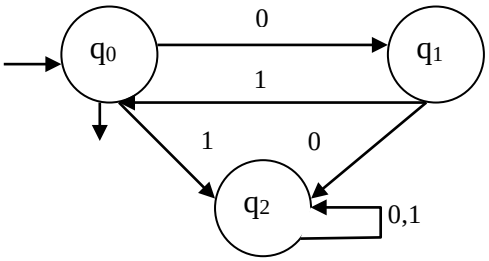
1.2 Konečný automat

Mluvíme-li o jazyce jako množině slov a gramatice jako o souboru pravidel, která umožňují generovat slova z jazyka, pak automat je prostředkem, jak zjistit, která slova do jazyka patří a která ne.

Konečný automat

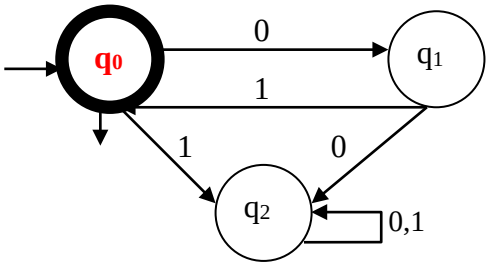
Pokuste se představit si následující stroj - automat. Má pásku, na kterou můžete zapisovat po symbolech slova, která chcete rozpoznat, zda patří do jazyka nebo ne. Dále má řídící jednotku se stavy, které si můžete představit jako žárovky, které se rozsvítí, pokud je automat právě v tom konkrétním stavu. Z pásky umí automat číst pouze po jednotlivých symbolech a nemůže se vracet zpět na již přečtené symboly. Je důležité, abyste si uvědomili, že automat si nemůže nic pamatovat. Vždy vidí jen jeden symbol ze zkoumaného slova. Dále automat ví, jak reagovat na situaci pokud je v nějakém stavu a na pásce vidí určitý symbol. Vždy je pak výsledkem takové akce přechod do nějakého stavu (může jít i o stejný stav). Nakonec má automat k dispozici informaci, ve kterém stavu má začít a který stav je koncový (nebo více stavů) a pokud se dostane do takového stavu, pak je slovo z jazyka.
Podívejme se na příklad:

010101..... Páska se zkoumaným slovem

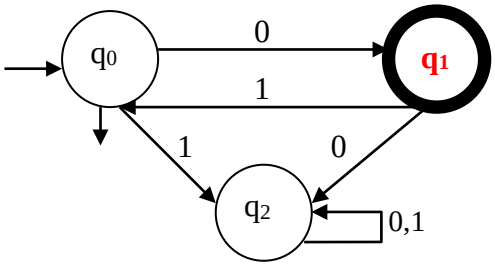


Řídící jednotka má kruhové symboly (stavy), které se během práce automatu mohou stávat aktivní (při začátku čtení slova je aktivní stav do kterého vstupuje šipka), koncový stav je stav, ze kterého šipka vystupuje.
Šipky mezi stavy určují, jaký stav se stane aktivní místo stávajícího při přečtení symbolu, který je u šipky, z pásky. (nazývají se přechody mezi stavy)

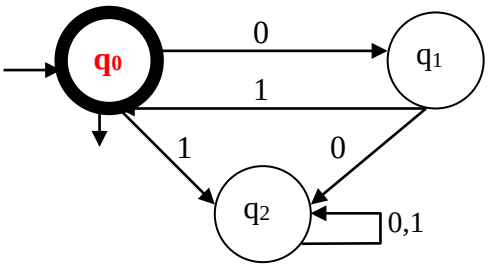
A nyní si znázorněme automat v činnosti. Postupně načteme slovo 0101 a vždy zvýrazníme ten stav, který je právě aktivní. Ve čteném slově pak bude zvýrazněn ten symbol, který bude právě přečten.



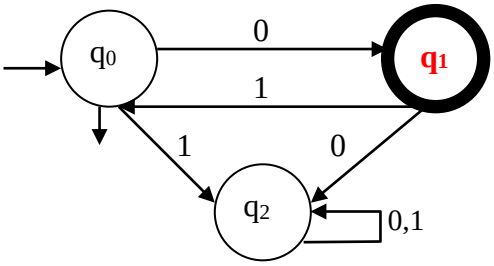
Počáteční situace – automat začíná svůj výpočet a je právě ve stavu q_0 a chystá se přečíst ze slova **0**101 první symbol.



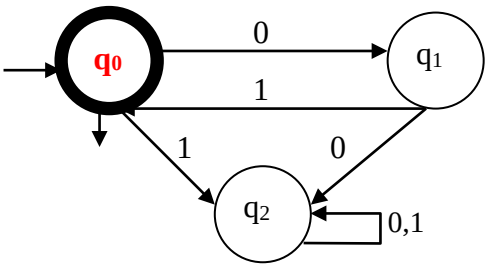
Automat přečetl první symbol 0 ze slova 0**1**01 a podle definovaného přechodu se stal aktivní stav q_1



Po přečtení symbolu 1 ze slova se automat dostal opět do stavu q_0. Ve slově nyní následuje další symbol 0 - 01**0**1



Automat je opět ve stavu q_1 a zbývá přečíst poslední symbol slova 010**1**



V poslední fázi výpočtu již není symbol, který by se dal z pásky přečíst. Proto zbývá zjistit, zda se automat dostal do stavu, který je koncový – má výstupní šipku. Je tomu tak, proto říkáme, že automat slovo 0101 rozpoznal – přijal.

Jak vidíte, princip tohoto typu automatu není nijak složitý. Doufám, že jste pochopili, jak se z počátečního (vstupního) stavu postupně dostává do jiných pomocí čtení jednotlivých symbolů na pásce. Uvědomte si, že automat přečte celé slovo a pak je otázka, zda skončil ve stavu koncovém nebo ne. Pokud ano, slovo patří do jazyka, který automat umí rozpoznávat. Tedy můžeme hovořit o slovech, které automat rozpoznává (patří do jazyka rozpoznatelného tímto automatem) a které ne. Zkusme spolu uvažovat o jaký jazyk v tomto konkrétním případě jde.

Pokud se blíže na automat podíváte, pak uvidíte, že pokud dostává na pásku slova, která obsahují přesně za sebou sled symbolů 01, pak se pohybuje mezi stavy q_0 a q_1. Pokud by však dostal po symbolu 0 další symbol nula, pak skončil ve stavu q_2, ze kterého by se už pak nemohl dostat jinam, protože přechod na 0 i 1 ze stavu q_2 vede opět do q_2. Stejná situace nastane, pokud se po symbolu 1 bude číst další symbol 1. Stav q_2 je v podstatě jakási „černá díra“, která se stará o situace, které nastanou ve slovech, která nechceme přijmout. Takže tento automat přijímá slova, která jsou libovolně krát zřetězeným slovem 01. A všimněte si, že rozpozná i slovo, které neobsahuje nic (tedy prázdné slovo), neboť počáteční stav q_0 je zároveň koncovým.
Zapišme tedy jazyk, který automat rozpoznává do matematického zápisu.

$L = \{ (01)^n, n \geq 0 \}$ (jinak řečeno, automat rozpoznává ta slova, která obsahují posloupnosti 01 v libovolném množství za sebou)

Uvědomme si, že automat také nemusí v určitém kroku mít definováno, do jakého stavu jít, nebo může mít více možností. Pak hovoříme o nedeterministickém automatu. S pojmem nedeterministický se budeme setkávat i u dalších typů automatů. Nedeterministický by se dalo volně přeložit jako takový, který nemůže v určitém okamžiku jednoznačně určit, co dělat.

1.3 Konečný automat (deterministický) - KA

Definice 7: ***(Deterministickým) konečným automatem (DKA)*** **nazýváme každou pětici $A = (Q,\Sigma,\delta, q_0, F)$, kde**

- **Q je konečná neprázdná množina (*množina stavů, stavový prostor*)**
- **Σ konečná neprázdná množina (*množina vstupních symbolů, vstupní abeceda*)**
- **δ je zobrazení $Q \times \Sigma \rightarrow Q$ (*přechodová funkce*)**
- **$q_0 \in Q$ (*počáteční stav, iniciální stav*)**
- **$F \subseteq Q$ (*množina koncových stavů, cílová množina*).**

Deterministický konečný automat

Definice 8: Přechodovou funkci $\delta: Q \times \Sigma \rightarrow Q$ konečného automatu $A = (Q,\Sigma,\delta, q_0, F)$ rozšíříme na *zobecněnou přechodovou funkci* $\delta^*: Q \times \Sigma^* \rightarrow Q$ následovně: $\delta^*(q,e)=q$, $\forall q \in Q$ a $\delta^*(q,wa)=\delta(\delta^*(q,w),a)$, $\forall q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$.

Pozn. Zobecněná přechodová funkce není opět nic složitého – jde o rozšíření, které umožňuje zkoumat, kam se dostaneme z určitého stavu na slovo (tedy několik symbolů místo jednoho). Definici si lépe přečtete, pokud si vzpomenete na funkci faktorial. Zřejmě jste v programování konstruovali algoritmus této funkce pomocí rekurze. Víte, že faktorial(n) = n * faktorial(n-1) a faktorial(0) = 1. Přesně toto říká definice pro tuto zobecněnou funkci. Tedy, že problém přechodu na slovo lze rozložit na problém přechodu na poslední symbol slova (což umíme podle obyčejné přechodové funkce) a pak jen stačí zbytek slova řešit stejným postupem, až dojdeme na prázdné slovo. Prázdné slovo odkudkoliv nemůže způsobit nic jiného, než že se zůstane ve stejném stavu.

Zobecněná přechodová fce

Jazyky rozpoznávané a rozpoznatelné KA

Jazykem rozpoznávaným konečným automatem ***A*****, pak nazveme množinu** $L(A)=\{ w|w \in \Sigma^* \wedge \delta^*(q_0,w) \in F \}$.

Řekneme, že *jazyk* L (nad abecedou Σ) *je rozpoznatelný konečným automatem* , jestliže existuje konečný automat *A* takový, že L(*A*)=L.

Konečné automaty reprezentují výše uvedené množiny a zobrazení. Kromě přechodové funkce by mělo být pro Vás poměrně jednoduché pochopit, co jsou jednotlivé množiny. Přechodová funkce určuje, do jakého stavu se přechází na daný symbol a aktuální stav. Proto má strukturu danou kartézským součinem v definici (osvěžte si pojem kartézského součinu z teorie množin).

Rozlišujte prosím, co je jazyk rozpoznávaný a rozpoznatelný!

Rozpoznávaný je jazyk konkrétním automatem – jde o slova, která ho dostanou do koncového stavu. Tedy například u našeho automatu na kávu jde o všechny posloupnosti, jak lze do něj vhodit mince o celkové hodnotě 5 Kč (např. posloupnost 1Kč, 2 Kč, 2 Kč, a další)

Rozpoznatelný je jazyk tehdy, pokud k němu vůbec lze sestrojit KA. Říkali jsme, že jazyky jsou různě složité. Např. pro jazyk Pascal byste nedokázali sestrojit takový automat (jenž by skončil v koncovém stavu pro programy správně napsané), protože na to je jazyk příliš složitý. Jak uvidíme v pokročilých kapitolách, je konečný automat příliš „slabá“ formalizace na takový jazyk jako je Pascal.

Reprezentace automatu

Možnosti reprezentace konečného automatu (způsobu zápisu):

- zápis výčtu jednotlivých prvků pětice konečného automatu
- tabulka
- stavový diagram
- stavový strom

Řešený příklad 4:

Mějme konečný automat $A=(Q,\Sigma,\delta, q_0, F)$

$Q=\{q_0,q_1,q_2,q_3\}$, $\Sigma = \{0,1\}$

$\delta(q_0,0)=q_0$, $\delta(q_0,1)=q_2$

$\delta(q_1,0)=q_1$, $\delta(q_1,1)=q_3$

$\delta(q_2,0)=q_1$, $\delta(q_2,1)=q_3$

$\delta(q_3,0)=q_1$, $\delta(q_3,1)=q_3$

$F=\{q_1,q_2\}$

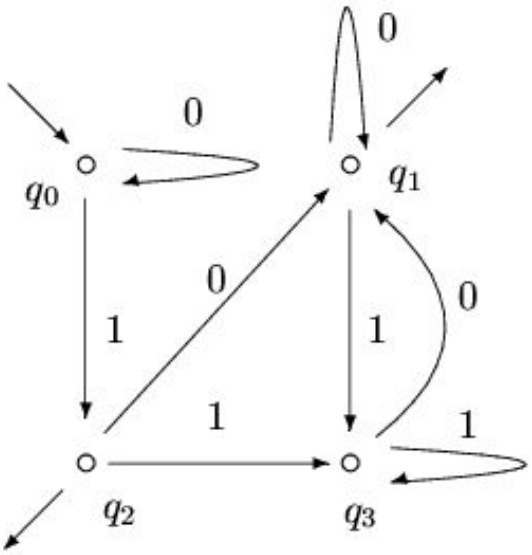
Tabulka

Reprezentace KA A tabulkou:

	$Q \backslash \Sigma$	0	1
→	q_0	q_0	q_2
←	q_1	q_1	q_3
←	q_2	q_1	q_3
	q_3	q_1	q_3

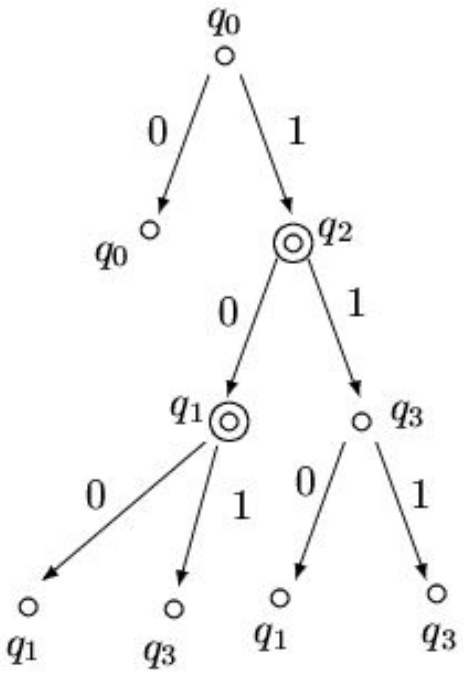
Stavový diagram

Reprezentace KA A stavovým diagramem:



Stavový strom

Reprezentace KA A stavovým stromem:



1.4 Konečný automat (nedeterministický)

Nedeterministický automat

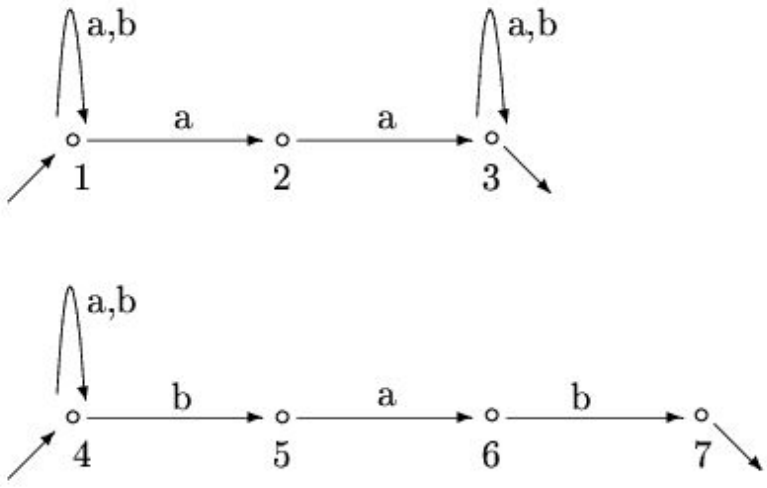
Definice 9: ***Nedeterministickým konečným automatem*** **(NKA) budeme nazývat pětici A =(Q,Σ,δ, I, F) , kde**

- **Q a Σ jsou po řadě neprázdné množiny stavů a vstupních symbolů,**
- **δ:Q×Σ → P(Q) je přechodová funkce (P(Q) je množina všech podmnožin množiny Q).**
- **I ⊆ Q je množina počátečních stavů a F ⊆ Q je množina koncových stavů.**

Poznámka: O dosavadním konečném automatu budeme hovořit jako o *deterministickém*. Všimněte si dvou základních rozdílů:

- NKA už nemusí začínat jen v jednom počátečním stavu, ale může jich mít několik (může si „vybrat“ kde začít)
- přechodová funkce je zobrazením do potenční množiny, tedy množiny všech podmnožin; ze stavu se na symbol může jít nejen do jednoho stavu, ale do libovolného počtu (podmnožiny) nebo i nikam (do prázdné podmnožiny)

Řešený příklad 5:
Příklad nedeterministického konečného automatu:



	$Q\backslash^{\Sigma}$	a	b
→	1	1,2	1
	2	3	–
←	3	3	3
→	4	4	4,5
	5	6	–
	6	–	7
←	7	–	–

Neformálně: NKA přijímá slovo w právě tehdy, když existuje cesta z nějakého z počátečních stavů do nějakého koncového stavu, jejíž ohodnocení je rovno w. V tom je potíž nedeterminismu – vy jako informatici, pro které je základním myšlenkovým principem řešení problému algoritmus, budete mít s pochopením této věty problém. Nedeterminismus totiž vyžaduje pouze, aby řešení existovalo. Cest, přes které se NKA může dostávat z počátečního stavu do koncového může být mnoho. Je to jako pro Vás opět známou hru šachy. Víte, že v ní je obrovské množství variant, jak hra může probíhat. V každém kroku máte několik možností, kterou figurkou a kam táhnout. Přesto nevíte, zda zrovna tato Vaše volba Vám nakonec zajistí vítězství. U automatu jde o něco podobného. „Nezajímá" nás, jak si tuto hru automat rozehraje, ale pokud šance na vítězství existuje (slovo bude rozpoznáno), pak nám to stačí na tvrzení, že vyhrát lze.

Definice 10: Pro NKA A =(Q,Σ,δ, I, F) definujeme *zobecněnou přechodovou funkci* δ^*:P(Q)×Σ* → P(Q) následující rekurzivní definicí:
1. $\delta^*(K,e)=K \ \forall K \in P(Q)$ (tedy $K \subseteq Q$)
2. $\delta^*(K,wa)=\cup_{q \in \delta^*(K,w)}\delta(q,a) \ \forall K \in P(Q)$, $w \in \Sigma^*$, $a \in \Sigma$.
***Slovo* $w \in \Sigma^*$ *je přijímáno* NKA A, jestliže $\delta^*(I,w) \cap F \neq \emptyset$.**

Pozn. Definice této funkce je v tomto případě složitější. Uvědomte si, že výsledkem klasické přechodové funkce je množina stavů, nikoliv jen jeden stav. Proto se musí procházet všechny a je nutná operace sjednocení výsledků.

Jazyk **L(A)** ***rozpoznávaný*** **NKA A je množina všech slov přijímaných automatem A. (L(A)={ $w \in \Sigma^* \mid \delta^*(I,w) \cap F \neq \emptyset$})**

Jazyk **L *je rozpoznatelný* NKA, právě když existuje NKA A takový, že L(A)=L.**

Poznámka: Slovo $w=a_1 a_2 ... a_n$ ($a_i \in \Sigma$) je tedy přijímáno NKA A právě tehdy, když existuje posloupnost $q_1 q_2 ... q_{n+1}$ stavů z Q taková, že $q_1 \in I$, $q_{n+1} \in F$, a pro všechna $i \in \{1,2,...n\}$ je $q_{i+1} \in \delta(q_i,a_i)$. Speciálně $e \in L(A)$ právě, když $I \cap F \neq \emptyset$.

Jazyk rozpoznávaný konečným automatem

Definice 11: Vstupní slovo $w=x_1x_2 \ldots x_m \in \Sigma^+$ je rozpoznáváno nedeterministickým nebo deterministickým konečným automatem A, jestliže existuje posloupnost stavů $q_0,q_{i1},q_{i2},\ldots,q_{im}$ taková, že:
$q_{ik} \in \delta(q_{ik-1},x_k)$
v případě nedeterministického automatu nebo
$q_{ik} = \delta(q_{ik-1},x_k)$
v případě deterministického automatu a $q_{im} \in F$ (stav, ve kterém automat skončil činnost, je koncový stav).

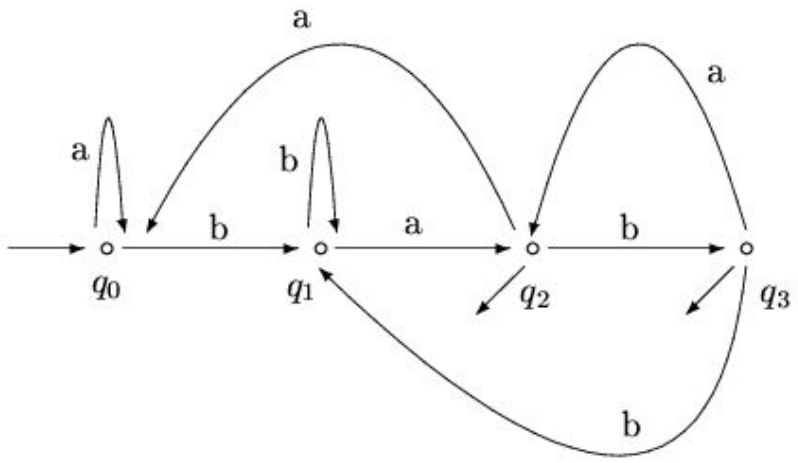
Množina všech slov $w \in \Sigma^*$, jež jsou rozpoznávána (přijata) automatem A, tvoří jazyk rozpoznávaný automatem A. Označujeme ho L(A).

Jazyk rozpoznatelný konečným automatem

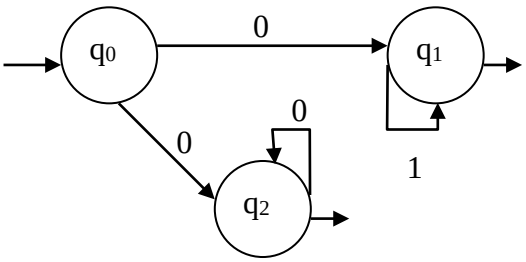
Definice 12: Řekneme, <u>že *jazyk*</u> L (nad abecedou Σ) <u>*je rozpoznatelný konečným automatem*</u>, jestliže existuje konečný automat *A* takový, že L(*A*)=L.

Řešený příklad 6:

L={w ∈ {a,b}*|w končí ba nebo bab}
je jazyk rozpoznatelný konečným automatem (∃A_1; L(A_1)=L):
Automat A_1:



V matematických definicích se nyní pokusíme udělat jasno. Vidíte, že konečný automat, který jsme poznali v jeho intuitivní podobě, je ve formalizované podobě matematickou strukturou – pětic, která ovšem reprezentuje popsané složky automatu, který jsme si představovali jako konkrétní stroj, který byste si třeba sami mohli sestavit doma v dílně. Ta pětice obsahuje stavy (žárovky), abecedu symbolů (písmena na vstupní pásce), počáteční stav (tedy označení toho, od kterého se začíná výpočet stroje, množinu koncových stavů (tedy takových, ve kterých pokud automat skončí, pak čtené slovo je přijato). Nejdůležitější je přechodová funkce, která má jako argument aktuální stav a četný symbol a k němu zobrazení dává nový stav (resp. celou množinu stavů u nedeterministického automatu). Kartézský součin tedy udává co se zobrazuje na co (stavy a symboly na stavy).
V další kapitole se budeme rozdílu mezi nedeterministickým a deterministickým automatem zabývat detailně. Pokuste se do další lekce promyslet, jak by vypadala a chovala se sada „žárovek“ (stavů) u nedeterministického automatu. Tedy například, co by se stalo, kdyby bylo možné se ze stavu q_0 do q_1 a q_2 zároveň (viz následující obrázek popisující přechody automatu).



Pokud jste se dostali až sem, pak jste úspěšně zvládli jednu z nejtěžších úloh celého semestru s kurzem Regulární a bezkontextové jazyky. Podařilo se vám úspěšně pochopit pojem konečného automatu – jeho „žárovkové“ verze i jeho matematické formalizace. Možná se teď ptáte: „Potřebujeme vůbec tu matematickou verzi?“. Odpovídám, že rozhodně ano! Pokud budete chtít s automaty pracovat, převádět je, upravovat, je vždy nutné mít přesnou matematickou formulaci. Přesto se snažte i za těmito formulacemi vidět konkrétní příklady. Vzpomeňte si na známou úlohu z matematiky, kdy máte dva proti sobě jedoucí vlaky, různými rychlostmi a vy máte určit, kde nebo kdy se střetnou. Jistě víte, jak jednoduché je tento problém vyřešit, pokud jej formulujete ve formě rovnic a řešíte naučeným způsobem tyto rovnice jako manipulaci se symboly. Přesně to je i případ této formalizace. Jde o to, abyste problémy z „reálného života“ uměli vyřešit dokazatelnými a přesně formulovanými (algoritmickými postupy). Zamyslete se nad tím a pochopíte, že formalizace není tak samoúčelná, jak se může na první pohled zdát.

V předchozí podkapitole jsme se seznámili s konečným automatem a jeho deterministickou a nedeterministickou verzí. Viděli jsme, že rozdíl mezi nimi spočívá především v možnosti přecházet z jednoho stavu do více stavů (nebo žádného) na symbol u NKA. Každý NKA lze ale pomocí postupu, který se naučíte, převést na deterministický. Pokud jste se zamýšleli nad úkolem na konci kapitoly, dospěli jste zřejmě k poznání, že „žárovky“ (jak jsme velmi zjednodušeně pojmenovali stavy) budou v případě automatu z příkladu svítit současně. Co z toho ale vyplývá? Asi Vás také napadne, že bychom mohli nahradit několik svítících žárovek jednou, která by svítila za určitou rozsvícenou část žárovek – tím bychom mohli odstranit všechny možné kombinace rozsvícených žárovek v původním NKA a získat tím automat, který už nebude obsahovat více svítících žárovek najednou. Prostě pokud máme jako v příkladě aktivní stavy q_1 i q_2 , pak je nahradíme v novém automatu stavem $\{q_1,q_2\}$, který bude tuto situaci reprezentovat a přitom bude novým jedním stavem – tedy je to jakýsi makrostav, který může obsahovat více možných aktivních stavů z výchozího NKA.

1.5 Konstrukce automatů pro zadané jazyky

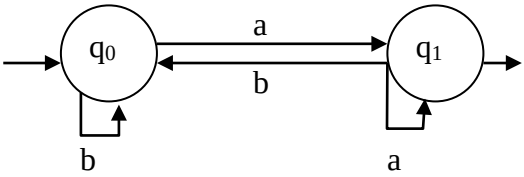
Základním úkolem je pro Vás sestrojení automatu, který rozpoznává jistý jazyk. Samozřejmě jsou i jazyky, pro které to nelze provést, ale nyní se soustřeďme na jednoduché jazyky. Uvidíte, že mnohdy je mnohem jednodušší sestrojit NKA pro zadaný jazyk než DKA. U DKA totiž musíte do všech detailů promyslet, na jaký stav se má přejít na všechny symboly. Zatímco u nedeterministického automatu se můžete soustředit jen na to „co má dělat“ a nemusíte promýšlet, co se má stát, když automat „dělá něco, co dělat nemá“. Podívejme se na příklad, který to vysvětlí:

Řešený příklad 7:

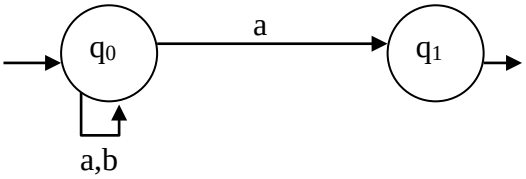
Mějme jazyk L={w ∈ {a,b}*|w končí symbolem a}. Navrhněte konečný automat, který rozpoznává jazyk L.

Výhody NKA

Pokusme se nejprve navrhnout deterministický automat:



A nyní nedeterministický automat:



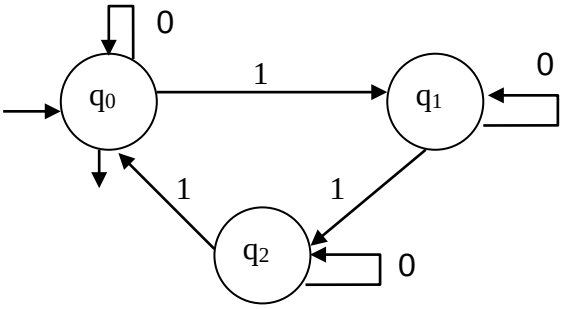
Vidíte, že nedeterministický automat je mnohem jednodušší. Ve stavu q_0 načítá libovolný symbol a na konci slova „uhodne“, že má přejít do q_1, jen pokud je poslední symbol ‚a‘. V tom spočívá síla nedeterminismu, i když z algoritmického hlediska je tato situace nepřípustná. Naučíme se však převádět jakýkoliv nedeterministický automat na deterministický a díky tomu budete schopni navrhovat automaty i pro velmi složité problémy.
Naproti tomu deterministický automat je mnohem složitější. Ve stavu q_0 se cyklí symbol ‚b‘, neboť slovo má končit na ‚a‘. Pokud je nalezen symbol ‚a‘, přejde se do koncového stavu q_1. Jenže co když ještě nejde o poslední symbol? Pak je třeba se vrátit buď se vrátit do q_0, pokud přišel symbol ‚b‘ nebo setrvat v koncovém stavu, přišel-li symbol ‚a‘.

V této kapitole se pokusíme rozebrat některé vybrané úlohy, se kterými se můžete potkat při konstrukci automatu. Budeme je navrhovat jak deterministicky, tak nedeterministicky. V příští podkapitole si ukážeme, jak se dá každý NKA na DKA převést. Když na cvičení pracujeme se studenty prezenčního studia na těchto příkladech, stává se, že někteří studenti jdou „cestou nejmenšího odporu“ a navrhnou si nejprve NKA, který pak tímto postupem převedou. Někteří naopak nad problémem dlouho uvažují a navrhnou rovnou DKA (což je těžší). Někteří to zkoušejí a nejde-li jim to, vydají se jednodušší cestou. Nemohu Vám dát přesný recept, který způsob používat. Záleží to na Vašem způsobu myšlení, trpělivosti – je to spíše psychologická otázka. Mohu Vám říct, že já sám většinou jednoduché příklady napíši přímo jako DKA, ale ve složitějším případě se mi časově lépe osvědčuje navrhnou jednodušeji NKA a ten si převést. Opravdu záleží na konkrétním

případě. Na druhou stranu pokud se pokouším přímo navrhnout DKA, je to velmi dobré mentální cvičení – rozvíjí schopnost prozkoumat možné situace, do kterých se dostane automat a ošetřovat je.

Řešený příklad 8:

Řešme problém, jak sestrojit automat, který rozpoznává jazyk z abecedy {0,1}, přičemž slova z jazyka obsahují počet symbolů 1, který je dělitelný 3 nebo 0. Tedy L = {ε, 0, 00, 000, ..., 010101, 111, 0111,001101, 111111 atd.....}
Takový automat by měl vyjít z počátečního stavu, který by měl být zároveň výstupní (slovo nemusí obsahovat žádný symbol), také nás však nezajímá kolik je ve slově symbolů 0, takže v tomto stavu na symbol 0 můžeme zůstávat. Pokud přijde symbol 1, pak takové slovo už neobsahuje počet dělitelný 3. Proto musíme přejít do stavu jiného (nekoncového). To samé platí, i pokud se vyskytne další 1. Přijde-li však třetí symbol 1, pak je toto slovo opět z jazyka a automat by měl přejít do stavu koncového. Jelikož je však zbytečné toto řešit novým stavem (je to stejná situace jako na začátku), vrátíme se do počátečního stavu a celý průběh se může opakovat do nekonečna. Během celé činnosti (výpočtu) automatu „ignorujeme“ symboly 0, protože jejich počet je nedůležitý. Ignorováním se myslí, že se na něj nemění stav. A nyní jak se tato úvaha prakticky realizuje:



1.6 Algoritmus převodu NKA na DKA

Abychom mohli provádět převody automatů, které vytvoříme jako nedeterministické, máme k dispozici přesný postup, jak kterýkoliv NKA

převést na DKA. Spočívá přesně na principu, který jsme již zmiňovali. Tedy vytváříme z původního automatu podmnožiny stavů, do kterých se lze dostat na určitý symbol. To znamená, že máme-li automat, který se dostane z q_0, jak do q_0 i q_1, pak vytvoříme na tento symbol podmnožinu $\{q_0, q_1\}$. Takto konstruujeme vlastně strom, jenž nám pak reprezentuje nový automat, který je již deterministický. Pozor! Tento stromový algoritmus budeme používat s mírnými obměnami i u dalších převodů, jako je sjednocení či průnik automatů. Věnujte mu proto velkou pozornost.

Algoritmus (stromový):

Stromový algoritmus NKA -> DKA

Na tento převod lze použít podmnožinovou stromovou konstrukci. Máme nedeterministický automat $A_1=(Q_1,\Sigma,\delta_1,I,F_1)$. Chceme sestrojit deterministický automat $A_2=(Q_2,\Sigma,\delta_2,q_0,F_2)$, který bude pro každou dvojici stav-symbol obsahovat právě jeden přechod narozdíl od A_1.

Proces převodu:

1. Vytvoříme množinu, která bude kořenem stromu a bude obsahovat všechny stavy z I (všechny počáteční stavy automatu A_1).
2. Sestrojíme větev s označení symbolu a uzel A_i, pro každý symbol abecedy (tedy uzlů a větví bude tolik, kolik je symbolů abecedy). Obsah každého uzlu vytvoříme tak, že vezmeme všechny stavy z nadřazeného uzlu a zjistíme, kam mohou na daný symbol abecedy přecházet. Výsledný uzel tedy bude opět podmnožina vzniklá sjednocením všech těchto přechodů.
3. Postup z bodu 2. aplikujeme na všechny nově vzniklé uzly, které bude považovat za nadřazený uzel (podobně jako ten z bodu 1.) a vznikat tak bude vždy nový podstrom celého stromu, který vychází z kořene. Výjimkou jsou podmnožiny, které se již někde ve stromě vyskytují. Na tyto již se vyskytující podmnožiny se nebude znova aplikovat bod 2., ale tyto se stanou koncovými uzly (listy stromu).
   Pozn.: Pozor! Podmnožina je jednoznačně určena pouze svými prvky, nikoliv jejich pořadím. např. $\{q_1,q_3,q_5\}$ a $\{q_3,q_1,q_5\}$ jsou stejné množiny!
4. Celý postup vytváření stromu je konečný, protože množina má konečně mnoho prvků (automat je konečný!) a tedy i množina všech podmnožin je konečná. Vytváření uzlů podle bodu 2. skončí, když již nevznikne žádná nová podmnožina.
5. Po vytvoření stromu označíme každou podmnožinu symboly $q_{1'},...,q_{k'}$, které budou stavy automatu A_2. Označení musí být jednoznačné (tedy každá unikátní podmnožina má různé označení než všechny ostatní). Počátečním stavem A_2 bude podmnožina, která je kořenem stromu. Koncovými stavy A_2 budou ty podmnožiny, které obsahují některý z koncových stavů automatu A_1.
6. Sestavíme přechodovou funkci, takže ve stromu zjistíme všechny možné dvojice – nadřazená podmnožina $q_{i'}$, větev se symbolem a, k ní přísluší podřízená podmnožina na dolním konci větve $q_{j'}$. Z těchto údajů sestrojíme $\delta_2(q_{i'},a) = q_{j'}$.

Pozn.: Pokud při tvorbě uzlu podle bodu 2. nenajdeme žádný přechod na žádný ze stavů v nadřazené podmnožině, pak vzniká prázdná množina. Tato prázdná množina je stavem, který ošetřuje chybovou situaci v automatu („zaseknutí“ nedeterministického automatu). Z prázdné množiny se logicky přechází na všechny symboly abecedy opět do prázdné množiny, která nemůže být výstupní.

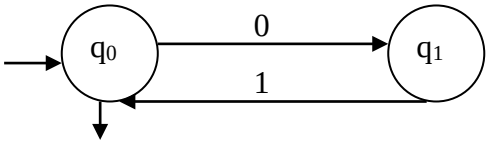
Řešený příklad 9:

Mějme NKA A_1, který rozpoznává jazyk $L = \{(01)^n , n \geq 0\}$.

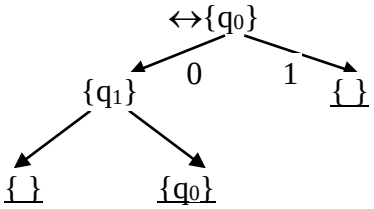
$A_1 = (\{q_1,q_2\}, \{0,1\}, \delta, q_0, \{q_0\})$, kde

$\delta(q_0, 0) = q_1$, $\delta(q_1, 1) = q_0$

Stavový diagram NKA vypadá takto (nemá určeno kam jít na všechny symboly a tudíž není deterministický):



Provedeme převod dle stromového algoritmu:

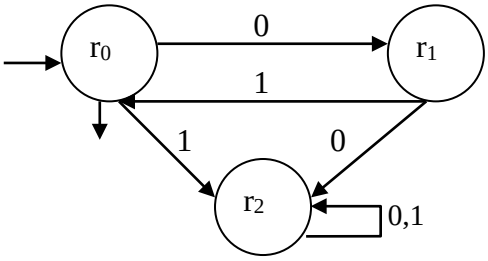


Označíme množiny takto: $\{q_0\} = r_0$, $\{q_1\} = r_1$, $\{\ \} = r_2$.

Pak $q_0 = r_0$, $F_2 = \{r_0\}$,

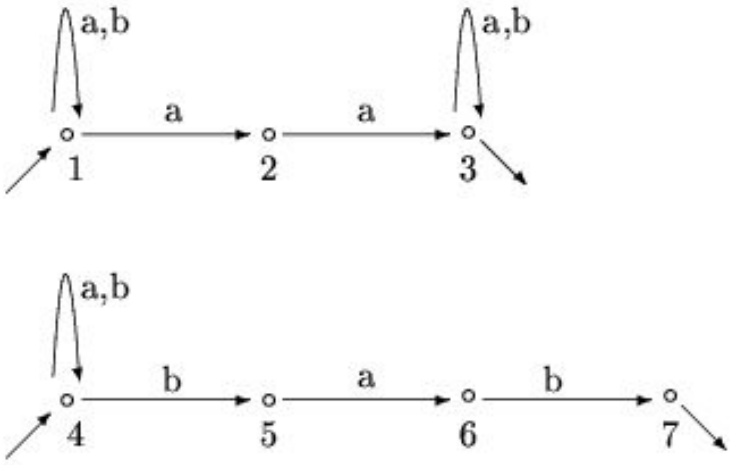
δ_2: $(r_0, 0) = r_1$, $(r_0, 1) = r_2$, $(r_1, 0) = r_2$, $(r_1, 1) = r_0$, $(r_2, 0) = r_2$, $(r_2, 1) = r_2$

Výsledný deterministický automat zapsaný stavovým diagramem:

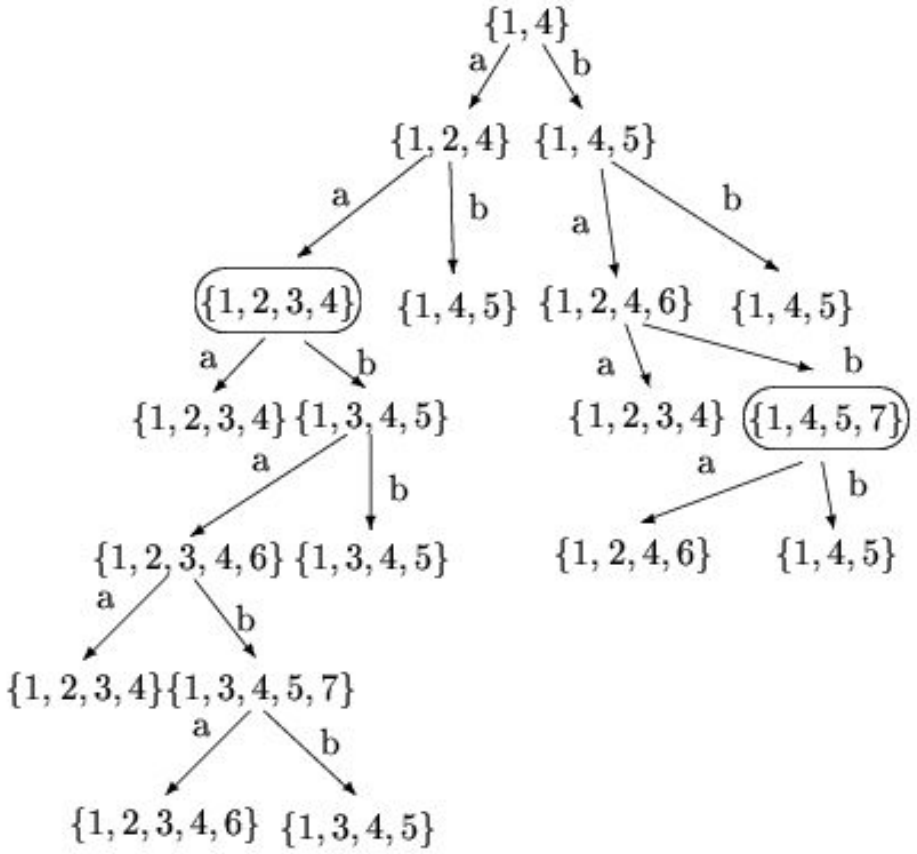


Řešený příklad 10:

Nyní si proberme složitější příklad. Pokusme se navrhnout jednoduše nedeterministicky automat pro jazyk z abecedy {a,b}, který obsahuje slova s výskytem podslova aa nebo slova, která končí na bab. Pro jednoduchost návrhu můžeme automat zapsat jakoby do dvou podautomatů, kde každý z nich rozpoznává slova s aa a slova končící na bab. Náš celkový automat potom má dva vstupní stavy – 1 a 4 a může si nedeterministicky „vybrat“ – „uhádnout“, kterým podautomatem se má vydat. Tento NKA je na následujícím obrázku.



Nyní s pomocí stromového algoritmu převodu na DKA vytvoříme deterministickou verzi.
Postupným procházením podmnožin jsme vytvořili strom DKA. Nyní můžeme tento strom přepsat do libovolné reprezentace KA, např. do tabulky:



Označíme podmnožiny následovně:
A={1,4},B={1,2,4},C={1,4,5},D={1,2,3,4},E={1,2,4,6},F={1,3,4,5},
G={1,4,5,7},H={1,2,3,4,6},I={1,3,4,5,7}
Výstupní stavy jsou ty, které obsahují alespoň jeden výstupní stav z původního NKA: D,F,G,H,I

	$Q\backslash^{\Sigma}$	a	b
→	A	B	C
	B	D	C
	C	E	C
←	D	D	F
	E	D	G
←	F	H	F
←	G	E	C
←	H	D	I
←	I	H	F

Úkol k textu: Přepište tento DKA do formy stavového diagramu a proveďte výpočet podle přechodové funkce pro slova – baaaa, abba, baba, abab a zkontrolujte – zdůvodněte, že automat opravdu rozpoznává stejný jazyk jako NKA.

1.7 Vztah jazyků rozpoznatelných NKA a DKA

V předchozí kapitole jsme poznali, že každý nedeterministický automat lze převést pomocí stromového algoritmu na deterministický. Nicméně, kde bereme tu jistotu, že tímto algoritmem vždy dostaneme automat, který bude rozpoznávat stejný jazyk? Jistě intuitivně tušíme, že převod pouze odhalí kombinace stavů (podmnožiny stavů), do kterých se lze dostat v určité situaci a pak prostě tyto množiny budeme vydávat za stavy nového automatu. Pokud tedy existovala cesta pro rozpoznání slova v původním automatu, pak bude existovat i v sestrojeném (bude vést přes ty podmnožiny, které obsahují stavy přes něž se šlo v původním automatu). To je však pouze tvrzení, které není důkazem. Chceme-li mít jistotu bude nutné toto dokázat.
Zamysleme se však, co to znamená pro jazyky rozpoznatelné NKA a DKA. Je-li automat rozpoznatelný NKA, pak pro něj existuje automat. Jelikož jej ale dokážeme převést na DKA, pak bude rozpoznatelný i DKA. Každý DKA je vlastně speciální případ NKA (neporušuje jeho definici). To znamená, že stejná množina (třída) jazyků by měla být rozpoznatelná, jak DKA, tak NKA. Vidíme tedy, že vztah těchto jazyků je rovnost jejich tříd.
Formulujeme tuto první vlastnost, kterou jsme v rámci studia vyslovili do matematické věty (teorému), jako ekvivalenci dvou vlastností, kterou pak dokážeme.

Konstruktivní důkazy tvrzení v TFJA

Konstruktivní důkaz, který zde uvidíte Vás bude provázet celým studiem. Jako informatici doufám oceníte, že nemusíte jako matematici vymýšlet různé „triky“, jak dokázat jistou vlastnost, ale můžete vyjít z konstrukce pomocí algoritmu, kterou již znáte z příkladů. Vašim úkolem je pak pochopit zobecnění konstrukce a její vlastnosti. V tomto konkrétním případě nám jde o tu vlastnost, zda sestrojený DKA rozpoznává stejný jazyk jako výchozí NKA. Pokusím se Vám tento důkaz maximálně okomentovat, jelikož jde o Váš první důkaz. Možná se ptáte, proč se důkazy vůbec učit, když je již někdo udělal a máme tedy potvrzeno, že tvrzení platí. Důvěřujte mi, že právě důkaz Vám umožní díky své obtížnosti v porovnání s pouhou konstrukcí lépe pochopit podstatu problému. Ostatní důkazy již budu komentovat mnohem méně. Jednak abyste byli nuceni nad nimi opravdu přemýšlet a nikoliv se nechat jen vést mým výkladem a jednak by vzhledem k velkému množství vlastností, které probereme byl rozsah této opory neúnosný. Snažte se tento první důkaz beze zbytku pochopit a bude se Vám u většiny dalších důkazů velmi lehce chápat jejich postup.

Věta 1: Pro libovolný jazyk L jsou následující dvě podmínky ekvivalentní:

1. L je rozpoznatelný (deterministickým) konečným automatem.
2. L je rozpoznatelný nedeterministickým konečným automatem.

Ekvivalence NKA a DKA

Důkaz:
a. 1.⇒2. triviální, neboť DKA je speciálním případem NKA.
b. 2.⇒1.

důkaz konstrukcí DKA:

Nechť L=L(A) pro
NKA A =(Q,Σ,δ, I, F) - *výchozí automat (NKA)*
definujme konečný automat DKA B =(Q′,Σ,δ′,q_0,F′) následovně:
Q′=P(Q); - *stavy jsou všechny možné kombinace stavů výchozího automatu*
δ′: Q′×Σ →Q′, tedy δ′:P(Q)×Σ →P(Q), kde
$\delta'(K,a)=\cup_{q \in K}\delta(q,a)$
∀K ∈ P(Q), a ∈ Σ; - *přechodovou funkci konstruujeme přesně jako ve stromovém algoritmu, tedy z podmnožiny se na symbol a jde do množiny všech možností*
q_0=I; - *počáteční je ta množina, která obsahuje všechny vstupní stavy*
F′={ K ∈ Q′(K ∈ P(Q))|K ∩F ≠ ∅}. – *výstupní je ten stav, který obsahuje alespoň jeden výstupní stav z výchozího automatu*
Dokážeme, že L(B)=L(A). – *čímž dokážeme tvrzení b.*
Pokusíme se prozkoumat postupně všechna slova, která mohou v jazyce být, musíme dokázat nejen že slovo rozpoznané automatem A je rozpoznáno i automatem B, ale i naopak. Jinak by mohlo existovat slovo, které automat A nerozpoznává B rozpoznává! Tím pádem by jazyky nebyly stejné. Proto musíme dokázat nejen implikaci, ale ekvivalenci.

Pro prázdné slovo ε platí: $\varepsilon \in L(B) \Leftrightarrow q_0 \in F' \Leftrightarrow I \in F' \Leftrightarrow I \cap F \neq \emptyset \Leftrightarrow \varepsilon \in L(A)$. – *prázdné slovo je rozpoznáno, pokud je vstupní stav zároveň výstupním, to ale postupně podle naší konstrukce znamená, že je rozpoznáno i B a naopak*
Ověříme, že $w \in L(B) \Leftrightarrow w \in L(A)$ pro neprázdné slovo.

1). Nechť $w=a_1a_2... a_n \in L(B)$ ($n \geq 1$, $a_i \in \Sigma$). Tedy existují $K_1,K_2,... K_{n+1}$ prvky Q' tak, že $K_1=q_0$ (=I), $K_{n+1} \in F'$ (neboli $K_{n+1} \cap F \neq \emptyset$) a $\forall i \in \{ 1,2,... n\}$ je $K_{i+1}=\delta'(K_i,a_i)$ $(=\cup_{q \in Ki}\delta(q,a_i))$.
Všimněme si, že pro libovolné $q \in K_{i+1}$ ($1 \leq i \leq n$) existuje nějaké $q' \in K_i$ takové, že $q \in \delta(q',a_i)$. Proto lze vybrat posloupnost $q_{n+1},q_n,... q_2,q_1$ prvků Q takovou, že $q_i \in K_i$ (pro i=n+1,n,n–1,... 1), také $q_{n+1} \in F$, také $q_{i+1} \in \delta(q_i,a_i)$ a také $q_1 \in I$. To znamená, že $w=a_1a_2... a_n \in L(A)$, tedy $L(B) \subseteq L(A)$. – *existuje tedy posloupnost stavů, kterými DKA B prochází, při rozpoznání nějakého slova až do koncového stavu. Jenže každý takový automat je množina, ve které musí existovat posloupnost stavů automatu A, která skončí v koncovém stavu B. Je-li ale koncový, pak podle naší konstrukce musel obsahovat koncový stav z A.*

2). Nechť $w=a_1a_2... a_n \in L(A)$. Existují stavy $q_1,q_2,... q_n,q_{n+1} \in Q$ tak, že $q_1 \in I$, $q_{n+1} \in F$ a pro $\forall i \in \{ 1,2,... n\}$ $q_{i+1} \in \delta(q_i,a_i)$. Definujme posloupnost $K_1,K_2,... K_{n+1}$ prvků P(Q) tak, že $K_1=I$ a $\forall i \in \{ 1,2,... n\}$ $K_{i+1}=\delta'(K_i,a_i)$ $(=\cup_{q \in Ki}\delta(q,a_i))$.
Zřejmě je $q_1 \in K_1$ a indukcí lze snadno ověřit, že $q_i \in K_i$ pro i=1,2,... n+1.
Tedy $q_{n+1} \in K_{n+1}$, z toho plyne $q_{n+1} \in K_{n+1} \cap F$ neboli $K_{n+1} \cap F \neq \emptyset$.
To znamená $K_{n+1} \in F'$, a tudíž $w=a_1a_2... a_n \in L(B)$. $L(A) \subseteq L(B)$.

Tím jsme dokázali $L(A)=L(B)$.

Projděte si jednotlivé kroky detailně. Uvědomujte si vždy v každém kroku důkazu, o co se snažíte a teprve pak se podívejte, jak se toho dosáhlo. Shrňme důkaz do několika kroků:

1. Je jasné, že musíme dokázat dvě implikace, z níž první tvrdí, že jazyk rozpoznatelný DKA je rozp. i NKA (to je triviální – DKA je v podstatě zároveň i NKA)
2. U druhé implikace potřebujeme nejprve nadefinovat exaktně to, co provádí stromový algoritmus – tedy definovat, jak k automatu NKA najdeme DKA.
3. Po nadefinování potřebujeme dokázat, že takto zkonstruovaný automat rozpoznává všechna slova stejně jako výchozí (a nerozpoznává!)
4. To provedeme tak že prozkoumáme všechna slova, která jazyk může mít – tedy prázdné a neprázdné slovo
5. Případ s prázdným slovem je triviální přepis definic, neprázdné slovo je třeba rozdělit na jednotlivé symboly, které automat postupně podle přechodové funkce převádějí přes mezistavy. Tyto mezistavy musí korespondovat mezi NKA a DKA, jelikož takto je automat podmnožinovou konstrukcí navržen.

Jelikož jsme dokázali toto tvrzení, víme již nyní, že není žádný rozdíl mezi „výpočetní silou“ NKA a DKA. Jinak řečeno to co dokáže NKA dokáže (sice obtížněji) i DKA. Přesto má NKA svůj smysl, jak jsme viděli v kapitole o konstrukci automatů. NKA se navrhuje „pohodlněji“; nemusíte při jeho konstrukci tolik přemýšlet. Máte pak možnost jej na DKA převést. Právě jste pochopili malou, ale důležitou část poznání z oblasti TFJA. Můžeme ji alternativně formulovat jako vlastnost, že ve třídě jazyků rozpoznatelných konečnými automaty nedělí nedeterminismus tuto třídu na podtřídy. Až se budeme zabývat vyšší třídou – bezkontextovými jazyky, zjistíte, že pro tuto třídu to již neplatí.

Převody automatů z nedeterministického na deterministický tvar lze rovněž realizovat automatizovaně s pomocí počítačových programů (jde o poměrně jednoduše implementovatelný postup). Jde nejen o různé jednoduché programy vytvořené studenty (např. GramAut vyvinutý na Ostravské Univerzitě [Hr01]), ale i o profesionální balíčky jako je například LEX známý především uživatelům platformy UNIX.

1.8 Ekvivalentní automaty

Na konec této podkapitoly se seznámíme s pojmem ekvivalentního automatu. Sami asi cítíte, že tento pojem znamená, že automaty jsou v jistém smyslu stejné. Nemusí být sice přímo naprosto stejné, ale rozpoznávají tentýž jazyk. Každý z Vás při řešení příkladů pro sestrojení automatu pro nějaký jazyk může navrhnout jiný automat (někdo se třemi, někdo s 5 stavy atd...) a přesto každý z Vás může navrhnout správný automat. V automatu se mohou vyskytovat například úplně zbytečné stavy, na které se nedá dostat z počátečního (jakési osamocené ostrůvky). Tyto stavy jsou nedosažitelné. A takových stavů může být v automatu libovolný počet, přesto automat stále rozpoznává stejný jazyk!

Definice 13: Dva konečné automaty *A*, *B* nazveme *ekvivalentní*, jestliže rozpoznávají tentýž jazyk, tedy L(*A*)=L(*B*).

Ekvivalentní automaty, nedosažitelné stavy

Definice 14: Stav q konečného automatu $A=(Q,\Sigma,\delta,q_0,F)$ nazveme *dosažitelný* , jestliže existuje $w \in \Sigma^*$ takové, že $\delta^*(q_0,w)=q$. Jinak nazveme q *nedosažitelný* .

Poznámka: Snadno lze sestavit algoritmus, který zjišťuje pro zadaný automat množinu všech jeho dosažitelných stavů (systematické procházení grafu automatu počínaje počátečním stavem). Jelikož je to opravdu triviální, formulujeme jej jako jednoduchý algoritmus a ukážeme si příklad. Všechny nedosažitelné stavy pak můžete z automatu odstranit, aniž byste porušili jazyk, který rozpoznává (tím si automat zjednodušíte).

Algoritmus nalezení dosažitelných stavů:

1. Označte (například kroužkem) počáteční stav.

Nalezení nedosažitelných stavů

2. Pro všechny nově označené stavy projděte jejich sloupce (u jednotlivého symbolu) a pokud stav, na který se má jít, ještě není označen, pak jej označte.
3. Bod 2. provádějte tak dlouho, dokud vznikají nově označené stavy.
4. Všechny označené stavy jsou dosažitelné.

Řešený příklad 11:
Zjistěte, které stavy automatu *A* jsou dosažitelné a které jsou nedosažitelné.
KA *A*:

	$Q\backslash\Sigma$	0	1
→	AX	BY	CZ
	AY	BX	CY
	AZ	BY	CZ
	BX	AY	DZ
	BY	AX	DY
	BZ	AY	DZ
←	CX	BY	CZ
←	CY	BX	CY
	CZ	BY	CZ
	DX	AY	DZ
	DY	AX	DY
	DZ	AY	DZ

Řešení:
Dosažitelné stavy: AX, BY, CZ, DY všechny ostatní stavy automatu jsou nedosažitelné (tj. AY, AZ, BX, BZ, CX,CY, DX, DZ).

Věta 2: **Nechť $A=(Q,\Sigma,\delta,q_0,F)$ je KA a nechť $P \subseteq Q$ je množinou všech dosažitelných stavů automatu *A*. Pak $B=(P,\Sigma,\delta_P,q_0,F\cap P)$, kde δ_P je restrikcí (parcializací) přechodové funkce δ na množinu $P\times\Sigma$, je automat ekvivalentní s *A* a neobsahuje nedosažitelné stavy.**

Důkaz:
Je zřejmé, že stavy, kterými automat projde, než se dostane do nějakého dosažitelného stavu, jsou všechny dosažitelné. Proto platí, že
$\delta^*(q_0,w)=\delta_p{}^*(q_0,w)\ \forall w \in \Sigma^*$. (5)
Množina koncových stavů, do kterých se *A* může dostat z počátečního stavu, je zřejmě rovna množině $F\cap P$. Pro všechna $w \in \Sigma^*$ je proto $\delta^*(q_0,w) \in F \Leftrightarrow \delta_p{}^*(q_0,w) \in F\cap P$, tzn. $L(A)=L(B)$. Ze vztahu (5) také plyne, že *B* neobsahuje nedosažitelné stavy.

1.9 Zobecněný nedeterministický automat

V předchozí kapitole jsme se naučili vytvářet nedeterministické konečné automaty a převádět je na deterministické. Vlastnost nedeterminismu – tedy schopnost automatu přecházet z jedné situace do více různých situací, lze ještě více zobecnit. Zavedeme si automat, který může přecházet mezi stavy i bez čtení jakéhokoliv symbolu (takovým přechodům se říká ε-přechody nebo e-přechody). Takový automat se bude nazývat zobecněný. Uvidíme, že rozpoznávací síla takového automatu je opět stejná a ukážeme si, jak lze i tyto automaty vždy převést na DKA.

Zobecněný nedeterministický automat

Definice 15: ***Zobecněným nedeterministickým konečným automatem*** **(ZNKA) nazveme pětici $A=(Q,\Sigma,\delta,I,F)$, kde Q,Σ,I,F jsou po řadě konečné množiny stavů, vstupních symbolů, počátečních a koncových stavů a δ je přechodová funkce $\delta:Q\times(\Sigma\cup\{\varepsilon\})\rightarrow P(Q)$.**

V definici zobecněné přechodové funkce využijeme označení E(q) množiny všech stavů, do kterých lze ze stavu q přejít jen po ε-přechodech, a jeho zobecnění E(K) pro množinu stavů K.

Množina stavů na ε-přechod

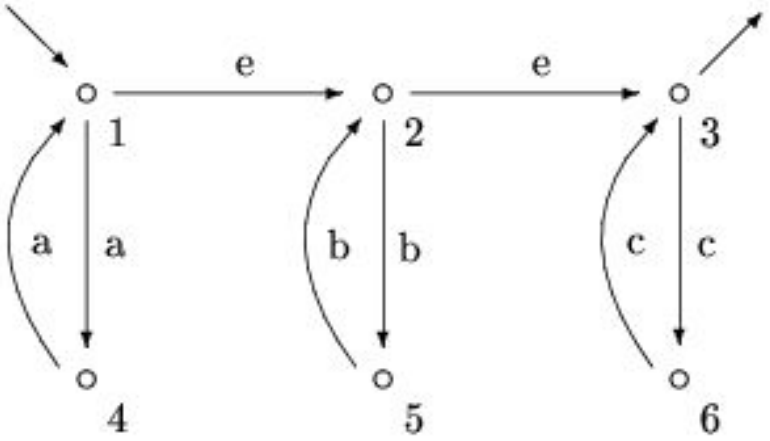
Definice 16: Mějme ZNKA $A=(Q,\Sigma,\delta,I,F)$. Množina E(q), kde $q\in Q$, je nejmenší množinou splňující $q\in E(q)$ a jestliže $q'\in E(q)$ a $q''\in\delta(q',e)$, pak $q''\in E(q)$.
Pro $K\subseteq Q$ definujeme takto: $E(K)=\cup_{q\in K}E(q)$.

Pozn. E funkce (množina pro daný stav) přiřazuje vlastně každému stavu, množinu stavů, na které se lze dostat na ε (e). (tedy bez čtení jakéhokoliv symbolu)

Zobecněná přechodová funkce **$\delta^*:P(Q)\times\Sigma^*\rightarrow P(Q)$ je pak definována následovně: $\delta^*(K,e)=E(K)\ \forall K\subseteq Q$ a $\delta^*(K,wa)=E(\cup_{q\in\delta^*(K,w)}\delta(q,a))\ \forall K\in P(Q), w\in\Sigma^*, a\in\Sigma$.**

Jazyk rozpoznávaný **ZNKA A je $L(A)=\{w\in\Sigma^*|\delta^*(I,w)\cap F\neq\emptyset\}$.**

Mějme zobecněný nedeterministický automat na obrázku.



Tento automat obsahuje ε-přechody (e) mezi stavy 1,2 a 3. Rozpoznává jazyk L = {(aa)*(bb)*(cc)*} – tedy jazyk, v jehož slovech je nejprve sudý počet a, pak sudý počet b a nakonec sudý počet c. Vidíte, že navrhnout tyto automaty je opět výhodné v porovnání s NKA. Jsou mnohem čitelnější, stejně jako byly jednodušší automaty nedeterministické v porovnání s DKA. Tento automat lze transformovat na NKA s pomocí jednoduchého algoritmu (anebo s pomocí stromového algoritmu přímo na DKA).

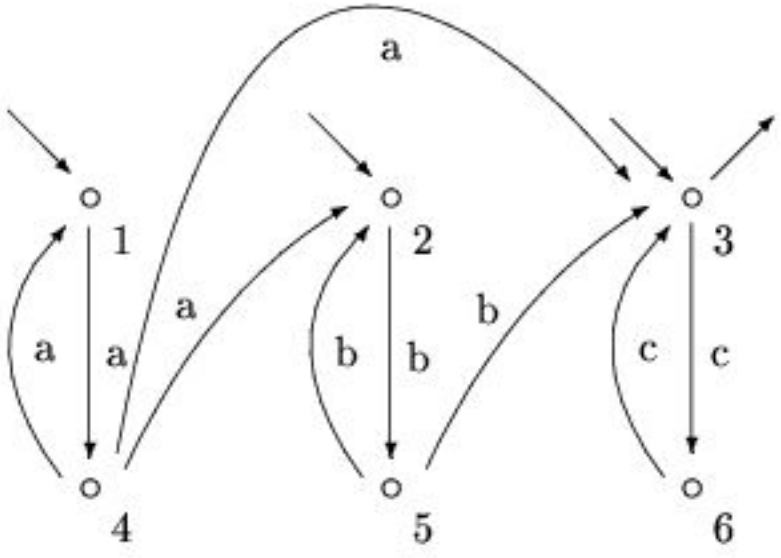
Algoritmus převodu ZNKA na NKA

Algoritmus převodu ZNKA na NKA:

1. Spočteme E funkci dle definice pro všechny stavy
2. Pro každý stav q, q′ a symbol a, kde δ(q′,a) = q, vytvoříme novou přechodovou funkci NKA, tak že do ní zahrneme δ(q′,a) = q a navíc přidáme δ(q′,a) = q′′ pro každý q′′ ∈ E(q). (jinak řečeno pokud na určitý symbol existuje z nějakého stavu 1. do stavu 2. přechod a ze stavu 2. navíc lze přejít ε-přechodem do stavu 3., pak musíme přechodovou funkci obohatit o přechod z 1. do 3.)
3. Vstupní stavy jsou v NKA všechny ze ZNKA a navíc ty, na které ze lze dostat ε-*přechodem*.

Řešený příklad 12:

Pro náš příklad by pak automat sestrojený tímto algoritmem vypadal takto:
E-funkce – E(1)={1,2,3}, E(2)={2,3}, E(3)={3}, E(4)={4}, E(5)={5}, E(6)={6}



Vidíte, že v uvedeném příkladu se například ze stavu 4 jde nejen na 1, ale i na 2, protože z 1 se lze dostat ε přechodem na 2 atd...

Vidíme, že každý ZNKA lze takto převést na NKA. Pokud chceme stejně jako pro vztah NKA-DKA formulovat i vztah ZNKA-NKA-DKA, pak stačí dokázat, že tímto postupem sestrojený automat rozpoznává stejný jazyk.

Ekvivalence ZNKA a NKA

Věta 3: Každý jazyk, který je rozpoznáván nějakým ZNKA, je také rozpoznáván jistým (deterministickým) konečným automatem.

Důkaz:
Nechť $A=(Q,\Sigma,\delta,I,F)$ je ZNKA. Podle věty o vztahu NKA-DKA stačí dokázat, že $L(A)=L(A')$ pro nějaký NKA A'. Ukážeme, že jako A' lze vzít NKA $A'=(Q,\Sigma,\delta',I',F)$, kde $I'=E(I)$ a $\delta'(q,a)=E(\delta(q,a))$ $\forall q \in Q$, $a \in \Sigma$.
Dokážeme: $L(A)=L(A')$.
Pro prázdné slovo e platí:
$e \in L(A) \Leftrightarrow (\delta^*(I,e) \cap F \neq \emptyset) \Leftrightarrow (E(I) \cap F \neq \emptyset) \Leftrightarrow (I' \cap F \neq \emptyset) \Leftrightarrow (e \in L(A'))$
Pro slovo $a_1a_2... a_n$ ($n \geq 1$, $a_i \in \Sigma$):
$a_1a_2... a_n \in L(A) \Leftrightarrow \exists q_0,q_1,q_2,... q_n,q_{n+1} \in Q$ tak, že $q_0 \in I$, $q_1 \in E(q_0)$, $q_{n+1} \in F$ a $\forall j=1,2,... n$ $q_{j+1} \in E(\delta(q_j,a_j)) \Leftrightarrow \exists q_1,q_2,... q_n,q_{n+1}$; $q_1 \in I'$, $q_{n+1} \in F$ a $\forall j=1,2,... n$ platí $q_{j+1} \in \delta'(q_j,a_j) \Leftrightarrow a_1a_2... a_n \in L(A')$.
(důkaz je obdobou důkazu předchozího, který jsme probrali podrobně)

Převod ZNKA přímo na DKA

Pokud chcete převádět ZNKA přímo na DKA je to také možné a také i rychlejší, neboť při něm odpadá nutnost vytvářet ze ZNKA nejprve NKA. Tento převod lze realizovat stromovým algoritmem, který již znáte z minulé kapitoly na převod NKA na DKA. Vyžaduje pouze malou modifikaci.

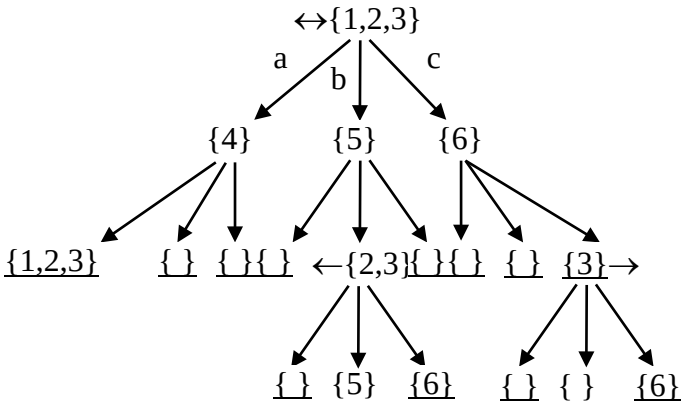
Algoritmus převodu ZNKA na DKA:

- použijeme algoritmus převodu NKA na DKA, pokud při něm vkládáme do množiny (kdykoliv) určitý stav q, pak přidáme do této množiny také všechny stavy z E(q), jinak vše děláme dle tohoto algoritmu

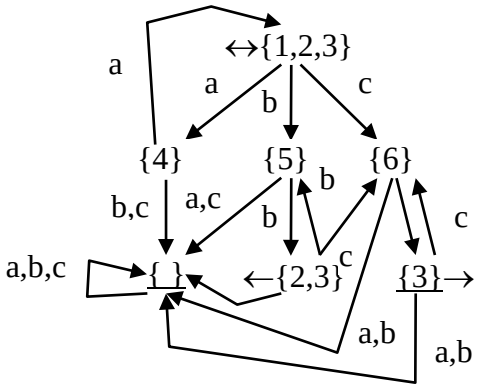
(*tento postup nám zaručí, že přechody na ε budou zahrnuty do přechodové funkce u DKA; Pozor! Nezapomeňte, že i při vytváření kořene stromu se musíte tohoto pravidla držet, tedy do vstupní množiny budou patřit i stavy, na které lze jít na ε)*

Řešený příklad 13:
Převedeme přímo automat na DKA. Množiny E(q) máme určeny, zbývá sestrojit strom.

Algoritmus převodu ZNKA na DKA



Vidíme, že nám vznikl automat deterministický se 7 stavy, který je ekvivalentní se ZNKA. Kořen stromu jsme vytvořili tak, že máme-li do něj vložit stav 1, pak podle modifikovaného algoritmu, do něj vložíme také stavy 2 a 3, neboť patří do E(1). Podobně například ze stavu {5} se lze dostat na stav 2, ale z něj se lze podle E(2) dostat také na stav 3. Přepišme strom na stavový diagram.



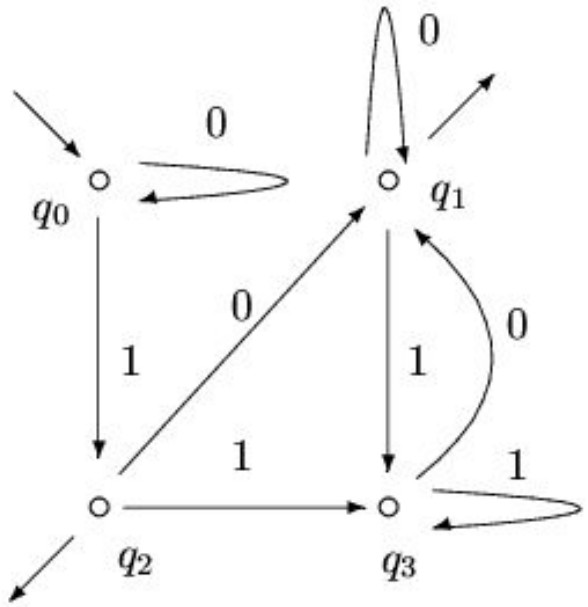
Pokud si zkusíte simulovat činnost automatu z počátečního stavu, zjistíte, že tento automat rozpoznává přesně ten jazyk, který ZNKA.

Kontrolní úkoly:

Úkol 1: Mějme slova u=0010=$0^2 1^1 0^1$, v=11000=$1^2 0^3$, u,v ∈ $\{0,1\}^*$. uv=?, vu=?, uvv=?, u^3=?, v^1=?, v^2=?, |v| = ?, |u| = ?, $|v^2|$ = ?, $|u^3|$ = ?

Úkol 2: Mějme abecedu $\Sigma = \{0,1\}$ a jazyky $L_1=\{0110,10\}$, $L_2=\{e\}$, $L_3=\emptyset$, $L_4=\{0^n1^n; n \geq 0\}$, $L_5=\{a^n01; n \geq 1\}$, $L_6=\{aa,a\}$, $L_7=\{b^{2k}a^kc; k \geq 0\}$. Určete, zda jazyky L_1, L_2, L_3, L_4, L_5, L_6, L_7 jsou jazyky nad abecedou Σ.

Úkol 3: Vezměme konečný automat:



Vyčíslete pomocí zobecněné přechodové funkce do jakého stavu se automat dostane v následujících případech:
$\delta^*(q_0,011001)=?$
$\delta^*(q_2,011001)=?$

Řešení 1:
uv=001011000, vu=110000010, uvv=00101100011000,
u^3=001000100010, v^1=11000, v^2=1100011000, |v| = 5, |u| = 4, $|v^2|$ = 10, $|u^3|$ = 12.

Řešení 2:
$L_1=\{0110,10\}$ je jazyk nad abecedou Σ (dvouprvkový).
$L_2=\{e\}$ je jazyk nad abecedou Σ obsahující pouze prázdné slovo.
$L_3=\emptyset$ je prázdný jazyk nad abecedou Σ.
$L_4=\{0^n1^n; n \geq 0\}$ je jazyk nad abecedou Σ obsahující slova sudé délky, skládající se ze dvou úseků stejné délky, z nichž první obsahuje pouze symboly 0 a druhý pouze symboly 1.
$L_5=\{a^n01; n \geq 1\}$ není jazyk nad abecedou Σ (ale je jazykem nad abecedou {a,0,1}).
$L_6=\{aa,a\}$ není jazyk nad abecedou Σ (ale je jazykem nad abecedou {a}).
$L_7=\{b^{2k}a^kc; k \geq 0\}$ není jazyk nad abecedou Σ (ale je jazykem nad abecedou {a,b,c}).

Řešení 3:

$\delta^*(q_0,011001) = \delta(\delta^*(q_0,01100),1) = \delta(\delta(\delta^*(q_0,0110),0),1)=$
$=\delta(\delta(\delta(\delta^*(q_0,011),0),0),1) = \delta(\delta(\delta(\delta(\delta^*(q_0,01),1),0),0),1)=$
$=\delta(\delta(\delta(\delta(\delta(\delta^*(q_0,0),1),1),0),0),1) = \delta(\delta(\delta(\delta(\delta(\delta(\delta^*(q_0,e),0),1),1),0),0),1)=$
$=\delta(\delta(\delta(\delta(\delta(q_0,0),1),1),0),0),1) = \delta(\delta(\delta(\delta(q_0,1),1),0),0),1)=$
$=\delta(\delta(\delta(\delta(q_2,1),0),0),1) = \delta(\delta(\delta(q_3,0),0),1)=$
$=\delta(\delta(q_1,0),1) = \delta(q_1,1)=q_3$
$\delta^*(q_2,011001) = \delta(\delta^*(q_2,01100),1) = \delta(\delta(\delta^*(q_2,0110),0),1)=$
$=\delta(\delta(\delta(\delta^*(q_2,011),0),0),1) = \delta(\delta(\delta(\delta(\delta^*(q_2,01),1),0),0),1)=$
$=\delta(\delta(\delta(\delta(\delta(\delta^*(q_2,0),1),1),0),0),1) = \delta(\delta(\delta(\delta(\delta(\delta(\delta^*(q_2,e),0),1),1),0),0),1)=$
$=\delta(\delta(\delta(\delta(\delta(q_2,0),1),1),0),0),1) = \delta(\delta(\delta(\delta(q_1,1),1),0),0),1)=$
$=\delta(\delta(\delta(\delta(q_3,1),0),0),1) = \delta(\delta(\delta(q_3,0),0),1)=$
$=\delta(\delta(q_1,0),1) = \delta(q_1,1)=q_3$

Úkoly a otázky k textu:

1. Je deterministický konečný automat speciálním případem nedeterministického nebo naopak?
2. Můžete pro konečný jazyk napsat vždy konečný automat?
3. Sestrojte a zapište všemi způsoby, které jste se naučili konečný automat reprezentující automat na jízdenky s následujícími vlastnostmi: přijímá mince v hodnotě 1 Kč, 2 Kč, 5 Kč, vydává jízdenky, buď pro děti za 3 Kč nebo za 7 Kč pro dospělé
4. Sestrojte DKA rozpoznávající jazyk $L=\{w \in \{a,b\}^* | w$ končí symbolem ‚a' nebo obsahuje ‚bab'$\}$.

Nejdůležitější probrané pojmy:

- JAZYK, GRAMATIKA, AUTOMAT
- slovo, abeceda, zřetězení, uzávěry množiny, formální jazyk
- deterministický konečný automat
- nedeterministický konečný automat
- přechodová funkce, zobecněná přechodová funkce
- jazyk rozpoznávaný a jazyk rozpoznatelný konečným automatem
- reprezentace KA: výčet matematické struktury, tabulka, stavový diagram, strom
- stromový algoritmus převodu NKA na DKA
- ekvivalence jazyků rozpoznatelných NKA a DKA + důkaz

2 Uzávěrové vlastnosti a redukce KA

V této kapitole se dozvíte:

- Jaké vlastnosti má třída jazyků rozpoznatelných konečnými automaty.
- Vůči kterým operacím s jazyky je tato třída uzavřená.
- Algoritmus redukce KA

Po jejím prostudování byste měli být schopni:

- Provádět základní operace s jazyky.
- Konstruovat KA k operacím s jazyky.
- Dokázat korektnost těchto konstrukcí.
- Odhalovat ekvivalentní stavy automatu a konstrukci redukovaného (minimalizovaného) automatu.
- Normovat konečné automaty.

Klíčová slova této kapitoly:

Uzávěrová operace, množinové operace, jazykové operace, podílový automat, redukt, normovaný redukt.

Průvodce studiem

Studium této kapitoly je vyžaduje především dobrou schopnost chápat algoritmy a datové struktury. To však není jediným problémem kapitoly, navíc se zde ověřují vlastnosti těchto algoritmických konstrukcí – provádějí se důkazy. Nalezení důkazu – v našem případě naštěstí konstruktivního typu (tedy pouze ověřujeme, zda algoritmus vede ke korektním výsledkům) – bývá pro studenty informatiky nejobtížnější dovednost.
Na studium této části si vyhraďte alespoň 10 hodin. Rozdělte si studium na dobré pochopení postupů-algoritmů a teprve po ověření pochopení na příkladech se pusťte i do složitějších partií věnovaných důkazům.

V minulých kapitolách jsme se naučili, jak sestrojovat automaty k zadaným jazykům v jejich různých modifikacích. Viděli jste, že někdy je rozumné si rozdělit problém sestrojení automatu na dva jednodušší automaty (resp. jeden nedeterministický automat rozdělený do dvou oddělených částí). Každý z automatů pak řešil podproblém. Podívejte se zpět do textu na příklad NKA. V tomto příkladu máme zadán automat, který rozpoznává jazyk

$L=\{w \in \{a,b\}^* | w$ obsahuje aa nebo končí na bab$\}$

Aplikace uzávěrových operací

Již samotné zadání v sobě obsahuje toto rozdělení problému. Sestrojíme dva samostatné úseky automatu – jeden pro slova obsahující v sobě řetězec „aa“ a druhý pro slova končící na „bab“. Z předchozí studia víte, že nedeterministický

automat může být takto navržen (může mít více vstupů). Nicméně na celý problém se můžeme podívat ještě z jednoho hlediska. Co kdybychom považovali tyto úseky za dva různé automaty. Pak bychom hledali jejich jakési „spojení“. Používáme-li však matematického aparátu, můžeme to nazvat přesně. Jde o nalezení automatu, který bude rozpoznávat sjednocení jazyků obou jednodušších automatů. A právě o to nám v této kapitole půjde. Definuje, co je to sjednocení, průnik a další operace nad jazyky. A dále se budeme zabývat tím, jak tyto „uzávěrové“ operace můžeme simulovat pomocí automatů (jak sestrojovat takové automaty). Uvidíme, že omezíme-li se na třídu jazyků rozpoznatelných KA, pak tyto operace uzavírají tuto třídu jazyků – jinak řečeno: Pokud vezmeme libovolné jazyky z této třídy a provedeme s nimi jakoukoliv z těchto operací, dostaneme opět jazyk z této třídy.

Poznámka: Symbolem F budeme označovat *třídu všech jazyků rozpoznatelných KA*. Pro jazyky L_1,L_2 nad abecedou Σ ($L_1,L_2 \subseteq \Sigma^*$), má smysl uvažovat množinové operace sjednocení ($L_1 \cup L_2$), průniku ($L_1 \cap L_2$) a množinového rozdílu ($L_1 - L_2$). Doplňkem jazyka L_1 rozumíme rozdíl $\Sigma^* - L_1$, když je Σ z kontextu jasné, píšeme $-L_1$.

Již na počátku textu jsem avizoval, že se v této opoře bude vycházet z Vašich znalostí teorie množin, základů matematiky. Proto nebudeme pojmy průniku, sjednocení a rozdílu definovat, pouze si ukážeme příklady. Věřím, že tak jednoduché pojmy bez problému chápete – vždyť při práci s jazyky nejde o nic jiného než o práci s jakýmikoliv množinami, které již znáte ze středoškolské matematiky. Vzpomeňte si na sjednocení, průniky například intervalů!

Řešený příklad 14:
Vezměme si například jazyky
$L_1=\{w \in \{0,1\}^*; w = 0^n1, \text{ kde } n \geq 0\} = \{1, 01, 001, ...\}$
$L_2=\{w \in \{0,1\}^*; w = 01^n, \text{ kde } n \geq 0\} = \{0, 01, 011, ...\}$

Pak $L_1 \cup L_2 = \{0,1,01,011,0111,...,001,0001,....\}$,
$L_1 \cap L_2 = \{01\}$, $L_1 - L_2 = \{1,001,...\}$, $L_2 - L_1 = \{0,011,...\}$

Tímto přístupem můžeme například poměrně složitý jazyk rozdělit na podjazyky:

Řešený příklad 15:

$L=\{w \in \{0,1\}^*; w$ obsahuje sudý počet nul a každá jednička je bezprostředně následována alespoň jednou nulou $\}$
L můžeme vyjádřit množinovými operacemi nad třemi jednodušeji charakterizovanými jazyky:
$L_1=\{w \in \{0,1\}^*; w$ obsahuje sudý počet nul $\}$
$L_2=\{w \in \{0,1\}^*; w$ obsahuje podslovo 11$\}$
$L_3=\{w \in \{0,1\}^*; w$ končí jedničkou $\}$
takto: $L=L_1-(L_2 \cup L_3)$.

V následujících podkapitolách budeme ukazovat algoritmy pro sestrojení různých uzávěrových operací. Půjde o úlohu, jak pro automaty $A_1=(Q_1,\Sigma,\delta_1,q_1,F_1)$, kde $L_1=L(A_1)$ a $A_2=(Q_2,\Sigma,\delta_2,q_2,F_2)$, kde $L_2=L(A_2)$ sestrojit automat A, který rozpoznává $L = L_1 \bullet L_2$, kde $\bullet$ je příslušná operace.

2.1 Sjednocení, průnik, doplněk, rozdíl, zrcadlový obraz

Co je sjednocení dvou automatů? Lze říct, že je to automat, který rozpoznává slova z jazyka L_1 nebo L_2 tedy rozpoznává slova z obou jazyků. Pokud si uvědomíte, jak jsme sestrojovali k NKA jeho deterministickou verzi, pak Vás možná i napadne, jak by se dal sestrojit automat pro sjednocení....
Je to jednoduché, stačí použít stromový algoritmus, s tím, že logicky budeme považovat vstupy automatů A_1, A_2 za vstupy zároveň. Jelikož pak má automat přijímat slova z obou automatů, výstupní stavy (množiny) budou ty, který obsahují výstupní stav ze kteréhokoliv z obou automatů. Tím si zajistíme, že všechna slova budou přijata.

Algoritmus pro sestrojení $A_1 \cup A_2$ (stromový):

1. Algoritmus pracuje jako stromový algoritmus s níže uvedenými modifikacemi.
2. Kořen stromu vytvoříme jako $\{q_1, q_2\}$ *(tedy vstupní množina je tvořena počátečními stavy obou automatů)*
3. Výstupní množina je ta, která obsahuje libovolný výstupní stav z A_1 nebo A_2.

Algoritmus sjednocení

Pokud jde o průnik, pak asi tušíte, že algoritmus bude velice podobný jako pro sjednocení. V tomto případě však očekáváme automat, který bude rozpoznávat jen ta slova, která rozpoznává automat A_1 a zároveň A_2. To tedy znamená, že celý algoritmus bude totožný až na výstupy. Výstupním stavem (množinou) bude jen ta, která obsahuje kombinaci s alespoň jedním výstupním stavem z A_1 a alespoň jedním z A_2.

Algoritmus pro sestrojení $A_1 \cap A_2$ (stromový):

1. Algoritmus pracuje jako stromový algoritmus s níže uvedenými modifikacemi.
2. Kořen stromu vytvoříme jako $\{q_1, q_2\}$ *(tedy vstupní množina je tvořena počátečními stavy obou automatů)*
3. Výstupní množina je ta, která obsahuje alespoň jeden výstupní stav z A_1 a zároveň z A_2.

Algoritmus průniku

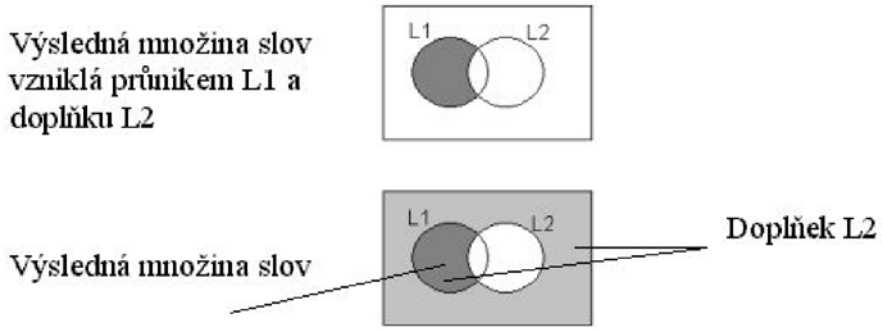
Doplněk jazyka obsahuje právě ta slova, která původní jazyk neobsahuje. Jeho sestrojení je tedy nejjednodušší operací. Stačí zaměnit stavy, které jsou výstupní na obyčejné a naopak. Pak budou slova automatem původním rozpoznávána v sestrojeném automatu nerozpoznávána a naopak:

Algoritmus doplňku

Algoritmus pro sestrojení -A_1 = A pro DKA:

1. Automat A, bude kopií A_1, s výjimkou F (množiny koncových stavů).
2. F = Q – F_1. *(výstupy jsou opačné stavy)*

Rozdíl lze sestrojit s pomocí již definovaných algoritmů. Stačí si opět uvědomit, jaké vlastnosti má množinový rozdíl, jak jej znáte ze střední školy:



L1-L2 se dá vytvořit průnikem L1 a doplňku L2. L1 – L2 = L1∩(-L2)

Algoritmus rozdílu a zrcadlového obrazu

Algoritmus pro sestrojení A_1 - A_2:

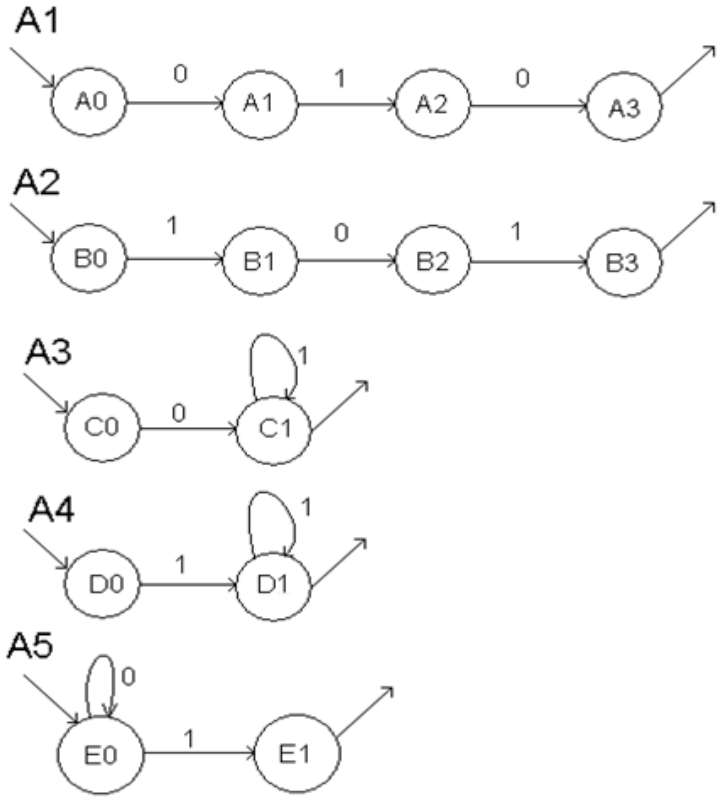
1. A sestrojíme postupně jako $A_1 \cap$ -A_2.

Zrcadlový obraz (značíme L^R) jazyka jsou všechna jeho slova, braná pozpátku. Tedy např.
L_1={w ∈ {0,1}*;w = 0^n1, kde n ≥ 0} = {1, 01, 001, ...}
L_1^R={w ∈ {0,1}*;w = 10^n, kde n ≥ 0} = {1, 10, 100, ...}

Sestrojení zrcadlového obrazu je také poměrně jednoduchou operací. Pokud mají být rozpoznávána slova obráceně, pak stačí, pokud automat bude také obráceně pracovat. Tedy laicky řečeno, všechny přechody a vstupy a výstupy budou obráceně.

Algoritmus pro sestrojení NKA A = A_1^R:

1. Sestrojíme A tak, že pro $\delta_1(p_1,a) = p_2$ definujeme $\delta(p_2,a) = p_1$ a zároveň I = F, F = q_1

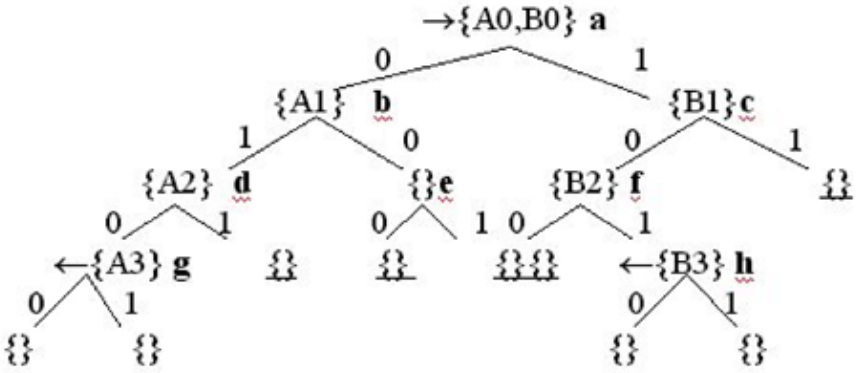


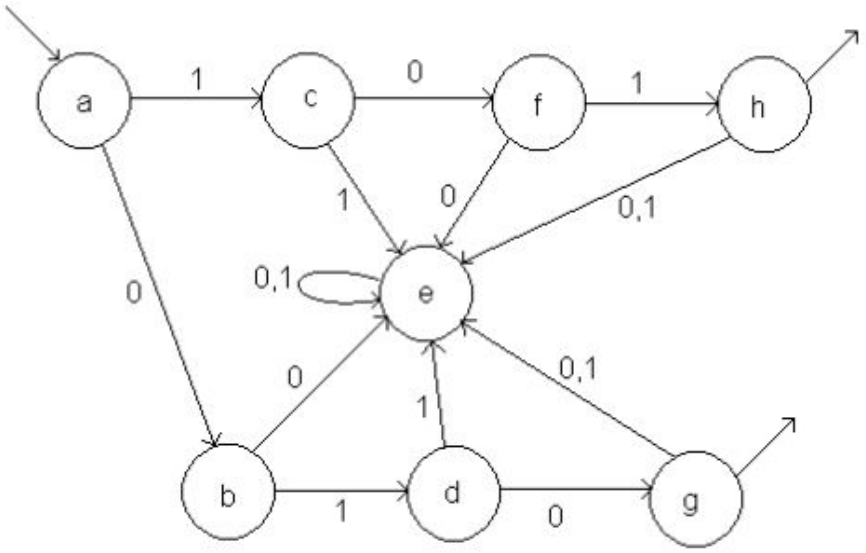
Nyní si ukážeme příklady použití algoritmů.
Jazyky rozpoznávané těmito automaty:
$L_1=\{010\}$, $L_2=\{101\}$, $L_3=\{01^n, n \geq 0\}$, $L_4=\{11^n, n \geq 0\}$, $L_5=\{0^n1, n \geq 0\}$.

Řešený příklad 16:

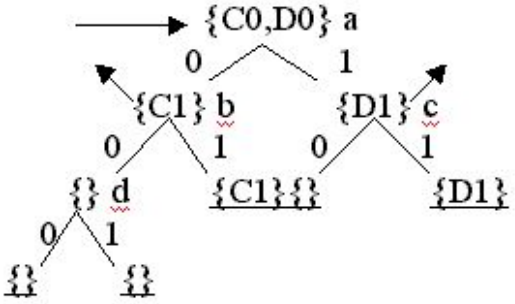
Příklad pro sjednocení pomocí stromového algoritmu:

L1∪L2: vstupním stavem bude množina se vstupy obou automatů a výstupem stav, který obsahuje libovolný výstupní stavu automatu A1 nebo A2





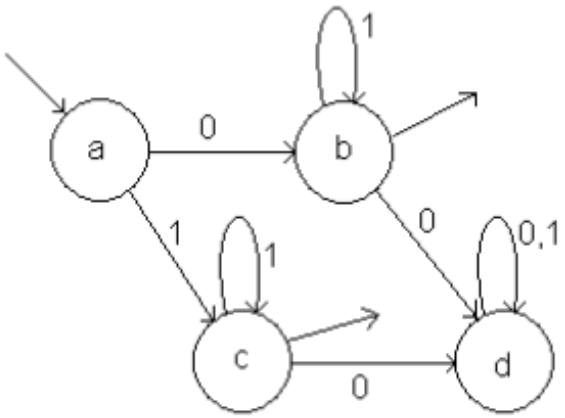
L3∪L4:



Řešený příklad 17:
L1∩L2:

Tvoří se podobně jako sjednocení, ale výstupní stav je tvořen stavem obsahujícím současně výstupní stavy obou automatů.

V příkladu sjednocení automatu A1 a A2 není žádný stav, který by obsahoval současně výstupní stavy obou automatů. Pokud A1 generuje slova 101 a A2 010 průnik těchto automatů je prázdný jazyk. Z toho důvodu L1∩L2=∅.

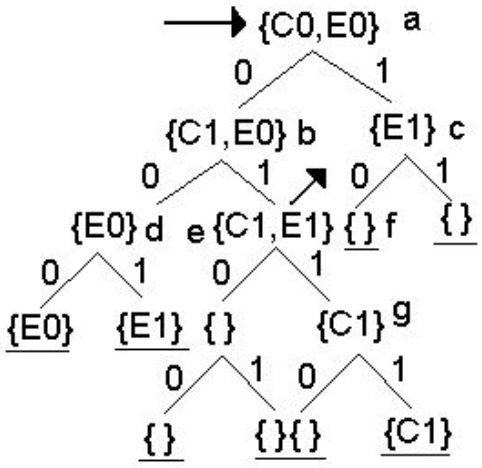


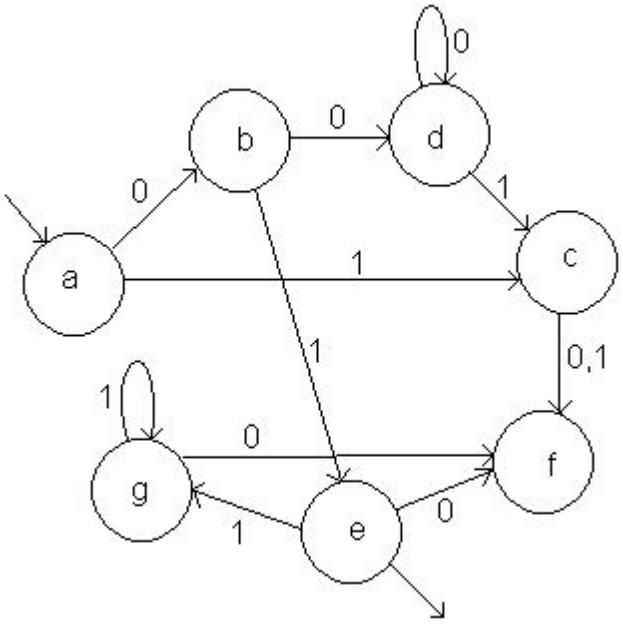
L3∩L4:

Podobně by na tom byl průnik automatů A3 a A4. Pokud máme slova 01, 011, 0111… automatu A1 a slova 1, 11, 111, 1111… automatu A4 z toho důvodu L3∩L4=∅.

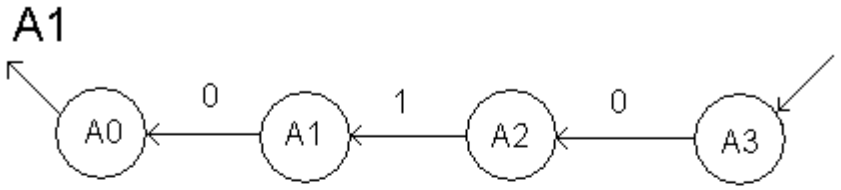
L3∩L5:

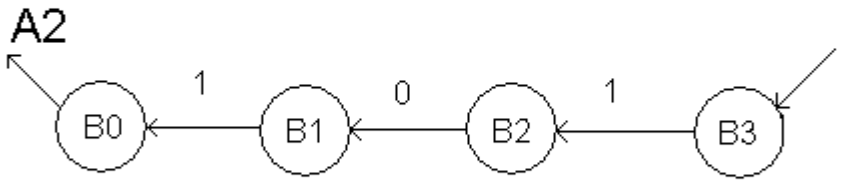
Neprázdný průnik ale tvoří automaty A3 a A5. Automat A3 generuje slova 0, 01, 011, 0111… a automat A5 slova 1, 01, 001, 0001… . Jejich průnikem je slovo 01. L3∩L5={01}.





Řešený příklad 18:
$L1^R$ $L2^R$ Zrcadlový obraz slov generovaných automatem se vytvoří tak, že se automat zkonstruuje pozpátku. Změní se směr šipek. Automaty A1, A2 generují stejná slova jak pro L1,L2 tak i $L1^R$,$L2^R$.





2.2 Uzávěrové vlastnosti jazyků rozpoznatelných KA

Uzávěrové vlastnosti -sjednocení, průnik,

Nyní si formulujeme tvrzení, která vycházejí z praktických dopadů algoritmů, které jsme si uvedli. Formulujeme tvrzení, že třída jazyků rozpoznatelných konečnými automaty je uzavřená na množinové operace. Důkazy vycházejí z postupů, které pak můžete prakticky vyzkoušet na řešených příkladech v dalších kapitolách.

Věta 4: **Pro libovolné jazyky $L_1,L_2 \subseteq \Sigma^*$ platí: jestliže $L_1,L_2 \in F$, potom také $L_1 \cup L_2 \in F$, $L_1 \cap L_2 \in F$, $-L_1 \in F$. (Neboli: Třída jazyků rozpoznatelných KA je uzavřena vůči operacím průniku, sjednocení a doplňku.)**

Důkaz:
$L_1=L(A_1)$, $L_2=L(A_2)$, kde $A_1=(Q_1,\Sigma,\delta_1,q_1,F_1)$, $A_2=(Q_2,\Sigma,\delta_2,q_2,F_2)$ jsou KA.

1. Průnik
Sestrojíme KA $A=(Q,\Sigma,\delta,q_0,F)$ takový, že $L(A) = L_1 \cap L_2$.
Předpokládejme: $Q_1 \cap Q_2 = \emptyset$. Položme $Q=Q_1 \times Q_2$, $q_0 = (q_1,q_2)$, $F=F_1 \times F_2$ a δ je dána následovně.
Pro lib. $p_1 \in Q_1$, $p_2 \in Q_2$, $a \in \Sigma$ je $\delta((p_1,p_2),a) = (\delta_1(p_1,a),\delta_2(p_2,a))$.
Indukcí lze ukázat, že
$\delta^*((q_1,q_2),w)=(\delta_1^*(q_1,w),\delta_2^*(q_2,w))$ pro lib. $w \in \Sigma^*$.
$w \in L_1 \cap L_2 \Leftrightarrow w \in L_1 \wedge w \in L_2 \Leftrightarrow (\delta_1^*(q_1,w) \in F_1) \wedge (\delta_2^*(q_2,w) \in F_2) \Leftrightarrow (\delta_1^*(q_1,w),\delta_2^*(q_2,w)) \in F_1 \times F_2 \Leftrightarrow \delta^*((q_1,q_2),w) \in F \Leftrightarrow w \in L(A)$.

2. Doplněk
Je zřejmé, že automat $(Q_1,\Sigma,\delta_1,q_1,Q_1-F_1)$ rozpoznává jazyk $\Sigma^* - L_1$.

3. Sjednocení
a) Lze postupovat stejně jako u průniku jen $F=(F_1 \times Q_2) \cup (Q_1 \times F_2)$.
b) Předpoklad $L_1,L_2 \in F$, podle 2. i $-L_1,-L_2 \in F$, podle 1. i $-L_1 \cap -L_2 \in F$, podle 2. dále $-(-L_1 \cap -L_2) \in F$. Ovšem podle jednoho z de Morganových zákonů $L_1 \cup L_2 = -(-L_1 \cap -L_2)$, takže $L_1 \cup L_2 \in F$.
c) Můžeme předpokládat, že $L_1=L(A_1)$, $L_2=L(A_2)$, kde $A_1=(Q_1,\Sigma,\delta_1,I_1,F_1)$, $A_2=(Q_2,\Sigma,\delta_2,I_2,F_2)$ jsou NKA. Pak sestrojíme $A=(Q,\Sigma,\delta,I,F)$ následovně: předpokládejme: $Q_1 \cap Q_2 = \emptyset$. Položíme $Q=Q_1 \cup Q_2$, $I=I_1 \cup I_2$, $F=F_1 \cup F_2$ a $\delta(q,a)=\delta_1(q,a)$ pro lib. $q \in Q_1$, $a \in \Sigma$, $\delta(q,a)=\delta_2(q,a)$ pro lib. $q \in Q_2$, $a \in \Sigma$. Snadno lze ukázat, že $L(A)=L_1 \cup L_2$.

-rozdíl

Věta 5:
Důsledek. předchozí věty: **Nechť $L_1,L_2 \subseteq \Sigma^*$. Jestliže $L_1,L_2 \in F$, pak také $L_1 - L_2 \in F$.**

Důkaz: $L_1 - L_2 = L_1 \cap (\Sigma^* - L_2)$

Dalším důležitým tvrzením je, že s pomocí množinových operací dokážeme zjistit, že dva automaty rozpoznávají stejný jazyk. Lze to provést opět úvahou nad významem tohoto tvrzení. Pokud odečteme $L(A_1) - L(A_2)$ a $L(A_2) - L(A_1)$ a ani v jednom případě nezůstane žádné slovo, pak jazyky nutně musí být stejné (ověřte si to na vlastním příkladě!). Pak už jen stačí zjistit, že sestrojený automat se nemůže nijak dostat do koncového stavu.

Věta 6: Existuje algoritmus, který pro libovolné dva konečné automaty A_1,A_2 rozhodne, zda $L(A_1) = L(A_2)$.

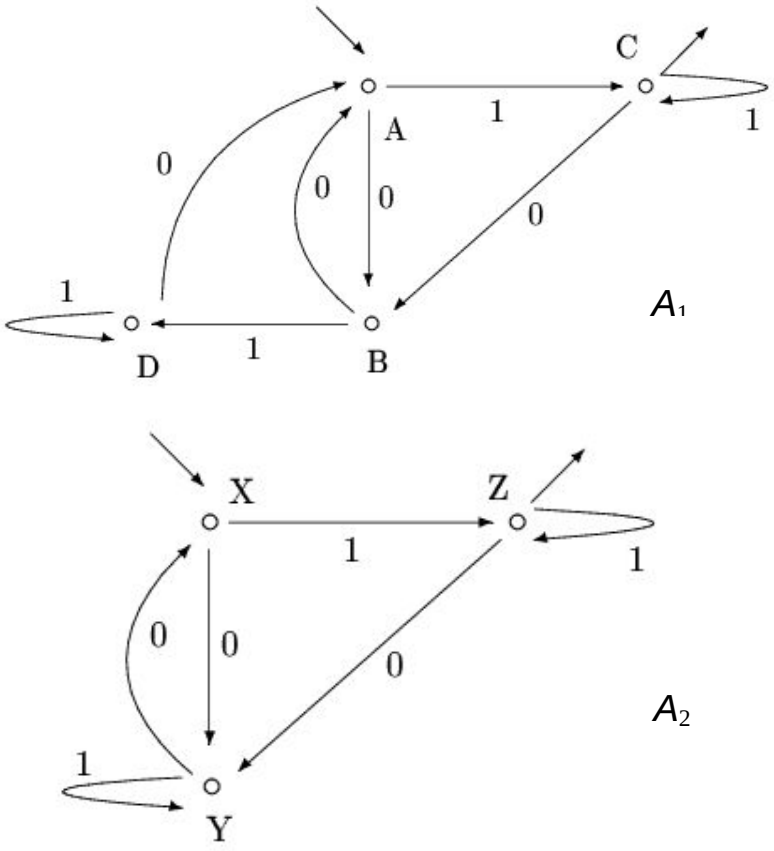
Důkaz:

Uvědomme si, že existuje algoritmus, který pro lib. KA A určí, zda $L(A)=\emptyset$. Ovšem $L(A_1)=L(A_2)$ platí právě tehdy, když $(L(A_1)-L(A_2))\cup(L(A_2)-L(A_1))=\emptyset$. Podle důkazu věty o uzavřenosti F lze zkonstruovat automat A takový, že $L(A)=(L(A_1)-L(A_2))\cup(L(A_2)-L(A_1))$. A pak stačí jen ověřit, zda $L(A)$ je prázdná množina.

Rozhodnutelnost ekvivalence automatů

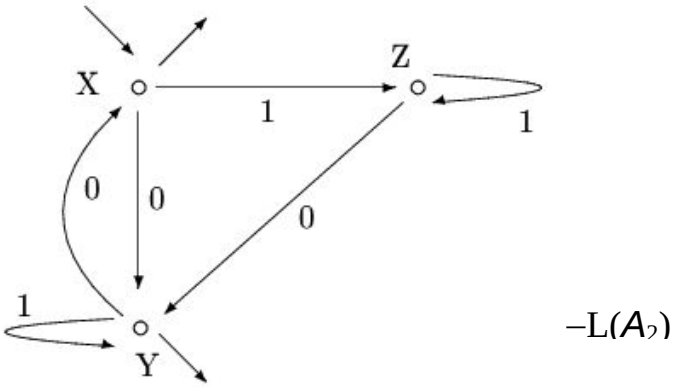
Kontrolní úkol:

Ukažte, že automaty A_1, A_2 na obrázku rozpoznávají tentýž jazyk.

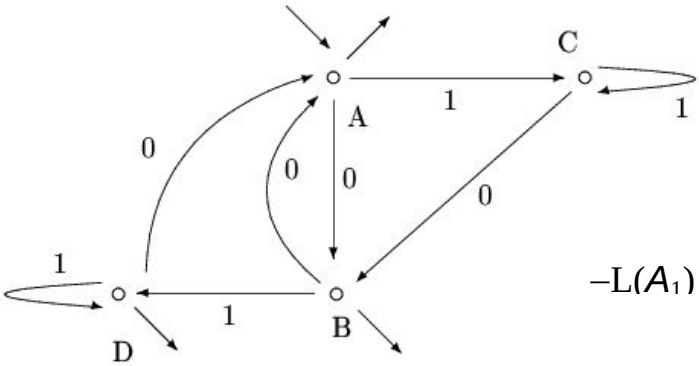


Řešení:

Pomocí postupu uvedeného v důkazu věty.
Jazyk $-L(A_2)$ a jazyk $-L(A_1)$ rozpoznávají KA na obrázku.



$-L(A_2)$



$-L(A_1)$

KA rozpoznávající $L(A_1)-L(A_2)$: (místo (i,j) používáme zápis ij)

	$Q\backslash\Sigma$	0	1
→	AX	BY	CZ
	AY	BX	CY
	AZ	BY	CZ
	BX	AY	DZ
	BY	AX	DY
	BZ	AY	DZ
←	CX	BY	CZ
←	CY	BX	CY
	CZ	BY	CZ
	DX	AY	DZ
	DY	AX	DY
	DZ	AY	DZ

Tento automat má dva koncové stavy, ale oba jsou nedosažitelné, tzn. $L(A_1)-L(A_2)=\emptyset$.
KA rozpoznávající $L(A_2)-L(A_1)$:

	$Q\backslash\Sigma$	0	1
→	AX	BY	CZ

	AY	BX	CY
←	AZ	BY	CZ
	BX	AY	DZ
	BY	AX	DY
←	BZ	AY	DZ
	CX	BY	CZ
	CY	BX	CY
	CZ	BY	CZ
	DX	AY	DZ
	DY	AX	DY
←	DZ	AY	DZ

Tento automat má celkem šest koncových stavů, ale všechny jsou nedosažitelné, tzn. $L(A_2)–L(A_1)=\emptyset$.
A odtud dále plyne, $L(A_1)–L(A_2)\cup L(A_2)–L(A_1)=\emptyset$ a tudíž podle důkazu věty platí, že $L(A_1)=L(A_2)$.

2.3 Zřetězení, mocnina, iterace, zrcadlový obraz a kvocient

Dalšími operacemi, na které je třída regulárních jazyků uzavřená, jsou zřetězení a iterace. Sestrojení automatů pro tyto operace je opět logické a není třeba za ním hledat silnou teorii. Pokud budeme vytvářet automat pro zřetězení dvou automatů $A_1 \cdot A_2$, znamená to, že by měl rozpoznávat slova, složená v první části ze slov automatu A_1 a v druhé části z A_2. Algoritmus pro vytváření zřetězení se tedy vytvoří tak, že naváže výstupy A_1 na vstupy A_2 pomocí ε-přechodů. Podobně tomu bude u iterace, kdy se budou výstupy automatu navazovat na jeho vlastní vstupy.

Definice 17: ***Součinem*** **(nebo též** ***zřetězením) jazyků*** **L_1,L_2 nazveme jazyk $L_1 \cdot L_2=\{ uv|u \in L_1 \wedge v \in L_2\}$**
n-tou mocninu **L^n** ***jazyka*** **L definujeme induktivně takto:**
$L^0=\{ e\}$
$L^{n+1}=L^n \cdot L=L^n L$ (pro $n \geq 0$)
Iterace **L^*** ***jazyka*** **L a** ***pozitivní iterace*** **L^+** ***jazyka*** **L jsou definovány následovně:**
$L^* = L^0 \cup L \cup L^2 \cup L^3 \cup ... = \cup_{0 \leq n \leq ...} L^n$
$L^+ = L \cup L^2 \cup L^3 \cup ... = \cup_{1 \leq n \leq ...} L^n$

Součin, mocnina, iterace jazyků

Řešený příklad 19:
Mějme jazyky: $L_1=\{a^2\}$, $L_2=\{b^n; n \geq 0\}$, $L_3=\{(ab)^n; n \geq 0\}$
zřetězení jazyků: $L_1L_2=\{a^2b^n; n \geq 0\}$, $L_2L_1=\{b^na^2; n \geq 0\}$,
$L_2L_3=\{b^n(ab)^k; n,k \geq 0\}$
n-tá mocnina jazyka: $L_1{}^3=\{a^6\}$

iterace jazyka: $L_1^*=\{a^{2n}; n \geq 0\}$

Algoritmus pro zřetězení A = $A_1 \cdot A_2$:

1. Sestrojíme ZNKA složený z automatů A_1, A_2.
2. Přechodovou funkci obohatíme o ε-přechody z výstupních stavů A_1 na vstupní stav A_2

Algoritmus pro zřetězení, iteraci

Algoritmus pro iteraci A = $(A_1)^*$:

1. Sestrojíme ZNKA složený z automatu A_1
2. Přechodovou funkce obohatíme o ε-přechody z výstupních stavů A_1 na vstupní stav A_1

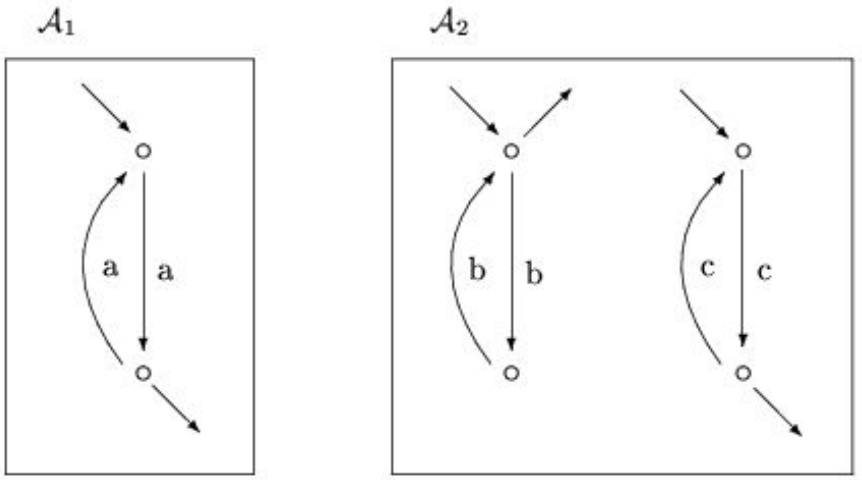
Věta 7: **Pro libovolné jazyky $L_1, L_2 \in F$ je i $L_1 \cdot L_2 \in F$ a také $L_1^* \in F$.**

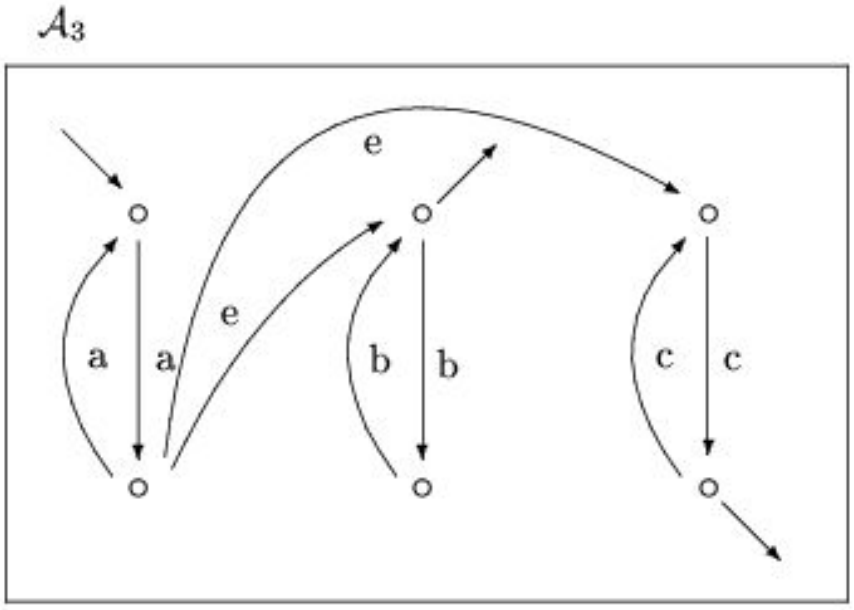
Uzávěrové vlastnosti -zřetězení, iterace

Důkaz:
Předpokládejme, že $L_1=L(A_1)$ a $L_2=L(A_2)$ pro NKA $A_1=(Q_1,\Sigma,\delta_1,I_1,F_1)$, $A_2=(Q_2,\Sigma,\delta_2,I_2,F_2)$ přičemž předpokládáme, $Q_1 \cap Q_2=\emptyset$.
1. Zkonstruujeme ZNKA $A=(Q,\Sigma,\delta,I,F)$ takový, že $L(A)=L_1 \cdot L_2$.
Stačí položit $Q=Q_1 \cup Q_2$, $I=I_1$, $F=F_2$ a definovat funkci $\delta: Q \times (\Sigma \cup \{e\}) \rightarrow P(Q)$ následovně:
pro lib. $q \in Q_1$, $a \in \Sigma$ je $\delta(q,a)=\delta_1(q,a)$, pro lib. $q \in Q_2$, $a \in \Sigma$ je $\delta(q,a)=\delta_2(q,a)$, pro $q \in F_1$ je $\delta(q,e)=I_2$, pro $q \in Q-F_1$ je $\delta(q,e)=\emptyset$. Ověříme $L(A)=L_1 \cdot L_2$.
a) Nechť $w \in L_1 \cdot L_2$, tedy $w=uv$, kde $u \in L_1$, $v \in L_2$, tedy existuje výpočet (posloupnost stavů), odpovídající přechodové funkci a slovu u, začínající v nějakém stavu $q_1 \in I_1$ a končící v nějakém stavu $q_2 \in F_1$, podobně existuje výpočet, odpovídající slovu v, začínající v nějakém stavu $q_3 \in I_2$ a končící v nějakém stavu $q_4 \in F_2$. Ovšem podle definice A lze z q_2 přejít ε-přechodem do q_3 a tedy $uv \in L(A)$. Tedy $L_1 \cdot L_2 \subseteq L(A)$.
b) Nechť $w \in L(A)$. Tedy existuje výpočet odpovídající slovu w, který začíná v nějakém $q_1 \in I(=I_1)$ a končí v nějakém $q_2 \in F(=F_2)$. Výpočet tedy začíná v množině Q_1 a končí v množině Q_2. Přechod z Q_1 do Q_2 mohl proběhnout jen jednou a to ε-přechodem z nějakého $s_1 \in F_1$ do nějakého $s_2 \in I_2$. To znamená, že w můžeme psát $w=uv$, kde slovu u odpovídá výpočet začínající v $q_1 \in I_1$ a končící v $s_1 \in F_1$ a slovu v odpovídá výpočet začínající v $s_2 \in I_2$ a končící v $q_2 \in F_2$. Tedy $u \in L(A_1)=L_1$ a $v \in L(A_2)=L_2$, tudíž $w \in L_1 \cdot L_2$. Platí tedy $L(A) \subseteq L_1 \cdot L_2$. A tedy $L(A)=L_1 \cdot L_2$.
2. Zkonstruujeme ZNKA $A'=(Q',\Sigma,\delta',I',F')$ takový, že $L(A')=L_1^*$.
Položme $Q'=Q_1 \cup \{r\}$, kde $r \notin Q_1$, $I'=I_1 \cup \{r\}$, $F'=F_1 \cup \{r\}$ a δ' definujme následovně:
$\forall q \in Q_1$, $\forall a \in \Sigma$ je $\delta'(q,a) = \delta_1(q,a)$, $\forall q \in F_1$ je $\delta'(q,e)=I_1$, $\forall q \in Q_1-F_1$ je $\delta'(q,e)=\emptyset$, $\forall a \in \Sigma$ je $\delta'(r,a)=\emptyset$.
Důkaz ověření $L(A')=L_1^*$ lze provést podobně jako u bodu 1.

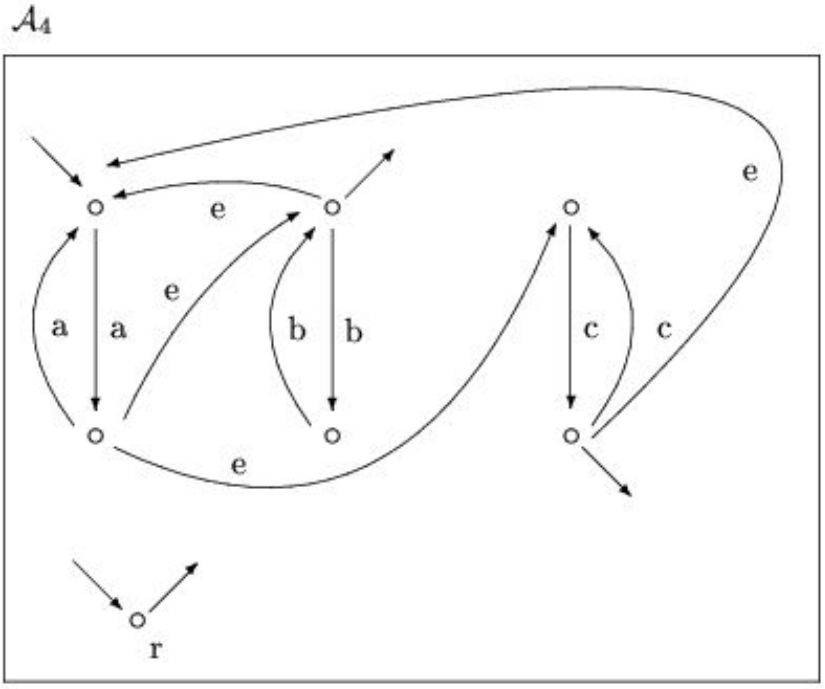
Řešený příklad 20:

$L_1=\{w;w=a^{2k+1}$ pro $k \geq 0\}$
$L_2=\{w;w=b^{2k}; k \geq 0$ nebo $w=c^{2k+1}; k \geq 0\}$
$L_1=L(A_1)$, $L_2=L(A_2)$ (obr.)





Obrázek znázorňuje A_3 ZNKA rozpoznávající jazyk L_1L_2, obrázek znázorňuje ZNKA A_4 rozpoznávající jazyk $(L_1L_2)^*$.



Definice 18: ***Inverzí slova*** **w=a_1 a_2... a_n ($a_i \in \Sigma$) rozumíme slovo $w^R=a_n$ a_{n-1}... a_1, speciálně $e^R=e$.** ***Inverzí jazyka*** **L rozumíme jazyk, který označíme $L^R=\{ w^R|w \in L\}$.**

Věta 8: **Pro libovolný jazyk L platí $L \in F \Leftrightarrow L^R \in F$.**

Důkaz:
(V automatu pro daný jazyk vyměníme počáteční a koncové stavy a opačně orientujeme šipky.) Stačí dokázat $L \in F \Rightarrow L^R \in F$. Pak totiž $L^R \in F \Rightarrow (L^R)^R \in F$, neboli vzhledem k faktu $(L^R)^R=L$ platí $L^R \in F \Rightarrow L \in F$.
Předpokládáme $L=L(A)$ pro NKA $A=(Q,\Sigma,\delta,I,F)$. Sestrojíme NKA $A'=(Q,\Sigma,\delta',I',F')$ takový, že $L(A')=L^R$. Položíme $I'=F$, $F'=I$ a δ' je definována následovně:
$(q' \in \delta'(q,a)) \Leftrightarrow (q \in \delta(q',a))$ pro všechny $q,q' \in Q$, $a \in \Sigma$. Snadno se ověří, že $w \in L(A) \Leftrightarrow w^R \in L(A')$, a tedy $L(A')=L^R$.

Definice 19: ***Levý kvocient jazyka*** **L_1** ***podle jazyka*** **L_2 je jazyk $L_2 \backslash L_1=\{ u|wu \in L_1$ pro nějaké $w \in L_2\}$.** ***Pravý kvocient jazyka*** **L_1** ***podle jazyka*** **L_2 je jazyk $L_1 \backslash L_2=\{ u|uw \in L_1$ pro nějaké $w \in L_2\}$.**

Věta 9: **Jestliže $L_1,L_2 \in F$, pak také $L_2 \backslash L_1 \in F$ a $L_1 \backslash L_2 \in F$.**

Důkaz tohoto tvrzení již nebudeme provádět, zájemce jej může najít v doporučené literatuře.

2.4 Konstrukce používané v důkazech

Stromový algoritmus v literatuře většinou definován není. Považuji však tuto variantu za mnohem lépe pochopitelnou a navíc není nutné se učit nový postup (umíte-li stromový algoritmus, pak se velmi lehce naučíte jeho modifikace). Výhodou tohoto přístupu je, že funguje i pro nedeterministické automaty na vstupu algoritmu, kdežto algoritmus, který si naznačíme níže vyžaduje, aby do něj vstupovaly již deterministické automaty (to může zbytečně zpomalit Váš výpočet). Na druhou stranu algoritmus uspořádaných dvojic je formálně velice detailně popsán v důkazech vět, proto by pro Vás mohlo být přínosné, vidět význam těchto důkazů v praktických příkladech.
Až si projdete příklady uvidíte, že tento algoritmus je vlastně speciální případ stromového algoritmu. Stromový algoritmus je tedy obecnější nicméně je překvapivě i efektivnější (kombinace těchto dvou vlastností je z hlediska algoritmizace i ilustrativnosti samozřejmě dobrá). Strom totiž vynechává zbytečné dvojice.

Algoritmus pro sestrojení $A_1 \cup A_2$ (uspořádané dvojice):

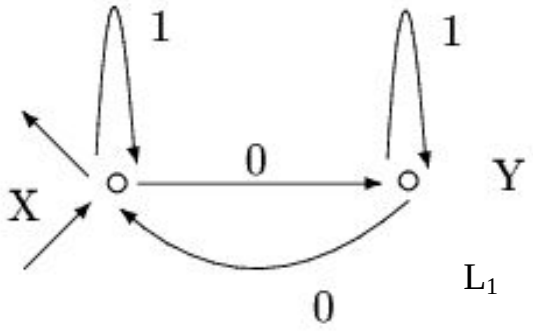
1. Sestrojíme všechny možné kombinace stavů $[q_i,q_j]$, kde $q_i \in Q_1$ a $q_j \in Q_2$ *(vytváříme spojení obou automatů – tedy proto dvojice)*
2. Vstupní dvojice je $[q_1,q_2]$ *(stejné vysvětlení jako u stromového alg.)*
3. Výstupní dvojice jsou všechny $[q_i,q_j]$, kde $q_i \in F_1$ nebo $q_j \in F_2$ *(vytváříme sjednocení obou automatů – všechny kombinace výstupních stavů)*
4. Přechodovou funkci δ vytvoříme tak, že pro lib. $p_1 \in Q_1$, $p_2 \in Q_2$, $a \in \Sigma$ je $\delta([p_1,p_2],a) = [\delta_1(p_1,a),\delta_2(p_2,a)]$. *(jinak řečeno přesně okopírujeme funkce obou automatů podle pozic ve dvojicích)*

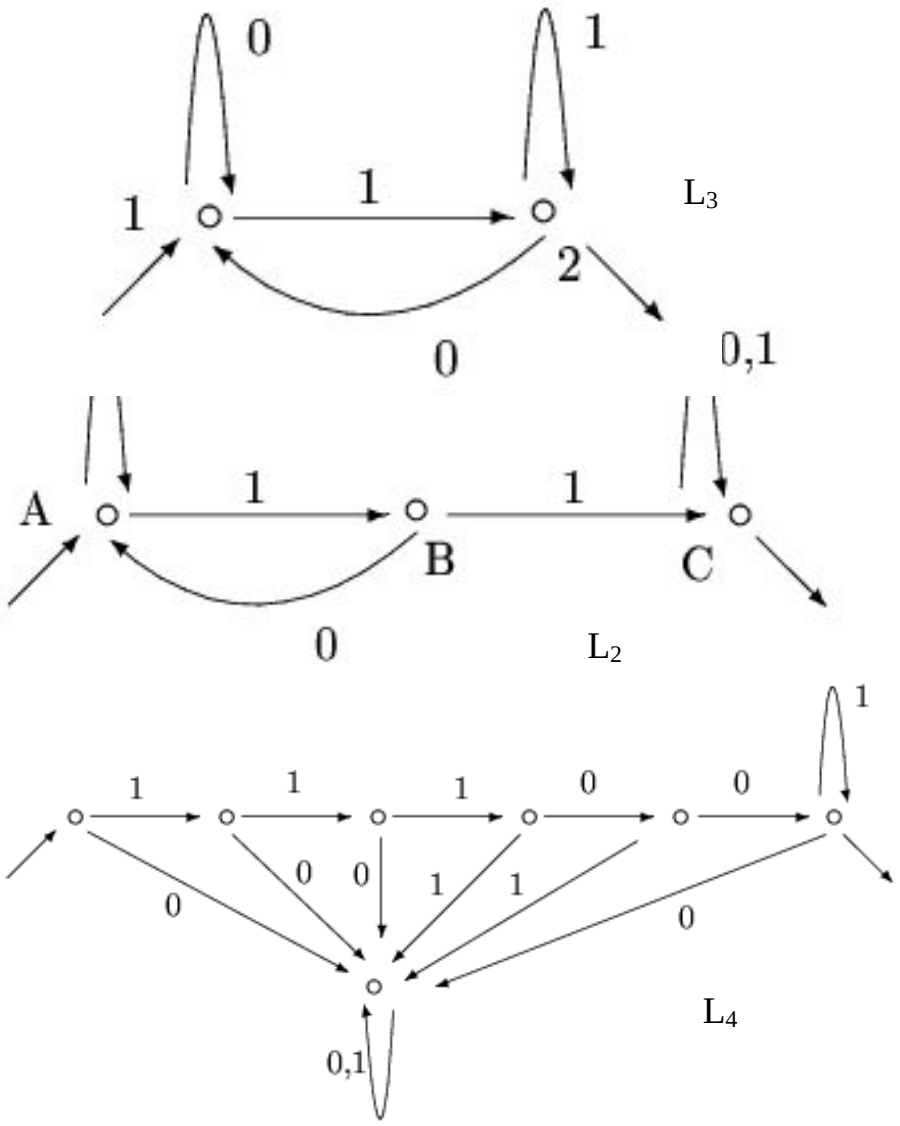
Algoritmus pro sestrojení $A_1 \cap A_2$ (uspořádané dvojice):

1. Sestrojíme všechny možné kombinace stavů $[q_i,q_j]$, kde $q_i \in Q_1$ a $q_j \in Q_2$ *(vytváříme spojení obou automatů – tedy proto dvojice)*
2. Vstupní dvojice je $[q_1,q_2]$ *(stejné vysvětlení jako u stromového alg.)*
3. Výstupní dvojice jsou všechny $[q_i,q_j]$, kde $q_i \in F_1$ a zároveň $q_j \in F_2$ *(vytváříme sjednocení obou automatů – všechny kombinace výstupních stavů)*
4. Přechodovou funkci δ vytvoříme tak, že pro lib. $p_1 \in Q_1$, $p_2 \in Q_2$, $a \in \Sigma$ je $\delta([p_1,p_2],a) = [\delta_1(p_1,a),\delta_2(p_2,a)]$. *(jinak řečeno přesně okopírujeme funkce obou automatů podle pozic ve dvojicích)*

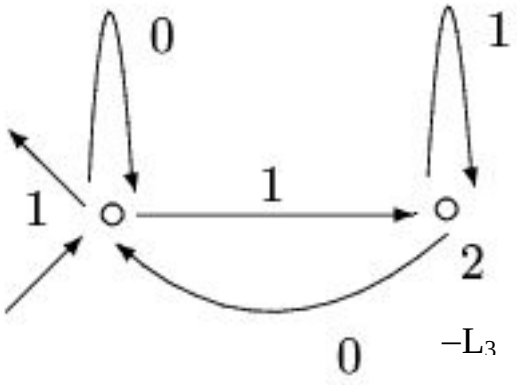
Kontrolní úkol:
Mějme jazyky L_1, L_2, L_3:
$L_1=\{w \in \{0,1\}^* | w$ obsahuje sudý počet symbolů $0\}$
$L_2=\{w \in \{0,1\}^* | w$ obsahuje podslovo $11\}$, $L_3=\{w \in \{0,1\}^* | w$ končí symbolem $1\}$. Mějme dále jazyk $L_4=\{w \in \{0,1\}^* | w$ se skládá ze dvou úseků, z nichž první je roven slovu 11100 a druhý obsahující pouze symboly 1 má délku d ($d \geq 0$)$\}$.
Sestrojte konečné automaty rozpoznávající jazyky: $L_2 - L_3$, $L_3 \cap L_1$, $L_2 \cup L_1$, $L_2 - L_1$, $-L_1$, $-L_3$.

Řešení:
Pro řešení uvedených problémů budeme potřebovat automaty rozpoznávající jazyky L_1, L_2, L_3, L_4.





$L_2 - L_3 (= \{w \in \{0,1\}^* | w$ obsahuje 11 a končí $0\})$ Jazyk $-L_3$ rozpoznává konečný automat na obrázku.

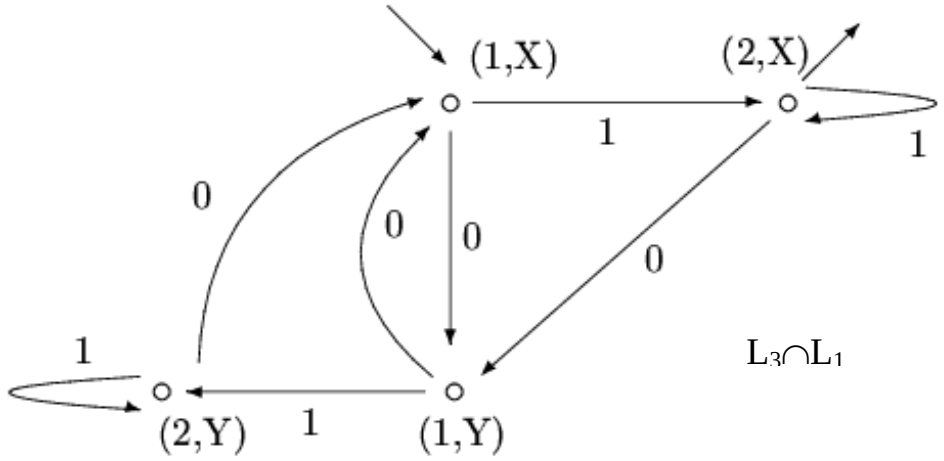


Jazyk $L_2 - L_3 = L_2 \cap (-L_3)$ rozpoznává automat A:

	$Q\backslash\Sigma$	0	1
→	(A,1)	(A,1)	(B,2)
	(B,1)	(A,1)	(C,2)
←	(C,1)	(C,1)	(C,2)
	(A,2)	(A,1)	(B,2)
	(B,2)	(A,1)	(C,2)
	(C,2)	(C,1)	(C,2)

$L_3 \cap L_1$(={w ∈ {0,1}*|w obsahuje sudý počet symbolů 0 a končí symbolem 1})

Konstrukce automatu pro jazyk $L_3 \cap L_1$. Jazyk $L_3 \cap L_1$ rozpoznává konečný automat A:



	$Q\backslash\Sigma$	0	1
→	(1,X)	(1,Y)	(2,X)
	(1,Y)	(1,X)	(2,Y)
←	(2,X)	(1,Y)	(2,X)
	(2,Y)	(1,X)	(2,Y)

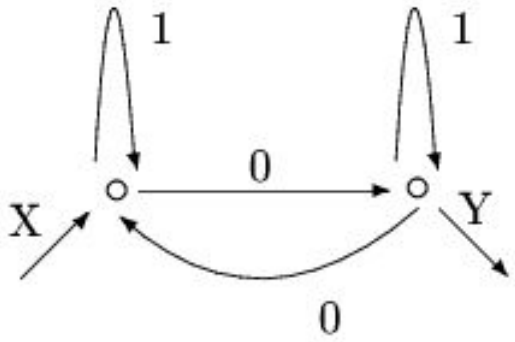
$L_2 \cup L_1$(={w ∈ {0,1}*|w obsahuje sudý počet symbolů 0 nebo obsahuje 11})
Konstrukce automatu pro jazyk $L_2 \cup L_1$ pomocí uspořádaných dvojic:

Jazyk $L_2 \cup L_1$ rozpoznává deterministický konečný automat A:

	$Q\backslash\Sigma$	0	1

↔	(A,X)	(A,Y)	(B,X)
	(A,Y)	(A,X)	(B,Y)
←	(B,X)	(A,Y)	(C,X)
	(B,Y)	(A,X)	(C,Y)
←	(C,X)	(C,Y)	(C,X)
←	(C,Y)	(C,X)	(C,Y)

2. $L_2 - L_1 (= \{w \in \{0,1\}^* | w$ obsahuje podslovo 11 a neobsahuje sudý počet symbolů 0$\})$ Konstrukce automatu pro jazyk $L_2 - L_1$. Jazyk $-L_1$ rozpoznává konečný automat na obrázku.



$-L_1$

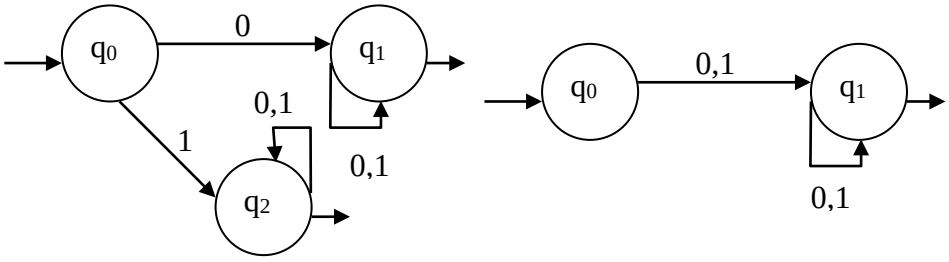
Jazyk $L_2 - L_1 = L_2 \cap (-L_1)$ rozpoznává automat A:

	$_{Q \setminus \Sigma}$	0	1
→	(A,X)	(A,Y)	(B,X)
	(A,Y)	(A,X)	(B,Y)
	(B,X)	(A,Y)	(C,X)
	(B,Y)	(A,X)	(C,Y)
	(C,X)	(C,Y)	(C,X)
←	(C,Y)	(C,X)	(C,Y)

$-L_1 = \{w \in \{0,1\}^* | w$ neobsahuje sudý počet symbolů 0$\}$ Konstrukce automatu pro jazyk $-L_1$.

$-L_3 = \{w \in \{0,1\}^* | w$ nekončí symbolem 1$\}$. Konstrukce automatu pro jazyk $-L_3$ na obrázku.

Již na začátku Vašeho studia jste se seznámili s pojmem ekvivalentní automat. Víte, že dva automaty jsou ekvivalentní, pokud rozpoznávají stejný jazyk. Příkladem mohou být dva následující automaty:



Oba tyto automaty rozpoznávají jazyk L = { [(0+1)(0+1)*]}. Přesto automat druhý má o jeden stav méně. Naopak automat první má stavy q_1,q_2, které jsou ekvivalentní (laicky řečeno – dělají stejnou práci). Právě takové stavy by bylo dobré odhalovat a nahrazovat je pouze jedním stavem. Tím bychom dostali minimální počet stavů – tzv. redukovaný automat. V tomto případě je redukovaný automat druhý automat – méně stavů mít nemůže.

2.5 Algoritmus redukce

Pro odhalování ekvivalentních stavů slouží algoritmus redukce. Tento postup vychází z toho, že v automatu se vyskytují dvě jistě neekvivalentní množiny stavů – výstupní a nevýstupní. Tyto množiny pak postupně rozdělujeme pomocí zkoumání, jaká je jejich přechodová funkce. Pokud se v množině vyskytne více kombinací, pak je všechny rozdělíme do nových množin podle těchto kombinací. Pokud je nějaká množina jednoprvková, pak už je jasné, že nelze takovou množinu rozdělit. Algoritmus končí, když už nedochází k žádnému rozdělování. Pokud se všechny množiny rozpadnou do jednoprvkových, tak automat neobsahuje žádné ekvivalentní stavy – jinak řečeno má minimální počet stavů. Pokud ne, pak můžeme sestrojit redukovaný automat tak, že považujeme množiny za stavy, podobně jako při sestrojování DKA z NKA.

Definice 20: Nechť $A=(Q,\Sigma,\delta,q_0,F)$ je KA a $p,q \in Q$. Řekneme, že *stavy* p,q jsou *ekvivalentní* , píšeme $p \sim q$, jestliže $\forall w \in \Sigma^*$ je $\delta^*(p,w) \in F \Leftrightarrow \delta^*(q,w) \in F$. Snadno lze ověřit, že $\sim$ je relací ekvivalence na množině Q.

Ekvivalentní stavy

Poznámka: Nyní ukážeme jak zkonstruovat příslušný rozklad Q podle $\sim$ ($Q \setminus \sim$).

Definice 21: Pro KA $A=(Q,\Sigma,\delta,q_0,F)$ definujeme posloupnost relací $\sim^0, \sim^1, \sim^2,...$ na množině Q takto: pro libovolné $p,q \in Q$

$p \sim^0 q \Leftrightarrow_{def.} (p \in F \Leftrightarrow q \in F)$

$p \sim^i q (i \geq 1) \Leftrightarrow_{def.} p \sim^{i-1} q \wedge \forall a \in \Sigma$ je $\delta(p,a) \sim^{i-1} \delta(q,a)$.

Relace rozkladu

Je zřejmé, že $\sim^i$ jsou relace ekvivalence. Označme R_i příslušný rozklad Q podle $\sim^i$ (tedy $R_i = Q \setminus \sim^i$).

Poznámka: Všimněme si, že $p \sim^{i} q$ ($i \geq 1$) právě, když libovolné slovo délky nejvýše i převede automat ze stavu p, respektive q, v obou případech do koncového stavu nebo v obou případech do nekoncového stavu (důkaz indukcí podle i).

Věta 10: Při značení jako v předchozí definici platí:
$\forall i \geq 0$ je R_{i+1} zjemněním R_i ($p \sim^{i+1} q \Rightarrow p \sim^{i} q$)
Jakmile $R_i=R_{i+1}$ pro nějaké $i \geq 0$, pak $R_i=R_{i+j}$ $\forall j \geq 1$
Jestliže $|Q| = n$, pak $\exists k \leq n-1$ tak, že $R_k=R_{k+1}$
Pro libovolné k takové, že $R_k=R_{k+1}$, je $\sim^{k}$ totožná s $\sim$ (tedy $\forall p,q \in Q$ platí $p \sim^{k} q \Leftrightarrow p \sim q$ neboli $R_k=Q\backslash \sim$).

Důkaz:
Triviálně z definice.
Předpoklad $R_i=R_{i+1}$ pro nějaké $i \geq 0$.
Odtud plyne $R_{i+1}=R_{i+2}$
$((p \sim^{i+1} q) \Leftrightarrow (p \sim^{i} q \wedge \forall a \in \Sigma$ je $\delta(p,a) \sim^{i} \delta(q,a)) \Leftrightarrow (p \sim^{i+1} q \wedge \forall a \in \Sigma$ je $\delta(p,a) \sim^{i+1} \delta(q,a)) \Leftrightarrow (p \sim^{i+2} q))$
a tedy $R_{i+2}=R_i$
Indukční předpoklad $R_i=R_{i+k}$ $\forall k \leq n (n \geq 2)$
Chceme dokázat, že $R_i=R_{i+k}$ platí pro všechna $k \leq n+1$
Platí $R_i=R_{i+n}$, ale také $R_i=R_{i+n-1}$ a odtud $R_{i+n-1}=R_{i+n}$ a odtud dále (podobně jako v a)) $R_{i+n}=R_{i+n+1}$, ale platí-li zároveň $R_i=R_{i+n}$, pak $R_i=R_{i+n+1}$ a důkaz je hotov.
Počet tříd rozkladu R_i označíme n_i, jestliže $R_0,R_1,...R_k$ jsou navzájem různé, pak $1 \leq n_0 < n_1 < ... n_k \leq n$. Z toho plyne, že $k \leq n-1$. Existuje tedy $k \leq n-1$ takové, že $n_k=n_{k+1}$, tj. $R_k=R_{k+1}$.
Z definice lze odvodit, že pro libovolné $i \geq 0$ a pro libovolné stavy $p,q \in Q$ je $p \sim^{i} q$, právě když pro každé slovo $w \in \Sigma^*$ délky nejvýše i je $\delta^*(p,w) \in F \Leftrightarrow \delta^*(q,w) \in F$.
- Dál z definice je jasné, že dva libovolné stavy p,q jsou ekvivalentní, právě když $p \sim^{i} q$ pro všechna i.
- Jestliže existuje-li k takové, že $R_k=R_{k+1}$, pak platí 2. Tzn., že pro libovolné $p,q \in Q$ takové, že $p \sim^{k} q$, každé slovo (libovolné délky) ze Σ^* převede automat z p resp. z q současně do koncového nebo nekoncového stavu $\Leftrightarrow p \sim q$.

Algoritmus redukce

Algoritmus: (Rozklad stavové množiny podle ~)
Vstup: KA $A=(Q,\Sigma,\delta,q_0,F)$;
Výstup: Rozklad $Q\backslash \sim$ (~ podle definice)
Sestroj rozklad R_0; i:=0;
repeat
i:=i+1;
sestroj R_i;
until $R_i=R_{i-1}$;
výstupem bude R_i

Definice 22: **Nechť $A=(Q,\Sigma,\delta,q_0,F)$ je KA a $\sim$ je ekvivalence. Definujme *podílový automat* $A\backslash\sim$ automatu A podle ekvivalence $\sim$ takto:**
$A\backslash\sim = (Q\backslash\sim,\Sigma,\delta_\sim,[q_0],\{[q], q\in F\})$, kde $\delta_\sim([q],a)=[\delta(q,a)]\ \forall q\in Q, a\in\Sigma$.
(definice je korektní, neboť $[p]=[q]\Rightarrow[\delta(p,a)]=[\delta(q,a)]$).

Ekvivalence podílového automatu

Věta 11: **Podílový automat $A\backslash\sim$ je ekvivalentní s automatem A ($L(A\backslash\sim)=L(A)$). Navíc $A\backslash\sim$ neobsahuje žádné dva různé vzájemně ekvivalentní stavy, a dále pokud A neobsahuje nedosažitelné stavy, pak ani $A\backslash\sim$ neobsahuje nedosažitelné stavy.**

Důkaz:
Nechť $A=(Q,\Sigma,\delta,q_0,F)$ a $A\backslash\sim = (Q\backslash\sim,\Sigma,\delta_\sim,[q_0],F_\sim=\{[q],q\in F\})$.
Indukcí podle délky w snadno ukážeme, že pro lib. $w\in\Sigma^*$ a lib. $q\in Q$ platí $\delta_\sim^*([q],w)=[\delta^*(q,w)]$.
Dále je zřejmé, že pro libovolné $q\in Q$ platí $q\in F\Leftrightarrow[q]\in F_\sim$.
Pak ovšem $w\in L(A)\Leftrightarrow\delta^*(q_0,w)\in F\Leftrightarrow[\delta^*(q_0,w)]\in F_\sim\Leftrightarrow\delta_\sim^*([q_0],w)\in F_\sim\Leftrightarrow w\in L(A\backslash\sim)$ (pro lib. $w\in\Sigma^*$),
tedy $L(A)=L(A\backslash\sim)$.

(Vezměme) nechť [p],[q] jsou ekvivalentní stavy.
Tedy $\forall w\in\Sigma^*$ platí $\delta_\sim^*([p],w)\in F_\sim\Leftrightarrow\delta_\sim^*([q],w)\in F_\sim$.
Tedy $[\delta^*(p,w)]\in F_\sim\Leftrightarrow[\delta^*(q,w)]\in F_\sim$.
Ale také $[\delta^*(p,w)]\in F_\sim\Leftrightarrow\delta^*(p,w)\in F$.
A také $[\delta^*(q,w)]\in F_\sim\Leftrightarrow\delta^*(q,w)\in F$.
Proto $(p\sim q)$ neboli $[p]=[q]$.

Poslední tvrzení, tj. neobsahuje-li A nedosažitelné stavy, neobsahuje nedosažitelné stavy ani $A\backslash\sim$ můžeme dokázat takto:
Automat $A\backslash\sim$ má jen stavy [q] pro které platí, že q je stav automatu A a žádné jiné. Je-li q dosažitelný, platí $\exists w\in\Sigma^*$ tak, že $\delta^*(q_0,w)=q$.
Ale odtud plyne $[\delta^*(q_0,w)]=[q]$, ale dál také $[\delta^*(q_0,w)]=\delta_\sim^*([q_0],w)$ a pak také $\delta_\sim^*([q_0],w)=[q]$, a to znamená, že [q] je dosažitelný stav automatu $A\backslash\sim$.
A tvrzení, že neobsahuje-li A nedosažitelné stavy, neobsahuje nedosažitelné stavy ani $A\backslash\sim$ je dokázáno.

Definice 23: **KA A nazveme *redukovaným* jestliže**

1. A nemá nedosažitelné stavy
2. žádné dva různé stavy A nejsou vzájemně ekvivalentní.

Definice 24: **KA B nazveme *reduktem* KA A jestliže**
1. $L(B)=L(A)$
2. B je redukovaný.

Redukt

Věta 12: **Existuje algoritmus, který k libovolnému KA A sestrojí (nějaký) jeho redukt.**

Důkaz:
Odstraníme-li z A nedosažitelné stavy a ke vzniklému A_1 sestrojíme podílový automat $A_1\backslash\sim$, pak je $A_1\backslash\sim$ reduktem automatu A.

Poznámka: Snadno lze ověřit, že vede k reduktu výchozího automatu i tento postup - sestrojení podílového automatu s následným odstraněním nedosažitelných stavů.

Ukážeme, že redukt automatu je určen jednoznačně, až na pojmenování stavů.

Definice 25: Mějme dva KA $A_1=(Q_1,\Sigma,\delta_1,q_1,F_1)$, $A_2=(Q_2,\Sigma,\delta_2,q_2,F_2)$. Zobrazení $h:Q_1\rightarrow Q_2$ nazveme *automatovým homomorfismem* , jestliže platí:

1. $h(q_1)=q_2$

2. $h(\delta_1(q,a))=\delta_2(h(q),a)\ \forall q\in Q_1, a\in\Sigma$

3. (musí zachovávat koncové stavy) $q\in F_1\Leftrightarrow h(q)\in F_2\ \forall q\in Q_1$

Je-li navíc h bijekcí (vzájemně jednoznačným přiřazením), nazýváme h *automatovým izomorfismem* . Jestliže aut. izomorfismus existuje říkáme, že A_1 a A_2 jsou *izomorfní* .

Následující tvrzení nám říká, že když vytvoříte redukt z jakýchkoliv automatů, které jsou ekvivalentní, pak musí být stejné až na uspořádání stavů. Uspořádání pak řeší normování automatu, které způsobí, že stavy budou vždy uspořádány stejným způsobem a tedy dva ekvivalentní automaty znormované musí být naprosto totožné.

Věta 13: Každé dva ekvivalentní redukované automaty jsou izomorfní.

Důkaz:

Nechť $A_1=(Q_1,\Sigma,\delta_1,q_1,F_1)$ a $A_2=(Q_2,\Sigma,\delta_2,q_2,F_2)$ jsou ekvivalentní redukované automaty. Ukážeme jisté zobrazení $h:Q_1\rightarrow Q_2$ a dokážeme, že je automatovým izomorfismem.
Definice h:
Ukážeme, že pro libovolné $q\in Q_1$, existuje právě jeden $p\in Q_2$ tak, že:
$\forall w\in\Sigma^*:\delta_1^*(q,w)\in F_1\Leftrightarrow\delta_2^*(p,w)\in F_2$ (6)
Kdyby existovali dva různé takové stavy p_1,p_2, pak by podle (6) byl p_1 ekvivalentní p_2, což je spor s tím, že A_2 je redukovaný. Tedy p existuje nejvýš jeden.
Libovolný $q\in Q_1$ je dosažitelný, tedy existuje $u\in\Sigma^*$ takové, že $\delta_1^*(q_1,u)=q$. Zvolme $p=\delta_2^*(q_2,u)$ a dokažme, že platí (6).
(6) je ekvivalentní tvrzení:
$\forall w\in\Sigma^*:\delta_1^*(q_1,uw)\in F_1\Leftrightarrow\delta_2^*(q_2,uw)\in F_2$
To ale plyne z předpokladu $L(A_1)=L(A_2)$.

Můžeme tedy položit $h(q)=p \Leftrightarrow_{def.}$ (6) platí pro p,q.

h je bijekce: p a q mají v (6) symetrické postavení.

h je homomorfismus:
1. Z podm. (6) a ekvivalence A_1 a A_2 plyne $h(q_1)=q_2$
2. pro libovolné $q \in Q_1$ existuje u tak, že $\delta_1^*(q_1,u)=q$.
$h(\delta_1(q,a)) = h(\delta_1^*(q_1,ua)) = \delta_2^*(q_2,ua) = \delta_2(\delta_2^*(q_2,u),a) = \delta_2(h(\delta_1^*(q_1,u)),a) = \delta_2(h(q),a)$
3. z (6) pro w=e plyne $q \in F_1 \Leftrightarrow h(q) \in F_2$.

Důsledek: Libovolné dva redukty daného konečného automatu jsou izomorfní (redukt je určen jednoznačně až na izomorfismus).

Vlastnosti reduktů

Důsledek: Mezi všemi automaty ekvivalentními s daným automatem A má (libovolný) jeho redukt nejmenší počet stavů.

Důkaz:
Mějme tedy automat A, který má n stavů a automat A_1, který je jeho reduktem a má n_1 stavů. Určitě platí $n_1 \leq n$. Předpokládejme, že existuje automat A_2 takový, že $L(A)=L(A_2)$ a A_2 má n_2 stavů a platí $n_2 < n_1$.
Ale odtud plyne:
redukt A_3 automatu A_2 má n_3 stavů a platí $n_3 \leq n_2$, A, A_2 - jsou ekvivalentní a jejich redukty A_1 a A_3 musí být izomorfní, ale to je ve sporu s tím, že $n_3 < n_1$ ($n_3 \leq n_2 < n_1 \leq n$).
Sporem jsme ukázali, že mezi všemi automaty ekvivalentními s daným automatem A má nejmenší počet stavů jeho libovolný redukt.

Poznámka: Libovůli v pojmenování stavů odstraníme převodem reduktu do ***normovaného tvaru .***

Normování spočívá v tom, že označujeme stavy arabskými číslicemi od vstupního stavu. Procházíme sloupce v daném pořadí symbolů tabulky a snažíme vždy stavu, který ještě není označen přiřadit nejnižší nepoužitou arabskou číslici. Pokračujeme se stavem s následující arabskou číslicí, který jsme ještě neprocházeli a aplikujeme stejný postup.

Algoritmus: Algoritmus převodu KA do normovaného tvaru.
Vstup: KA $A=(Q,\Sigma,\delta,q_0,F)$ bez nedosažitelných stavů, množina Σ uspořádaná.
Výstup: KA B, který je ekvivalentní s A a je v normovaném tvaru.

Nechť $|Q| = n$ a prvky Σ jsou uspořádané v pořadí $a_1,a_2,\ldots a_n$.

Ozn[q_0]:=1;InvOzn[1]:=q_0;
MnozOzn:={ q_0};PoslOzn:=1;
for i: = 1 to n do
for j: = 1 to m do
begin
q:=δ(InvOzn[i],a_j);
if q in MnozOzn then TAB[i,a_j]:=Ozn[q]
else
begin
PoslOzn:=PoslOzn + 1;Ozn[q]:=PoslOzn;
InvOzn[PoslOzn]:=q;MnozOzn:=MnozOzn ∪{ q};
TAB[i,a_j]:=PoslOzn;
end;
end;
V TAB označit počáteční stav (1) a koncové stavy.

Poznámka: Nedosažitelné stavy není třeba ve vstupu předcházejícího algoritmu odstraňovat. Tabulka se zaplňuje jednoznačně, až do doby, kdy jsou vyčerpány všechny dosažitelné stavy. (Toto v případě, že použijeme k vytvoření reduktu automatu postup uvedený v poznámce za důkazem věty).

Poznámka: Z věty v kapitole o uzávěrových vlastnostech víme, že lze algoritmicky ověřovat, zda L(A_1)=L(A_2) pro zadané KA A_1 a A_2. Jiný postup než v důkazu věty o uzávěrových vl. spočívá v sestrojení reduktů obou automatů a v jejich převodu do normovaného tvaru. Rovnají-li se výsledné tabulky (automaty), jsou A_1 a A_2 ekvivalentní, v opačném případě nikoliv.

2.6 Příklady redukce a normování

Řešený příklad 21:

Najděte množiny všech vzájemně ekvivalentních stavů automatu A. (Proveďte rozklad Q\~).
KA A:

	$Q\backslash^{\Sigma}$	0	1
→	A	A	B
	B	A	C
	C	D	C
←	D	D	C

Řešení:
(Algoritmus)
R_0={I={D},II={A,B,C}}

	0	1
A	II	II
B	II	II
C	I	II

R_1={I={D},II={A,B},III={C}}

	0	1
A	II	II
B	II	III

R_2={I={D},II={A},III={B},IV={C}}. Množinu Q nelze dále rozkládat a proto $Q\backslash \sim = R_2$.

Řešený příklad 22:

Sestrojte podílový automat automatu A:

	$Q\backslash^{\Sigma}$	a	b
	A	H	G
←	B	B	A
	C	E	D
	D	D	B
	E	C	D
	F	F	E
↔	G	G	F
	H	A	G

Řešení:

R_0={I={A,C,D,E,F,H},II={B,G}}

	a	b
A	I	II
C	I	I
D	I	II
E	I	I
F	I	I
H	I	II

	a	b
B	II	I
G	II	I

R_1={I={A,D,H},II={C,E,F},III={B,G}}

	a	b
A	I	III

D	I	III
H	I	III

	a	b
C	II	I
E	II	I
F	II	II

	a	b
B	III	I
G	III	II

R_2={I={A,D,H},II={C,E},III={F},IV={B},V={G}}

	a	b
A	I	V
D	I	IV
H	I	V

	a	b
C	II	I
E	II	I

R_3={I={A,H},II={D},III={C,E},IV={F},V={B},VI={G}}

	a	b
A	I	VI
H	I	VI

	a	b
C	III	II
E	III	II

$R_4=R_3=Q\backslash \sim$ = {I={A,H},II={D},III={C,E},IV={F},V={B},VI={G}}
Podílový automat:

	$Q\backslash^{\Sigma}$	a	b
	I	I	VI
	II	II	V
	III	III	II
	IV	IV	III
←	V	V	I
↔	VI	VI	IV

Řešený příklad 23:
Převeďte následující automat do redukovaného normovaného tvaru.

Uzávěrové vlastnosti a redukce KA

KA A:

	$Q\backslash^{\Sigma}$	0	1
→	A	D	B
←	B	D	B
	C	A	C
	D	A	C

Řešení:

R_0={I={B},II={A,C,D}}

	0	1
A	II	I
C	II	II
D	II	II

R_1={I={B},II={A},III={C,D}}

	0	1
C	II	III
D	II	III

$R_2=R_1=Q\backslash \sim$ = {I={B},II={A},III={C,D}}

	$Q\backslash^{\Sigma}$	0	1
←	I	III	I
→	II	III	I
	III	II	III

Tento automat je redukovaný (neobsahuje žádné nedosažitelné stavy a ani žádné dva různé vzájemně ekvivalentní stavy) a platí, že rozpoznává stejný jazyk jako původní automat. Je tedy reduktem a převedeme jej do normovaného tvaru.

	$Q\backslash^{\Sigma}$	0	1
→	1	2	3
	2	1	2
←	3	2	3

Kontrolní úkol:
Mějme automat A:

	$Q\backslash^{\Sigma}$	0	1
→	A	D	B
←	B	D	B
	C	A	C
	D	A	C
	E	D	B

Sestrojte podílový automat automatu A a je-li výsledný automat reduktem původního automatu, převeďte jej do normovaného tvaru.

Řešení:

R_0={I={B},II={A,C,D,E}}

	0	1
A	II	I
C	II	II
D	II	II
E	II	I

R_1={I={B},II={A,E},III={C,D}}

	0	1
A	III	I
E	III	I

	0	1
C	II	III
D	II	III

$R_2=R_1=Q\backslash \sim$ = {I={B},II={A,E},III={C,D}}.

	$Q\backslash^{\Sigma}$	0	1
←	I	III	I
→	II	III	I
	III	II	III

Tento automat je redukovaný (neobsahuje žádné nedosažitelné stavy a ani žádné dva různé vzájemně ekvivalentní stavy) a platí, že rozpoznává stejný jazyk jako původní automat. Je tedy reduktem a převedeme jej do normovaného tvaru .

	$Q\backslash^{\Sigma}$	0	1
→	1	2	3
	2	1	2
←	3	2	3

Kontrolní úkol:

Mějme automat A:

	$Q\backslash^{\Sigma}$	0	1

→	A	D	B
←	B	D	B
	C	A	C
	D	A	C
	E	A	D

Sestrojte podílový automat automatu A a je-li výsledný automat reduktem původního automatu, převeďte jej do normovaného tvaru.

Řešení:

R_0={I={B},II={A,C,D,E}}

	0	1
A	II	I
C	II	II
D	II	II
E	II	II

R_1={I={B},II={A},III={C,D,E}}

	0	1
C	II	III
D	II	III
E	II	III

$R_2=R_1=Q\setminus\sim$ = {I={B},II={A},III={C,D,E}}.

	$Q\setminus^{\Sigma}$	0	1
←	I	III	I
→	II	III	I
	III	II	III

Tento automat je redukovaný (neobsahuje žádné nedosažitelné stavy a ani žádné dva různé vzájemně ekvivalentní stavy) a platí, že rozpoznává stejný jazyk jako původní automat. Je tedy reduktem a převedeme jej do normovaného tvaru.

	$Q\setminus^{\Sigma}$	0	1
→	1	2	3
	2	1	2
←	3	2	3

Nejdůležitější probrané pojmy:

- množinové a jazykové operace
- algoritmy pro sjednocení, průnik, doplněk, rozdíl, zrcadlový obraz
- uzávěrové vlastnosti třídy jazyků rozpoznatelných KA
- rozhodnutelnost ekvivalence dvou automatů pomocí množinových operací
- ekvivalentní stavy

- relace rozkladu
- algoritmus redukce, podílový automat, ekvivalence podílového automatu
- redukt a jeho vlastnosti
- normovaný tvar reduktu

3 Regulární jazyky

V této kapitole se dozvíte:

- Jak je definován pojem regulárního jazyka.
- Jaký je vztah konečných automatů a regulárních jazyků
- Jaké vlastnosti mají jazyky nerozpoznatelné KA

Po jejím prostudování byste měli být schopni:

- Definovat regulární jazyk pomocí regulárního výrazu.
- Konstruovat KA k regulárnímu výrazu.
- Dokázat, že jazyk není regulární.

Klíčová slova této kapitoly:

Regulární jazyk, regulární výraz, Kleeneho věta, Nerodova věta.

Průvodce studiem

Studium této kapitoly navazuje na předešlé pojmy a postupy. Opět se neučíte algoritmické postupy pro převody (zejména převod regulárních výrazů na konečné automaty). To však není jediným problémem kapitoly, navíc se zde ověřují vlastnosti těchto algoritmických konstrukcí – provádějí se důkazy. Nalezení důkazu – v našem případě naštěstí konstruktivního typu (tedy pouze ověřujeme, zda algoritmus vede ke korektním výsledkům) – bývá pro studenty informatiky nejobtížnější dovednost.
Na studium této části si vyhraďte alespoň 10 hodin. Rozdělte si studium na dobré pochopení postupů-algoritmů a teprve po ověření pochopení na příkladech se pusťte i do složitějších partií věnovaných důkazům.

V kapitolách minulých jste se seznamovali s automaty – tedy nástroji, které rozpoznávají jazyky (umožňují Vám zjistit, zda slovo do jazyka patří). Jinak řečeno jsme analyzovali konkrétní jazyk pomocí automatu. Je však možné se na tento problém podívat z jiného – syntetického – hlediska. To znamená, že budeme chtít vytvářet (generovat) jazyk. K tomu nám slouží (kromě gramatik, které budeme studovat v druhé části) regulární výrazy, které generují regulární jazyky. Můžete si je představit jako jakýsi předpis (šablonu), podle které jsou slova z tohoto výrazu tvořena. Jelikož jsme v kapitolách 1 – 4 poměrně solidně pokročili s výkladem a mnohé jsme již naznačili pevně věřím, že Vám tato kapitola nebude dělat velké problémy. Pokusme se již nyní dát jednoduchý příklad takového výrazu:

Příklad:

Výraz (a + b)*b generuje jazyk L složený ze slov, která obsahují libovolnou kombinaci symbolů ‚a' a ‚b' a končí na symbol b. Vidíte, že operace, které se ve výrazu používají již znáte (kromě +). (a+b)* je zřetězeno s ‚b'. Operace ‚*' je iterací a ‚+' neznamená nic jiného než variantu – ‚a' nebo ‚b'.
L = {b,ab,bb,aab,abb,bab,bbb,...}

Regulární výrazy nejsou opět pouze teorií bez významu pro praxi. Uvědomte si, že v mnoha prostředcích, které používáte jsou implementovány. Ve vstupech tabulkových procesorů, databází se setkáte se vstupními filtry, které umožňují kontrolu, zda je například správně definováno datum v různých tvarech (DD/MM/RRRR, RRMMDD, atd.). Nebo například číslo s desetinnou čárkou lze definovat regulárním výrazem.

Příklad:

Jazyk L čísel s desetinnou tečkou lze intuitivně definovat takto:
(‚0' + ‚1' +...+ ‚9')(‚0' + ‚1' +...+ ‚9')*'.'(‚0' +...+ ‚9') (‚0' +...+ ‚9')*
tedy L={0.1, 0.23,123.456,.....}
Tedy tento výraz definuje číslo složené na začátku alespoň z jedné číslice (nebo více), pak následuje desetinná tečka a pak opět alespoň jedna číslice (nebo více).

3.1 Regulární jazyky a výrazy

Regulární jazyk je takový, který je vytvořen ze základních symbolů abecedy pouze s pomocí operací sjednocení, zřetězení a iterace (postupnou aplikací v libovolném počtu a pořadí). Cítíte asi, že to přesně koresponduje s operacemi, které se používají ve výše zmíněných výrazech. Proto regulární výrazy generují právě regulární jazyky. Definujme je nyní exaktně:

Definice 26: ***Třída*** **RJ(Σ)** ***regulárních jazyků v konečné abecedě*** **Σ je nejmenší třída jazyků v abecedě Σ, která obsahuje jazyky ∅ a { a} pro všechna a ∈ Σ, a je uzavřena na tzv. regulární operace, tj. operace ∪, ·, * (∪ sjednocení,· zřetězení,* iterace). Tedy pro lib. L_1,L_2 platí**

$$L_1, L_2 \in RJ(\Sigma) \Rightarrow L_1 \cup L_2 \in RJ(\Sigma)$$

$$L_1, L_2 \in RJ(\Sigma) \Rightarrow L_1 \cdot L_2 \in RJ(\Sigma)$$

$$L_1 \in RJ(\Sigma) \Rightarrow L_1^* \in RJ(\Sigma)$$

Regulární výrazy slouží k přehlednějšímu zápisu regulárních jazyků.

Příklad:
$\{e\} \in RJ(\Sigma)$
Důkaz:
$\varnothing \in RJ(\Sigma)$
$L \in RJ(\Sigma) \Rightarrow L^* \in RJ(\Sigma)$
$\varnothing^* = \varnothing^+ \cup \{e\} = \{e\}$

Definice 27: ***Třídu*** **RV(Σ)** ***regulárních výrazů nad abecedou*** **$\Sigma = \{ a_1, a_2, \ldots a_n\}$ definujeme jako nejmenší množinu slov v abecedě $\{ a_1, a_2, \ldots a_n, \varnothing, e, +, \cdot, *, (,)\}$, $\varnothing, e, +, \cdot, *, (,) \notin \Sigma$ splňující následující podmínky:**
$\varnothing \in RV(\Sigma)$, $e \in RV(\Sigma)$, $a \in RV(\Sigma)$ pro všechna $a \in \Sigma$
$\alpha, \beta \in RV(\Sigma) \Rightarrow (\alpha+\beta) \in RV(\Sigma)$
$\alpha, \beta \in RV(\Sigma) \Rightarrow (\alpha \cdot \beta) \in RV(\Sigma)$
$\alpha, \beta \in RV(\Sigma) \Rightarrow (\alpha^*) \in RV(\Sigma)$

Regulární výrazy

Každý regulární výraz označuje (reprezentuje) konkrétní regulární jazyk.

$\varnothing$ označuje jazyk $\varnothing$

e označuje jazyk $\{ e\}$

a označuje jazyk $\{ a\}$ pro libovolné $a \in \Sigma$

Jestliže regulární výraz α označuje L_1, a regulární výraz β označuje L_2, pak

$(\alpha+\beta)$ označuje jazyk $L_1 \cup L_2$

$(\alpha \cdot \beta)$ označuje jazyk $L_1 \cdot L_2$

α^* označuje jazyk $(L_1)^*$

Obecně budeme jazyk reprezentovaný regulárním výrazem α značit $[\alpha]$. Budeme vynechávat zbytečné závorky (např. vnější pár, zbytečné závorky vzhledem k asociativitě operací $\cup$, $\cdot$), tečky ($\cdot$), další závorky můžeme vynechávat na základě priorit operací. * má větší prioritu než $\cdot$ a ta má větší prioritu než +.

Procvičte si pochopení těchto pojmů nyní na konstrukci výrazů na těchto řešených příkladech:

Řešený příklad 24:
$L=\{w \in \{0,1\}^* | w$ obsahuje podslovo 101 nebo končí podslovem 00 předcházeným libovolným počtem trojic $011\}$.
$\alpha = (0+1)^* 101(0+1)^* + (011)^* 00$
$L=[\alpha]$

Řešený příklad 25:

L={w ∈ {0,1}*|w=uv, u obsahuje sudý počet symbolů 0, v obsahuje slovo 11}).
Regulární výraz popisující tento jazyk:
$(1^*01^*01^*)^*$ $(1+0)^*11(0+1)^*$
Vidíte, že tento výraz rozděluje slova na dvě části – první řeší sudý počet nul $(1^*01^*01^*)^*$ a druhá podslovo 11 $(1+0)^*11(0+1)^*$.

Řešený příklad 26:
L={w ∈ {0,1}*|w=uv, u končí symbolem 1, v obsahuje slovo 11}).
Regulární výraz popisující tento jazyk:
$(1+0)^*1(1+0)^*11(0+1)^*$

Na příkladech jste viděli, že jsou velmi podobné konstrukci automatů. V podstatě regulární výraz je laicky řečeno jen jiným způsobem, jak popisovat stejnou třídu jazyků. Tento fakt je ale třeba přesně exaktně formulovat a dokázat. Uvidíte, že to není nijak složité. Formulujeme si větu, která říká, že ke každému výrazu lze sestrojit automat a naopak, že automat rozpoznává jazyk, který je regulární. Jde o velmi zásadní vlastnost v teorii formálních jazyků, proto prosím věnujte jak tvrzení (Kleeneho věta), tak důkazu patřičnou pozornost. Důkaz je opět konstruktivní a jeho postup nám umožní formulovat i algoritmus v další podkapitole, díky němuž budete schopni ke každému výrazu sestrojit konečný automat naprosto automaticky. Tvrzení se dá rozdělit do dvou vět.

Věta 14: Každý regulární jazyk je rozpoznatelný konečným automatem.

Důkaz:
Snadno sestrojíme automaty pro jazyky ∅ a {a} (a ∈ Σ). Zbytek plyne z vět o uzavřenosti třídy F vůči regulárním operacím – sjednocení a zřetězení z minulé kapitoly. Poznámka: Důkazy zmíněných vět obsahují návod k sestavení algoritmu, který k libovolnému regulárnímu výrazu α sestrojí ZNKA rozpoznávající jazyk [α]. Počet stavů automatu A zhruba odpovídá délce α.

RJ -> KA

Druhý směr je těžší na dokazování. Přesto se pokuste jej pochopit. Spočívá v tom, že analyzujeme, do jakých stavů se může automat dostávat (a jazyky slov, které jej do těchto stavů dostávají). Pak vymezíme, které z těchto množin tvoří jazyk rozpoznávaný automatem a dokážeme, že k jeho složení není třeba jiných operací než sjednocení, zřetězení a iterace. Z toho logicky vyplývá, že jazyk je regulární.

Věta 15: Každý jazyk rozpoznatelný konečným automatem je regulární.

Důkaz:
Nechť A=(Q,Σ,δ,q_1,F) je konečný automat rozpoznávající L. Nechť

KA -> RJ

$Q=\{ q_1,...,q_n\}$ *(uspořádáme stavy)*
Pro každé i,j ($1 \le i, j \le n$) definujme
$R_{ij}=\{ w \in \Sigma^* | \delta^*(q_i,w)=q_j\}$ *(což jsou množiny slov, na které se automat dostane ze stavu i do j)*
tj. jako množinu slov, která převádějí automat ze stavu q_i do stavu q_j. Zřejmě platí, že

$$L(A)=L=\bigcup_{q_j \in F} R_{1j} \quad (3)$$ *(jazyk je konečným sjednocením takových množin)*

Stačí proto dokázat, že každé R_{ij} je regulární jazyk. Protože L vzniká podle (3) konečným sjednocením (tj. konečným počtem regulárních operací) z jistých R_{1j}, plyne z regulárnosti R_{1j}, že i L je regulární.
Dokazujeme tedy, že pro každé i,j je R_{ij} regulární.
Definujme R_{ij}^k jako množinu slov převádějících automat ze stavu q_i do stavu q_j bez meziprůchodu jakýmkoli stavem q_m takovým, že m > k. Pro každé i,j je zřejmě $R_{ij}=R_{ij}^n$, a proto stačí dokázat, že R_{ij}^k je regulární pro všechna i,j,k ($1 \le i,j,k \le n$).
Důkaz tohoto tvrzení provedeme indukcí podle k.
Indukční předpoklad: Pro libovolná i,j je R_{ij}^0 množina slov, která převedou automat ze stavu q_i do stavu q_j bez meziprůchodů jakýmkoli stavem. Taková slova, pokud vůbec existují mají délku nejvýše 1. Je tedy
$R_{ij}^0 \subseteq \Sigma \cup \{ e\}$
a proto je pro libovolná i,j množina R_{ij}^0 regulární.
Indukční krok: Předpokládejme, že pro jisté k ($0 \le k < n$) a všechna i,j je R_{ij}^k regulární. Dokážeme, že R_{ij}^{k+1} je regulární pro libovolná i,j. R_{ij}^{k+1} totiž můžeme vyjádřit ve tvaru
$$R_{ij}^{k+1}=R_{ij}^k \cup R_{i,k+1}^k \cdot (R_{k+1,k+1}^k)^* \cdot R_{k+1,j}^k \quad (4)$$
neboť jestliže w převádí automat z q_i do q_j bez meziprůchodu stavem s indexem vyšším než k+1, mohou nastat tyto možnosti:
Nedojde ani k jednomu průchodu stavem q_{k+1}. Potom $w \in R_{ij}^k$.
Dojde k m meziprůchodům stavem q_{k+1} ($m \ge 1$). Potom lze slovo w vyjádřit ve tvaru $w=uv_1 v_2 ... v_{m-1}z$, kde m dělících bodů mezi slovy $u,v_1,v_2,...,v_{m-1},z$ odpovídá jednotlivým meziprůchodům stavem q_{k+1}. Potom je
$u \in R_{i,k+1}^k$, $z \in R_{k+1,j}^k$, $v_p \in R_{k+1,k+1}^k$
pro všechna p ($1 \le p \le m-1$).
Proto je
$w \in R_{i,k+1}^k \cdot (R_{k+1,k+1}^k)^* \cdot R_{k+1,j}^k$
Množina na pravé straně rovnosti je tvořena čtyřmi regulárními operacemi z jazyků, které jsou podle indukčního předpokladu regulární. Je tedy také R_{ij}^{k+1} regulární pro všechna i,j. Tím je důkaz dokončen.

Obě tato tvrzení pak dohromady tvoří Kleeneho větu:

Věta 16: (*Kleene*) Libovolný jazyk je regulární, právě tehdy když je rozpoznatelný konečným automatem.

Kleeneho věta

Důkaz:

Důsledek dvou vět předchozích.

Věta 17: **Podle věty o uzávěrových vlastnostech (uzavřenost *F* vůči průniku a doplňku) lze mezi regulární operace přibrat operace průniku i doplňku. Regulární výrazy můžeme obohatit o symboly &,¬, kde α&β představuje jazyk [α]∩[β] a ¬α označuje −[α].**

Další operací,kterou můžeme u regulárních jazyků realizovat je substituce a homomorfismus. Nejde o nic jiného než o nahrazení symbolu celým jazykem. Tedy jde o jakési dosazení do „proměnné". Pokud je to co dosazujeme regulární jazyk, pak vznikne opět regulární jazyk.

Definice 28: **Nechť Σ je konečná abeceda a pro každé a ∈ Σ je dán jazyk σ(a) v abecedě Δ_a. Položme σ(e)={ e} a σ(uv)=σ(u)σ(v) pro každé u,v ∈ Σ^*. Potom zobrazení σ:$\Sigma^* \rightarrow P(\Delta^*)$, kde $\Delta = \cup_{a \in \Sigma}\Delta_a$ se nazývá *substituce* . Pro každý jazyk $L \subseteq \Sigma^*$ definujeme $\sigma(L)=_{def.}\cup_{w \in L}\sigma(w)$ a říkáme, že σ(L) vznikl substitucí σ z jazyka L. Substituce σ, u níž pro každé a ∈ Σ obsahuje σ(a) jediné slovo, se nazývá *homomorfismus* . Homomorfismus lze tedy považovat za zobrazení σ:$\Sigma^* \rightarrow \Delta^*$.**

Věta 18: **Nechť Σ je konečná abeceda a σ je regulární substituce, tzn. σ(a) je regulární jazyk pro každé a ∈ Σ. Potom pro libovolný regulární jazyk L je σ(L) také regulární.**

Důkaz:
Nechť α je regulární výraz reprezentující L a α_a (a ∈ Σ) jsou regulární výrazy reprezentující σ(a)(a ∈ Σ). Jestliže pro každé a ∈ Σ dosadíme do α za každý výskyt symbolu a výraz α_a, dostaneme zřejmě regulární výraz reprezentující σ(L).

3.2 Sestrojení automatu (ZNKA) k regulárnímu výrazu

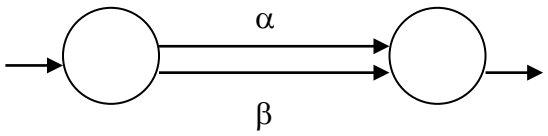
Na sestrojení automatu k regulárnímu výrazu můžete použít dva přístupy. První spočívá v postupné dekompozici výrazu na menší části podle regulárních operací a sestrojování schémat (připomínajících automaty), která nemusí mít ve svých přechodech pouze symboly, ale celé části výrazu. Až pak dojdeme na symboly, máme k dispozici zobecněný nedeterministický automat, který již můžeme převést na deterministický známými postupy.

Konstrukce automatu k regulárního výrazu (rozklad):

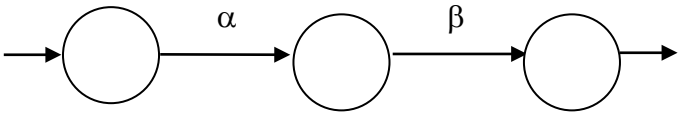
1. Mějme zadán výraz γ
2. K výrazu sestrojíme schéma podle jeho struktury:

- je-li γ = (α + β), pak

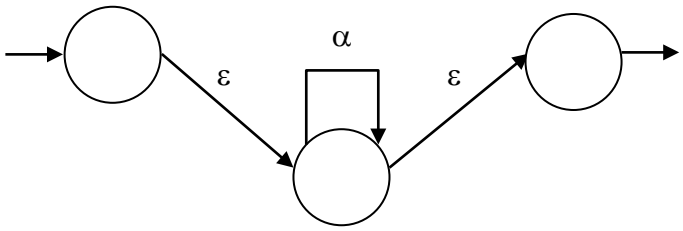
Konstrukce ZNKA k výrazu



- je-li $\gamma = (\alpha \cdot \beta)$, pak



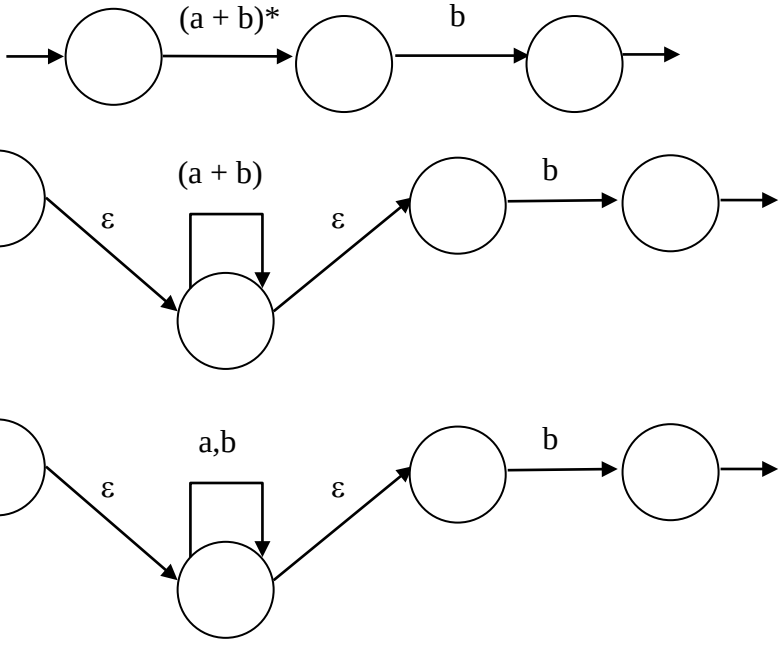
- je-li $\gamma = (\alpha)^*$, pak



3. Postup z bodu 2. aplikujeme na každou nově dekomponovanou část výrazu až jsou všechny přechody ve schématu pouze symboly abecedy

Řešený příklad 27:

Mějme výraz (a + b)*b. K němu postupnou dekompozicí sestrojíme ZNKA:



Na základě důkazu tvrzení, že ke každému regulárnímu výrazu existuje automat můžeme sestavit algoritmus pro takovou konstrukci. Tento postup je opačný k dekompozici podle algoritmu uvedeného výše.

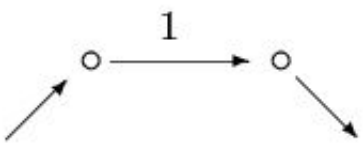
Konstrukce automatu k regulárního výrazu (skládání):

1. Mějme zadán výraz γ
2. Rozdělíme výraz na jednotlivé symboly
3. Jednotlivé části spolu spojujeme pomocí regulárních operací algoritmy pro automatové operace z předchozí kapitoly až sestrojíme výraz γ

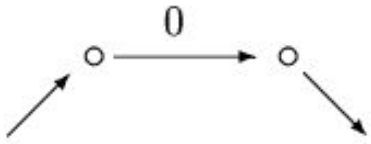
Řešený příklad 28:
Sestrojte KA, který rozpoznává jazyk $[(1+0)^*11]$.

Řešení:

1. Zkonstruujeme automat, rozpoznávající jazyk [1] a automat, rozpoznávající jazyk [0].

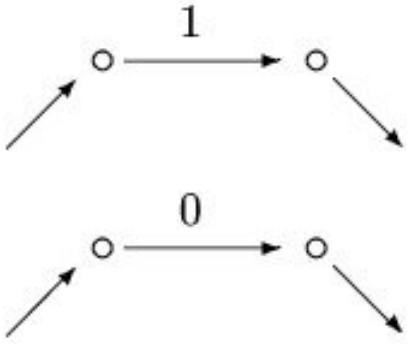


[1]



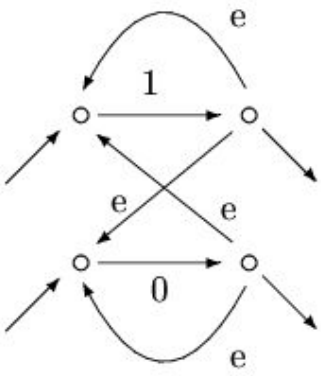
[0]

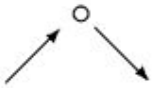
2. Zkonstruujeme automat, rozpoznávající jazyk [1+0], využijeme automaty pro [1] a [0].



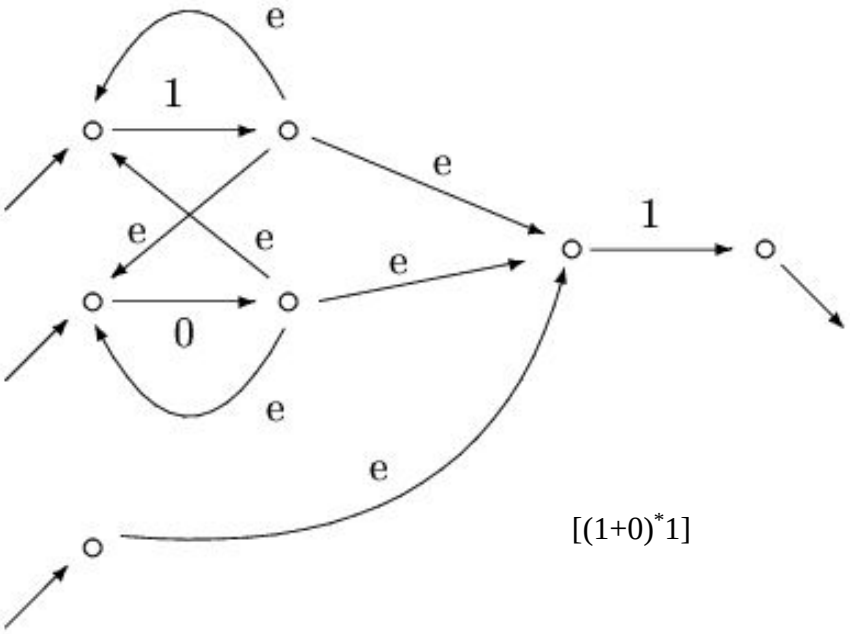
3. Zkonstruujeme automat, rozpoznávající jazyk $[(1+0)^*]$, využijeme automat pro [1+0].

4. Zkonstruujeme automat, rozpoznávající jazyk $[(1+0)^*1]$, využijeme automat pro $[(1+0)^*]$ a pro [1].
5. Zkonstruujeme automat, rozpoznávající jazyk $[(1+0)^*11]$, využijeme automat pro $[(1+0)^*1]$ a pro [1].

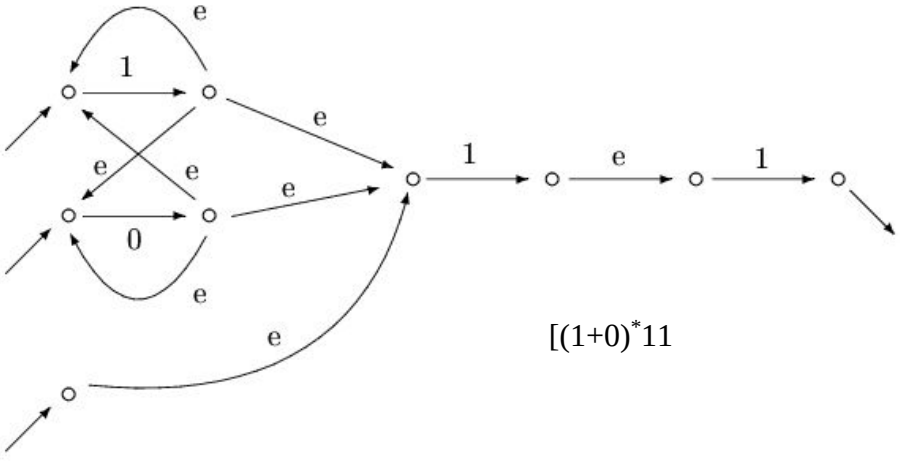




$[(1+0)^*]$



$[(1+0)^*1]$

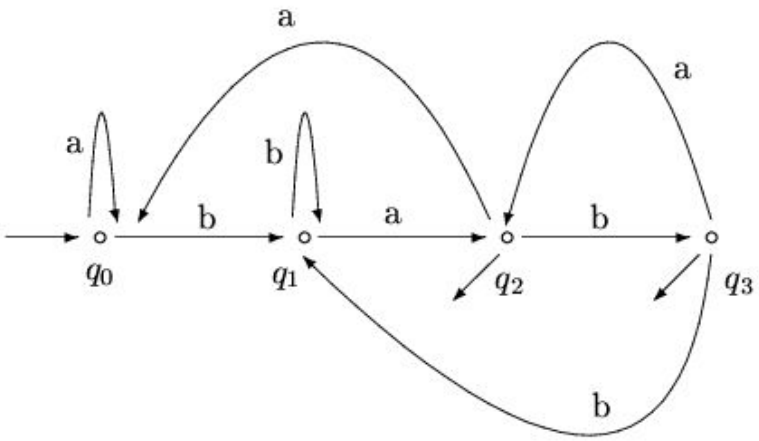


$[(1+0)^*11$

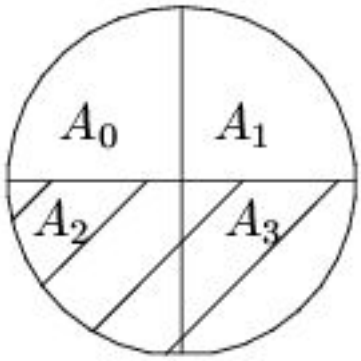
3.3 Pravá kongruence a Nerodova věta

Až do této kapitoly jsme se zabývali jazyky, ke kterým lze sestrojovat konečné automaty. Již na počátku studia jsme si ale naznačili, že jazyky jsou různě složité. Řekli jsme si také, že regulární jazyky jsou těmi nejjednoduššími jazyky v hierarchii jazyků. Uváděli jsme si, že nad touto třídou jsou jazyky bezkontextové – mezi které patří například umělé programovací jazyky. Je tedy jasné, že pro některé jazyky, již nebude možno sestrojit konečný automat – nebudou regulární. Jejich charakteristika odpovídá tomu, čeho není schopen dosáhnout konečný automat. Nejtypičtějším příkladem je jazyk L = { 0^n1^n, kde $n \geq 0$}. Jde tedy o jazyk, který obsahuje na počátku jistý počet nul a za nimi následuje stejný počet jedniček. Aby bylo možné rozpoznat takový jazyk, musel by konečný automat umět „spočítat“ počet nul. Jenže to je pravě věc, kterou neumí. Jelikož má konečný počet stavů, nemůže si nijak pamatovat, kolik jich už přišlo. Uměli bychom sice udělat automaty postupně pro slova: ε,01,0011, ... , jenž problém je v tom, že při vytváření sjednocení takových automatů by nám vznikl automat s nekonečným počtem stavů, což není konečný automat.
Rozpoznat takový jazyk je možné díky charakterizace pomocí pravé kongruence a pomocí velmi důležité Nerodovy věty. Proto této větě věnujte stejnou pozornost jako například větě Kleeneho. Dává Vám totiž nástroj, jak dokazovat, že jazyk není regulární. Myšlenka, kterou jsme uvedli v předchozím odstavci je sice logická, ale nejde o exaktní důkaz. Ten Vám u všech příkladů poskytne právě Nerodova věta. Intuitivně lze samozřejmě říct, že jazyk který v sobě obsahuje jistou závislost (to je 0^n1^n – tedy závislost počtu nul a jedniček) není regulární.

Pokud chceme pochopit, co je pravá kongruence, podívejme se nejprve na rozdělení slov pomocí automatu A_1. Rozdělíme je do množin podle toho, do jakého stavu se z počátečního na ně dostaneme.



Jak A_1 rozděluje $\{a,b\}^*$?



q_3 odpovídající třída $A_3=\{w|w$ končí bab$\}$
q_2 odpovídající třída $A_2=\{w|w$ končí ba$\}$
q_1 odpovídající třída $A_1=\{w|w$ končí b, ale ne bab$\}$
q_0 odpovídající třída $A_0=\{w|w$ končí a, ale ne ba$\}\cup\{e\}$ (ostatní posloupnosti)
Můžeme pozorovat, že platí: $\forall u,v,w \in \{a,b\}^*$ platí:
jestliže $u,v \in A_i$ pro nějaké i ($i \in \{0,1,2,3\}$), pak $uw,vw \in A_j$ pro nějaké j ($j \in \{0,1,2,3\}$).

To znamená, že když nějaká dvě slova leží ve stejné třídě a přidáme k nim stejné slovo, pak tato zřetězená slova budou opět patřit do stejné třídy.

Neformální charakteristika jazyků rozpoznatelných konečnými automaty: Jazyk L je rozpoznatelný konečným automatem, právě tehdy když existuje konečný rozklad množiny Σ^* takový, že pro označení jeho tříd $A_1,A_2,\ldots A_n$ platí:

$\forall u,v,w \in \Sigma^*$ jestliže $u,v \in A_i$, pro nějaké i ($i \in \{1,2,\ldots n\}$), pak $uw,vw \in A_j$, pro nějaké j ($j \in \{1,2,\ldots n\}$)

přičemž L je sjednocením některých tříd tohoto rozkladu.

Definice 29: Nechť Σ je konečná abeceda a ~ je relace ekvivalence na Σ^*. Relace ~ se nazývá *pravou kongruencí* , jestliže
$\forall u,v,w \in \Sigma^*$: u ~ v$\Rightarrow$ uw ~ vw (1)

Věta 19: Nechť ~ je ekvivalence na Σ^*, kde Σ je konečná abeceda. Pak ~ je pravá kongruence právě tehdy, když
$\forall u,v \in \Sigma^*$, a $\in \Sigma$: u ~ v$\Rightarrow$ ua ~ va (2)
(neboli [u]=[v]$\Rightarrow$ [ua]=[va]).

Důkaz:
1. Jestliže ~ je pravá kongruence, pak platí (2) - triviální z definice.
2. Jestliže platí (2), pak platí (1) a tedy ~ je pravá kongruence; dokážeme indukcí podle délky w.
a) |w| = 0 ... triviální

b) předpokládejme, že (1) platí pro všechna w, $|w| \leq n$ ($n \geq 0$) (indukční předpoklad) a dokažme (1) pro $w' = w''a$, kde $|w''| = n$, $a \in \Sigma$.
Z indukčního předpokladu plyne $u \sim v \Rightarrow uw'' \sim vw''$, podle předpokladu (2) $uw'' \sim vw'' \Rightarrow uw'' a \sim vw'' a$, tedy $u \sim v \Rightarrow uw' \sim vw'$, čímž je vztah (1) dokázán pro všechna w, $|w| \leq n+1$.

Nerodova věta

Věta 20: (Nerode) Nechť L je jazyk nad konečnou abecedou Σ. Pak L je rozpoznatelný konečným automatem právě tehdy, když existuje pravá kongruence ~ na množině Σ^*, která je konečného indexu a pro níž platí, že L je sjednocením jistých tříd rozkladu $\Sigma^* / \sim$. (O relaci ekvivalence ~ na množině M říkáme, že je *konečného indexu* , jestliže rozklad M/ ~ má konečný počet tříd.)

Důkaz:
1. Nechť L je rozpoznáván automatem $A=(Q,\Sigma,\delta,q_0, F)$. Definujme relaci ekvivalence ~ na množině Σ^* předpisem $u \sim v \Leftrightarrow_{def.} \delta^*(q_0,u)=\delta^*(q_0,v)$ $\forall u,v \in \Sigma^*$. Relace ~ je konečného indexu vzhledem ke konečnosti množiny Q, navíc je pravou kongruencí, neboť z $\delta^*(q_0,u)=\delta^*(q_0,v)$ plyne $\delta^*(q_0,ua)=\delta^*(q_0,va)$ pro lib. $a \in \Sigma$ ($u \sim v \Rightarrow ua \sim va$). L je sjednocením jistých tříd rozkladu $\Sigma^* / \sim$, protože $L=\{ w \in \Sigma^* | \delta^*(q_0,w) \in F\} = \cup_{q \in F}\{ w \in \Sigma^* | \delta^*(q_0,w)=q\}$.
2. Nechť ~ je pravá kongruence konečného indexu na Σ^* a nechť existují třídy $A_1,A_2,... A_n$ rozkladu $\Sigma^* / \sim$ takové, že $L=\cup_{1 \leq i \leq n} A_i$. Definujme automat $A=(Q,\Sigma,\delta,q_0, F)$ následovně: $Q=\Sigma^* / \sim$, $q_0=[e]$, $F=\{ A_1,A_2,... A_n\}$ a δ je zadána předpisem $\delta([u],a)=[ua]$ pro všechna $u \in \Sigma^*$, $a \in \Sigma$. δ je definována korektně, protože pro všechna $u,v \in \Sigma^*$ platí: ([u]=[v] znamená, že $u \sim v$, odtud plyne $ua \sim va$, což znamená [ua]=[va] a tedy $\delta([u],a)=\delta([v],a)$). Zbývá ověřit, že $L(A)=L$.
$w \in L \Leftrightarrow (w \in A_1) \vee (w \in A_2) \vee ... \vee (w \in A_n) \Leftrightarrow ([w]=A_1) \vee ([w]=A_2) \vee ... \vee ([w]=A_n) \Leftrightarrow (\delta^*([e],w)=A_1) \vee ... \vee (\delta^*([e],w)=A_n) \Leftrightarrow \delta^*([e],w) \in \{ A_1,A_2,...,A_n\} \Leftrightarrow \delta^*([e],w) \in F \Leftrightarrow w \in L(A)$.

Důležité tedy je, že pokud jazyk je regulární, pak pro něj musí existovat pravá kongruence, která (což je nejdůležitější) rozkládá všechna slova do konečně mnoha tříd. Dále se podíváme, jak se tento přístup aplikuje při dokazování, že jazyky nejsou regulární.

3.4 Aplikace Nerodovy věty

Nerodovu větu lze pro dokazování, že jazyk není regulární, využít principem nepřímého důkazu, který již znáte z matematiky či logiky. Jde o to, že o jazyku předpokládáme, že je regulární a pak pro něj musí platit tvrzení Nerodovy věty. Předpokládáme tedy, že existuje pravá kongruence konečného indexu. Pak už jen stačí najít dvě slova, ke kterým přidáme třetí slovo, tak aby jedno ze zřetězených slov bylo z jazyka a druhé ne. To pak způsobí spor, neboť nemohou být dvě slova ze stejné třídy, pokud platí vlastnosti pravé kongruence. Blíže to uvidíte na následujících příkladech:

Kontrolní otázka:
Dokažte, že jazyk $L=\{0^n1^n;\ n \geq 0\}$ není rozpoznatelný konečným automatem.

Řešení:
Důkaz povedeme sporem. Předpokládáme tedy, že L je rozpoznatelný konečným automatem, tzn. podle Nerodovy věty existuje pravá kongruence ~ na množ. $\{0,1\}^*$, konečného indexu taková, že L je sjednocením jistých tříd $\{0,1\}^*/\sim$.

Odtud plyne, že rozklad $\{0,1\}^*/\sim$ má n_0 tříd.
Vezměme slova:
0, 00, 00, 000, ..., 0^{n_0+1} (slov jsme vybrali (n_0+1), je jich tedy o jedno více než je tříd rozkladu)
$\Rightarrow \exists i,j \in \{1,2,3,...,n_0+1\}$, $i \neq j$ tak, že $0^i \sim 0^j$,
(protože ~ je pravá kongruence) $\Rightarrow 0^i1^i \sim 0^j1^i$, a zde platí, že $0^i1^i \in L$, $0^j1^i \notin L$,
což je **spor**: dvě slova jsou kongruentní (patří do téže třídy rozkladu) a jedno z nich do L patří a druhé do L nepatří (spor s tím, že L je sjednocením jistých tříd rozkladu $\{0,1\}^*/\sim$).
$\Rightarrow L=\{0^n1^n;\ n \geq 0\}$ není rozpoznatelný KA.

Kontrolní otázka:
Dokažte, že jazyk $L=\{b^{2k}a^kc;\ k \geq 0\}$ není rozpoznatelný konečným automatem.

Řešení:
Důkaz povedeme sporem. Předpokládáme tedy, že L je rozpoznatelný konečným automatem, tzn. podle Nerodovy věty existuje pravá kongruence ~ na množ. $\{a,b,c\}^*$, konečného indexu taková, že L je sjednocením jistých tříd $\{a,b,c\}^*/\sim$.

Odtud plyne, že rozklad $\{a,b,c\}^*/\sim$ má n tříd.
Vezměme slova:
$b^2, b^4, b^6, b^8, ..., b^{2(n+1)}$ (slov jsme vybrali (n+1), je jich tedy o jedno více než je tříd rozkladu)
$\Rightarrow \exists i,j \in \{1,2,3,...,n+1\}$, $i \neq j$ tak, že $b^{2i} \sim b^{2j}$,
(protože ~ je pravá kongruence) $\Rightarrow b^{2i}a^ic \sim b^{2j}a^ic$, a zde platí, že $b^{2i}a^ic \in L$, $b^{2j}a^ic \notin L$,
což je **spor**: dvě slova jsou kongruentní a jedno z nich do L patří a druhé do L nepatří (spor s tím, že L je sjednocením jistých tříd rozkladu $\{a,b,c\}^*/\sim$).
$\Rightarrow L=\{b^{2k}a^kc;\ k \geq 0\}$ není rozpoznatelný KA.

Další možností jak dokazovat neregularitu jazyka je využití množinových operací a důkazů předchozích.

Řešený příklad 29:
Ukážeme, že jazyk $L_2=\{ w \in \{0,1\}^* | w$ obsahuje stejný počet nul a jedniček$\}$ není rozpoznatelný konečným automatem.

Důkaz: by bylo možné provést přímým užitím Nerodovy věty. Jestliže už ale víme, že jazyk

$L=\{ 0^i 1^i; i \geq 1\}$
není rozpoznatelný konečným automatem, lze důkaz vést jednodušeji.
Předpokládáme, že L_2 je rozpoznatelný konečným automatem. Protože $L_1=\{ 0^i 1^j; i,j \geq 1\}$ je rozpoznatelný konečným automatem, je podle věty 15 také $L_1 \cap L_2$ rozpoznatelný konečným automatem. Ale
$L_1 \cap L_2=\{ 0^n 1^n; n \geq 1\} = L$
To je spor s tím, že L není rozpoznatelný konečným automatem, a proto L_2 není rozpoznatelný konečným automatem.

Nejdůležitější probrané pojmy:

- regulární jazyk, regulární výraz
- ekvivalence regulárních jazyků a jazyků rozpoznatelných konečnými automaty (Kleeneho věta + důkaz)
- konstrukce ZNKA k regulárnímu výrazu
- pravá kongruence
- Nerodova věta
- aplikace Nerodovy věty při důkazech, že jazyk není regulární

Úkoly k textu:

1. Dokažte, že jazyk $L=\{b^{k+2}a^{2k}; k \geq 0\}$ není regulární.
2. Sestrojte regulární výraz rozpoznávající jazyk
$L=\{w \in \{a,b\}^* | w$ končí symbolem ‚a' nebo obsahuje ‚bab'$\}$.
3. Převeďte výraz z úkolu 1 na DKA.
4. Lze sestrojit regulární výraz ke každému konečnému automatu?

Korespondenční úkol:

Část 1:

a. Pro regulární jazyk L= [(a + ba*)*(b + c)*(cc)*] sestrojte DKA (libovolným způsobem).
b. Automat z bodu a. převeďte do normovaného redukovaného tvaru.

Část 2:

Navrhněte (jakýmkoliv postupem) konečné automaty, které rozpoznávají následující jazyky:
L1 = {w; w ∈ {0,1}*, w obsahuje lichý počet symbolů 0 a zároveň sudý (i nulový) počet symbolů 1}
L2= {w; w ∈ {0,1}*, w obsahuje posloupnost 011}
Sestrojte deterministický konečný automat, který rozpoznává průnik jazyků L1 a L2 a to pomocí algoritmu na průnik dvou automatů.

Část 3:

Lze přesně daným postupem pro libovolné 2 regulární jazyky zjistit, zda rozpoznávají stejný jazyk? Své tvrzení zdůvodněte.

4 Bezkontextové gramatiky a jazyky

Cíl:

Po prostudování této kapitoly pochopíte:

- co je bezkontextová gramatika
- jak pojem gramatiky souvisí s jazykem
- vztah regulárních gramatik k regulárním jazykům

Naučíte se:

- tvořit automaty pro jednoduché bezkontextové jazyky
- převádět regulární gramatiky na automaty a naopak

Průvodce studiem

Nyní se v našem studiu dostáváme do druhé části. V předchozích kapitolách jsme se zabývali třídou jazyků rozpoznatelných konečnými automaty resp. regulárními jazyky. Viděli jste, že existují i jazyky, které nejsou regulární. Konečné automaty jsou pro ně příliš „slabé", nedokáží je rozpoznávat a regulární výrazy je nemohou generovat. Proto by bylo rozumné se ptát, zda neexistují nástroje, které nám umožní tyto jazyky generovat a analyzovat. Takové nástroje existují a postupně se s nimi i s jejich vlastnostmi seznámíme a opět se naučíte vytvářet k jazykům jejich instance.

Složitější jazyky než regulární

Těmito nástroji jsou bezkontextová gramatika a zásobníkový automat. Pojem gramatiky jsme si již částečně objasnili na intuitivním příkladě v kapitole 1. Je to prostředek, jak na základě pravidel lze generovat (terminální) slova v jisté (terminální) abecedě pomocí postupného dosazování do neterminálních symbolů (proměnných). Občerstvěte si tyto pojmy v paměti. Zásobníkový automat je konečný automat, který má však navíc možnost pracovat s jistým druhem paměťového zařízení – zásobníkem, na který si může ukládat symboly a vybírat je zpět. Nicméně může tak činit pouze přístupem – poslední dovnitř, první ven (to znamená může zapisovat jen na vrchol zásobníku – struktura LIFO, jak ji znáte z algoritmizace). Naučíte se tyto struktury používat a také si ukážeme jejich speciální tvary. Stejně jako u regulárních jazyků si pak ukážeme, že jejich výpočetní síla – třída jazyků rozpoznatelných zásobníkovými automaty a bezkontextových jazyků generovaných bezkontextovými gramatikami – je totožná.

Podívejme se nejprve na příklad, jak se s bezkontextovými gramatikami pracuje a pak si zavedeme jejich formální definice. Uváděli jsme si, že jazyk

$L=\{0^n1^n; n \geq 0\}$ není rozpoznatelný konečným automatem. Existuje však velice jednoduchá bezkontextová gramatika, která tento jazyk generuje:

Tato gramatika má vstupní abecedu (terminálních symbolů) – {0,1}, abecedu proměnných (neterminálů) – {S}, neterminál, od kterého se generování (odvozování) vždy začíná určený jako S a tato dvě pravidla, určující možnosti „dosazení“ za proměnné:

$S \rightarrow 0S1$, $S \rightarrow \varepsilon$.

Tato pravidla umožňují buď dosadit za S řetězec 0S1 (obsahuje opět v sobě S), nebo ukončit toto generování slovem prázdným. Díky tomu, že vždy ke každé nule na levé straně slova dodáme jedničku na pravé straně slova, můžeme si takto „napumpovat“ kolik chceme nul a jedniček, ovšem jedině ve stejném počtu. Toto konečný automat ani regulární výraz neuměl. V gramatice pak můžeme provádět odvození terminálních slov (která budou všechna vytvářet jazyk), např. takto

$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011$

(v posledním kroku jsme použili pravidlo na prázdné slovo, čímž jsme se zbavili S a dostali slovo složené jen z terminálů).

4.1 Bezkontextová gramatika a bezkontextový jazyk

Jak uvidíme v poslední kapitole, pojem gramatiky lze zobecnit. Tím dosáhneme i daleko větší síly než poskytuje bezkontextová gramatika. Obecný pojem gramatiky:

Gramatika G je určena konečnou množinou *neterminálů* (neterminálních symbolů - proměnných), konečnou množinou *terminálů*, která nemá společné prvky s množinou neterminálů, *počátečním neterminálem* a konečnou *soustavou* (množinou) *přepisovacích pravidel* typu $\alpha \rightarrow \beta$, (α přepiš na β), kde α,β jsou řetězece z neterminálů a terminálů, navíc $\alpha \neq \varepsilon$.

Gramatika, označme ji G, může být chápána jako čtveřice $G=(\Pi,\Sigma,S,P)$

Π je množina neterminálů,

Σ je množina terminálů, $\Pi \cap \Sigma = \varnothing$

$S \in \Pi$ je počáteční neterminál,

P je konečná množina přepisovacích pravidel.

Bezkontextová gramatika se od tohoto obecného pojmu odlišuje tím, že připouští, aby přepisovací pravidlo mělo pouze tvar $X \rightarrow \alpha$, to znamená že pouze můžeme přepisovat vždy jednu proměnnou na řetězec proměnných i terminálů.

Bezkontextová gramatika

Definice 30: *Bezkontextová gramatika* (BKG) je určena konečnou množinou *neterminálů* (neterminálních symbolů - proměnných), konečnou množinou *terminálů*, která nemá společné prvky s množinou neterminálů, *počátečním neterminálem* a konečnou *soustavou* (množinou) *přepisovacích pravidel* typu $X \rightarrow \alpha$, (X přepiš na α), kde X je neterminál a α je řetězec z neterminálů a terminálů.

Bezkontextová gramatika, označme ji G, může být chápána jako čtveřice $G=(\Pi,\Sigma,S,P)$

Π je množina neterminálů,
Σ je množina terminálů, Π∩Σ = ∅
S ∈ Π je počáteční neterminál,
P je konečná množina přepisovacích pravidel.

V takto definované gramatice můžeme pak odvozovat slova, jak jsme viděli na příkladu.

Definice 31: Nechť G=(Π,Σ,S,P) je bezkontextová gramatika a nechť α,β ∈ (Π∪Σ)*.

Odvození v gramatice

Řekneme, že α ***se přímo přepíše na*** β podle pravidel gramatiky G, označíme $\alpha \Rightarrow_G \beta$ (nebo $\alpha \Rightarrow \beta$, pokud je zřejmé o jakou G se jedná), právě tehdy, když existuje $\gamma_1,\gamma_2,\delta \in (\Pi \cup \Sigma)^*$ a $X \in \Pi$ takové, že:
$\alpha = \gamma_1 X \gamma_2$; $\beta = \gamma_1 \delta \gamma_2$; $X \rightarrow \delta$ patří do P
Řekneme, že α ***se přepíše na*** β, značíme $\alpha \Rightarrow_G^* \beta$ (nebo $\alpha \Rightarrow^* \beta$), jestliže existuje posloupnost (*) $\gamma_0,\gamma_1,\gamma_2 \ldots \gamma_n$ prvků $(\Pi \cup \Sigma)^*$ (pro nějaké $n \geq 0$) taková, že:
$\alpha = \gamma_0 \Rightarrow_G \gamma_1 \Rightarrow_G \gamma_2 \Rightarrow_G \gamma_3 \Rightarrow_G \ldots \Rightarrow_G \gamma_{n-1} \Rightarrow_G \gamma_n = \beta$
Posloupnost (*) nazveme ***odvození*** (***derivace***) slova β ze slova α.
Jazyk generovaný gramatikou G, označme jej L(G), je definován následovně:
$L(G)=\{ w | w \in \Sigma^* \text{ a } S \Rightarrow_G^* w \}$

Stejně jako u konečných automatů, můžeme definovat pojem ekvivalentních gramatik. Opět půjde o gramatiku, která generuje stejný jazyk. Příkladem budiž ekvivalentní gramatika ke gramatice v příkladu:
$S \rightarrow ASB$, $S \rightarrow \varepsilon$, $A \rightarrow 0$, $B \rightarrow 1$ (přidali jsme neterminály A,B)

Definice 32: Bezkontextové gramatiky G_1,G_2 nazveme *ekvivalentní*, právě když $L(G_1)=L(G_2)$.

Definice 33: *Bezkontextový jazyk (BKJ)* je jazyk (jazyk $L \subseteq \Sigma^*$ pro nějakou konečnou abecedu Σ) generovaný nějakou bezkontextovou gramatikou (tedy L=L(G) pro nějakou bezkontextovou gramatiku G).

Ekvivalentní gramatika, Bezkontextový jazyk

Bezkontextový jazyk tedy tvoří všechna terminální slova, která můžeme odvodit z počátečního neterminálu.

Poznámka:
$\Rightarrow^*$ je reflefivní a tranzitivní uzávěr relace $\Rightarrow$.
$\alpha \Rightarrow_G^* \beta$ často čteme "α generuje β", "z α se odvodí β" apod.

odvození (*) nazveme minimální, jestliže $\gamma_i \neq \gamma_j \ \forall i \neq j$; je zřejmé, že když $\alpha \Rightarrow^* \beta$, pak lze β odvodit z α nějakým minimálním odvozením. Dále budeme odvozením (derivací) myslet minimální odvození.

Konvence

obvykle
jednotlivé terminály značíme - a,b,c...
řetězce terminálů značíme - u,v,w...
jednotlivé neterminály - A,B,C... X,Y,Z
řetězce neterminálů a terminálů - α,β,γ...

Prázdné slovo budeme někdy označovat také jako e, stejně jako tomu bylo již u automatů.

Poznámka: Bezkontextovou gramatiku obvykle budeme zadávat pouze množinou přepisovacích pravidel. Budeme-li neterminály označovat velkými písmeny, terminály jinak a počáteční neterminál S, pak budou všechny parametry gramatiky zřejmé.

4.2 Tvorba gramatik k bezkontextovým jazykům

Podívejme se na některé řešené příklady tvorby bezkontextových gramatik:

Řešený příklad 30:
Sestrojte bezkontextovou gramatiku G_i tak, aby $L(G_i)=L_i$; $1 \leq i \leq 7$

$L_1=\{w \in \{0,1\}^*; w=1^k0^k; k \geq 0\}$

Řešení: G_1:
$S \rightarrow 1S0$, $S \rightarrow e$.

Tento příklad nám ukazuje, jak lze realizovat konstrukci pro typický příklad jazyka, který není regulární, jak jsme poznali v předcházejících kapitolách.Chcete-li zaručit stejný počet symbolů na opačných stranách slova, pak je musíte v jednom pravidle uvést na příslušných místech od stejného neterminálu (to samozřejmě není jediný způsob – ale ostatní budou principiálně podobné).

Řešený příklad 31:
$L_2=\{w \in \{0,1\}^*; w=(011)^j101^k0^{k+1}; j,k \geq 0\}$

Řešení: G_2:
$S \rightarrow ABC$, $A \rightarrow 011A$, $A \rightarrow e$, $B \rightarrow 10$,$C \rightarrow 1C0$, $C \rightarrow 0$.

Vidíte, že stejně jako u automatů je někdy dobré si složitý problém rozdělit na podproblémy. Zde jsme si rozdělili generování celého slova na neterminály ABC. A generuje regulární jazyk A = [(011)*], regulární jazyk (jedno slovo) B = [10], bezkontextový jazyk, který již umíme generovat (téměř) C = $\{ 1^k0^{k+1}, k \geq 0\}$. Poslední zmiňovaný jazyk je jednoduchou modifikací z ilustrativního

příkladu z počátku kapitoly. Liší se tím, že generování nekončí prázdným slovem, ale symbolem 0, který má být na pravé straně vždy navíc.

Řešený příklad 32:
$L_3=\{w \in \{0,1\}^*; w=1uu^R0;u \in \{0,1\}^+\}$

Řešení: G_3:
$S\rightarrow 1A0, A\rightarrow 1B1, A\rightarrow 0B0, B\rightarrow e,B\rightarrow 1B1, B\rightarrow 0B0.$

Pokud chcete zajistit jako v tomto případě, aby se rozpoznával zrcadlový obraz (tedy čím u začíná tím u^R končí, atd...), pak stačí v pravidlech vždy ke stejnému symbolu na počátku vygenerujeme stejný na konci. Takto se nám napumpují na bocích zrcadlově stejná slova.

Řešený příklad 33:
$L_4=\{w \in \{a,b\}^*; w=aba^* \text{ nebo } w=(ab)^*bab\}$

Řešení: G_4:
$S\rightarrow A, S\rightarrow B, A\rightarrow ab, A\rightarrow Aa,B\rightarrow bab, B\rightarrow abB.$

Opět jde o případ dekompozice problému. Máme zde dvě podmínky a stačí, aby byla splněna jedna z nich. Jinak řečeno, vygeneruje buď slovo podle 1. nebo 2. podmínky. Proto už od začátku můžeme tyto dvě alternativní cesty v gramatice zohlednit – tedy S se přepisuje buď na A nebo B.

4.3 Regulární gramatiky, vztah k regulárním jazykům

Bezkontextové jazyky lze dále omezit až na přesně třídu regulárních jazyků (jinak řečeno za jistých podmínek BKG generují jazyky rozpoznatelné KA). Stačí omezíme-li pravidla BKG na taková, která přepisují neterminál na slovo terminálů a za ním následuje maximálně jeden neterminál. Taková formalizace odpovídá tomu, co rozpoznává KA. Uvidíte, jak se dá velmi jednoduše ke KA sestrojit regulární gramatika a naopak. Vychází se z toho, že můžeme stavy konečného automatu považovat za neterminály, symboly u přechodů za začátek přepisovaného slova a stav, do kterého se jde, za neterminál za terminálním slovem. Je-li stav výstupní, generování slova může skončit a proto se tento stav (neterminál) přepíše na ε.

Definice 34: Bezkontextová gramatika G=(Π,Σ,S,P) se nazývá *regulární gramatika*, jestliže každé pravidlo v P je v jednom z tvarů $X \rightarrow wY$, $X \rightarrow w$, kde $X,Y \in \Pi$, $w \in \Sigma^*$.

***Jazyk* L nazveme *regulární*, jestliže je generován nějakou regulární gramatikou (tedy L=L(G) pro nějakou regulární gramatiku G).**

Regulární gramatika

Ukážeme, že definice nekoliduje s dřívější definicí regulárního jazyka.

Věta 21: Každý jazyk rozpoznatelný konečným automatem je regulární (ve smyslu definice pomocí regulární gramatiky).

Jazyky generované regulárními gramatikami

Důkaz:

1.

Mějme libovolný jazyk, označme jej L. Předpokládejme, že jazyk L je rozpoznáván nějakým konečným automatem, označme jej A. Ukážeme jak k automatu A zkonstruovat regulární gramatiku, označme ji G, která generuje jazyk L, tím bude důkaz hotov.

Mějme tedy nějak zadán automat A. Je tedy nějakým způsobem určena (vymezena) množina jeho stavů, dále je určena množina vstupních symbolů, jeden stav je označen za počáteční, některé stavy za koncové. Přitom je zadán předpis, který pro každý stav a každý vstupní symbol udává, do kterého stavu automat A přejde přečtením onoho vstupního symbolu nachází-li se v onom stavu.

Způsob převodu KA na BKG

Pro názornost uvedeme příklad, kde A je zadán tabulkou:

	$Q\backslash\Sigma$	0	1
↔	1	1	2
	2	1	3
	3	1	3

Budeme konstruovat gramatiku G.

Nejprve označme všechny stavy automatu A např. písmeny $A_1,A_2,\ldots A_n$, kde n je počet stavů.

V našem příkladu tedy:

	$Q\backslash\Sigma$	0	1
↔	A_1	A_1	A_2
	A_2	A_1	A_3
	A_3	A_1	A_3

Symboly $A_1,A_2,\ldots A_n$ budou tvořit množinu neterminálů gramatiky G. Symbol označující počáteční stav, bude počátečním neterminálem gramatiky G, v našem příkladu tedy A_1. Množina terminálů gramatiky G bude totožná se vstupní abecedou automatu A ({ 0,1} v našem příkladu).

Množinu přepisovacích pravidel konstruujeme následovně.

Vezmeme symbol A_1. Dále vezmeme některý terminál, označme jej a a zjistíme stav, do kterého automat A přejde ze stavu A_1 přečtením terminálu a. Tento stav nechť je označen A_j. Pak mezi přepisovací pravidla zahrneme pravidlo $A_1 \rightarrow aA_j$.

V příkladu pro a=0 dostaneme pravidlo $A_1 \rightarrow 0A_1$. Totéž provedeme pro všechny ostatní terminály (a symbol A_1). V příkladu tedy ještě přidáme

pravidlo $A_1 \rightarrow 1A_2$. Pak celý postup opakujeme pro A_2, pak pro A_3 atd., až pro A_n.

V příkladu tedy takto vytvoříme pravidla:
$A_1 \rightarrow 0A_1$, $A_1 \rightarrow 1A_2$, $A_2 \rightarrow 0A_1$, $A_2 \rightarrow 1A_3$, $A_3 \rightarrow 0A_1$, $A_3 \rightarrow 1A_3$.
Nakonec přidáme pravidla, která umožňují smazat, neboli přepsat na prázdné slovo, všechny ty neterminály, které označují koncové stavy automatu A. V příkladu tedy přidáme jen jedno pravidlo, a to $A_1 \rightarrow e$.
Tím jsme vytváření přepisovacích pravidel, a tím i celé gramatiky G ukončili.
Všimněme si, že každému "výpočtu" automatu A znázorněnému
$A_{i0} \rightarrow^{a}{}_1 A_{i1} \rightarrow^{a}{}_2 A_{i2} \rightarrow^{a}{}_3 ... \rightarrow^{a}{}_m A_{im}$
odpovídá v gramatice G odvození
$A_{i0} \Rightarrow a_1 A_{i1} \Rightarrow a_1 a_2 A_{i2} \Rightarrow ... \Rightarrow a_1 a_2 a_3 ... a_m A_{im}$
V příkladu např.
$A_1 \rightarrow^{0} A_1 \rightarrow^{1} A_2 \rightarrow^{1} A_3$

$A_1 \Rightarrow 0 A_1 \Rightarrow 01 A_2 \Rightarrow 011 A_3$
Když si nyní uvědomíme, že počáteční neterminál gramatiky G odpovídá počátečnímu stavu automatu A, a dále, že smazat lze jedině neterminály odpovídající koncovým stavům, je zřejmé, že každý přijímající výpočet automatu A (pro nějaké slovo, označme ho w) odpovídá odvození slova w z počátečního neterminálu v gramatice G a naopak. Tím jsme se přesvědčili, že gramatika G skutečně generuje jazyk L (rozpoznávaný automatem A).
2.(formálně zapsáno)
Nechť $L=L(A)$ pro konečný automat $A=(Q,\Sigma,\delta,q_0,F)$. Sestrojme gramatiku $G=(Q,\Sigma,q_0,P)$, kde $P=\{ q \rightarrow aq' | \delta(q,a)=q' \} \cup \{ q \rightarrow e \,|\, q \in F \}$.
Snadno lze ověřit, že pro libovolné $w \in \Sigma^*$ platí $w \in L(A) \Leftrightarrow (q_0 \Rightarrow_G^* wq$ pro něj. $q \in F) \Leftrightarrow (q_0 \Rightarrow_G^* w) \Leftrightarrow w \in L(G)$.
Stejně jako lze k automatu sestrojit gramatiku, lze i ke gramatice sestrojit automat. Postup je v podstatě reverzí postupu, který jste se právě naučili. Jelikož však v obecné regulární gramatice jsou celá slova, která bychom nemohli do přechodů přímo zapsat, je třeba formulovat tvrzení, že každá taková gramatika se může převést na gramatiku s pravidly, kde je nejvýše jeden terminální symbol před neterminálem.

Věta 22: Ke každé regulární gramatice existuje ekvivalentní regulární gramatika, která má pravidla pouze následující typů:
$X \rightarrow aY$, $X \rightarrow Y$, $X \rightarrow e$
kde X,Y jsou neterminály a a je terminál.

Důkaz:
Nechť $G=(\Pi,\Sigma,S,P)$ je regulární gramatika.
Sestrojíme $G'=(\Pi',\Sigma,S,P')$ následovně:
Do P' zahrneme všechna pravidla z P, která jsou v jednom z povolených typů:
$X \rightarrow aY$, $X \rightarrow Y$, $X \rightarrow e$

Místo každého pravidla z P tvaru $X\rightarrow a_1\ a_2...a_mY$ ($m \geq 2$) zahrneme do P′ soustavu pravidel $X\rightarrow a_1\ Y_1$, $Y_1\rightarrow a_2\ Y_2$, $Y_2\rightarrow a_3\ Y_3$ ... $Y_{m-2}\rightarrow a_{m-1}Y_{m-1}$, $Y_{m-1}\rightarrow a_m\ Y$, kde $Y_1,Y_2,Y_3...\ Y_{m-1}$ jsou nově přidané neterminály.
Místo každého pravidla typu $X\rightarrow a_1\ a_2\ ...\ a_n$ ($n \geq 2$) zahrneme do P′ soustavu pravidel $X\rightarrow a_1\ Y_1$, $Y_1\rightarrow a_2\ Y_2$, $Y_2\rightarrow a_3\ Y_3$ ... $Y_{n-1}\rightarrow a_n\ Y_n$, $Y_n\rightarrow e$, kde $Y_1,Y_2,Y_3...\ Y_n$ jsou nově přidané neterminály.
Π′ obsahuje neterminály z Π a všechny nově přidané neterminály.
Je zřejmé, že G′ je požadovaného typu a také lze snadno ověřit, že L(G)=L(G′).

Regulární gramatika -> KA

Věta 23: Každý regulární jazyk (ve smyslu definice podle regulární gramatiky) je rozpoznatelný konečným automatem.

Důkaz:
Nechť L=L(G) pro regulární gramatiku G=(Π,Σ,S,P). Podle předchozí věty, lze předpokládat, že pravidla G jsou pouze typů

$X\rightarrow aY$, $X\rightarrow Y$, $X\rightarrow e$
Sestrojme zobecněný nedeterministický konečný automat A=(Q,Σ,δ,I,F), kde Q=Π; I={S}, F={ q|(q→ e) ∈ P} a ∀q ∈ Q(=Π), ∀a ∈ Σ je δ(q,a)={ q′|(q → aq′) ∈ P} a δ(q,e)={ q′|(q → q′) ∈ P}.
Ověření L(G)=L(A) je podobné jako u důkazu věty opačné.

Tento reverzibilní postup nyní demonstrujeme na tomto kontrolním úkolu:

Kontrolní úkoly:
Sestrojte gramatiky pro následující jazyky:
$L_5=\{w \in \{a,b,c\}^*;\ w=a^ib^{i+2}uabcu^R;\ i \geq 1;\ u \in \{a,b\}^*\}$

$L_6=\{w \in \{0,1\}^*;\ w=(011)^i(110)^j(11)^{2j};\ i,j \geq 0\}$

$L_7=\{w \in \{a,b\}^*;\ w=(ab)^k(bab)^j;\ j \geq 0;\ k \geq j\}$

Řešení:
G_5: S→ AB, A→ abbb, A→ aAb, B→ abc,B→ aBa, B→ bBb.
G_6: S→ AB, A→ 011A, A→ e, B→ e,B→ 110B1111.
G_7: S→ abSbab, S→ abS, S→ e.

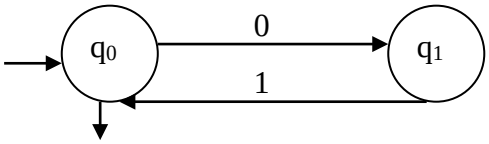
Kontrolní úkol:

Mějme regulární gramatiku:
S→ 01S, S→ ε, která rozpoznává jazyk $L = \{(01)^n\ , n \geq 0\}$.

Abychom mohli tuto gramatiku převést na automat, musíme nejprve postupem dle důkazu věty ji převést (na tvar X→ aY, X→ Y, X→ e).

$S \rightarrow 0A$, $A \rightarrow 1S$, $S \rightarrow \varepsilon$,

Automat pak vypadá takto:



Z tohoto automatu pak můžeme zpětně zase dojít k této gramatice.

Viděli jste, že BKG může mít jisté omezení pravidel, které ovlivňuje jazyky, které taková gramatiky generují. Například jazyk $L = \{ 0^n1^n , n \geq 0\}$ logicky nemůžeme generovat regulární gramatikou, neboť není regulární. Dalším tvarem pravidel je levá lineární gramatika. Její odvozovací síla je opět stejná jako u regulární, neboť pravidla si lze představit jako zrcadlové obrazy, ke kterým je lehké sestrojit automat rozpoznávající zrcadlový obraz.

Definice 35: ***Levá lineární gramatika*** **je bezkontextová gramatika, jejíž pravidla jsou pouze typu: $X \rightarrow Yw$ nebo $X \rightarrow w$, kde X,Y jsou neterminály a w je řetězec terminálů.**

Levá lineární gramatika

Věta 24: **Jazyk je regulární, právě když je generován nějakou levou lineární gramatikou.**

Důkaz:
Nechť $G=(\Pi,\Sigma,S,P)$ je levá lineární gramatika. Sestrojme gramatiku $G'=(\Pi,\Sigma,S,P')$ tak, že platí $X \rightarrow \alpha \in P \Leftrightarrow X \rightarrow \alpha^R \in P'$ $(\forall X \in \Pi, \alpha \in (\Pi \cup \Sigma^*))$
Je zřejmé, že $L(G)=(L(G'))^R$, přitom G' je regulární (pravá lineární). Věta tedy plyne z uzavřenosti třídy regulárních jazyků vůči zrcadlovému obrazu.

Dalším omezením je lineární gramatika, která již nemá stejnou sílu jako regulární. Například právě pro $L = \{ 0^n1^n , n \geq 0\}$ lze sestrojit gramatiku lineární, ale nikoliv regulární.

Lineární gramatika

Definice 36: ***Lineární gramatika*** **je bezkontextová gramatika, jejíž pravidla jsou pouze typu $X \rightarrow uYv$ nebo $X \rightarrow u$, kde X,Y jsou neterminály a u,v jsou řetězce terminálů.**

Jazyk je *lineární*, je-li generován nějakou lineární gramatikou.

Věta 25: **Třída regulárních jazyků je vlastní podtřídou třídy lineárních jazyků.**

Důkaz:
Že je podtřídou je zřejmé z definice, že je vlastní podtřídou, ukazuje např. jazyk $\{ 0^n 1^n | n \geq 0\}$ $(S \rightarrow 0S1|e)$.

Poznámka: Ukážeme, že se obejdeme bez $X \rightarrow e$ - vypouštějícího pravidla.

Řešený příklad 34:
Sestrojte regulární gramatiku G tak, aby
a)L(G)=[$a^*b+(bab)^*bb$]

Řešení:
G:
S→ A, S→ B, A→ aA, A→ b, B→ babB, B→ bb.

b)L(G)=[$((00)^*+(11)^*)101(110)^*$]

Řešení:
G:
S→ A, S→ B, A→ 00A, A→ C, B→ 11B, B→ C, C→ 101D, D→ 110D, D→ e.

c)L(G)=[$abab^*+((bb)^*a)^*$]

Řešení:
G:
S→ A, S→ B, A→ abaC, B→ e, B→ D, C→ bC, C→ e, D→ bbD, D→ aE, E→ e, E→ D.

Řešený příklad 35:
Určete jazyk, který generuje bezkontextová gramatika G.
a) G:S→ 11S0, S→ e.

Řešení:
L(G)={w ∈ {0,1}*; w=$(11)^j0^j$; j ≥ 0}=
={w ∈ {0,1}*; w=$1^{2j}0^j$; j ≥ 0}

b) G:S→ ABC, A→ bbA, A→ ccB,B→ bBb, B→ a, C→ aCa, C→ bb.

Řešení:
L(G)={w ∈ {a,b,c}*; w=$(bb)^iccb^jab^jb^kab^ka^mbba^m$; i,j,k,m ≥ 0}

c) G:S→ 001, S→ 01S, S→ 01S1.

Řešení:
L(G)={w ∈ {0,1}*; w=$(01)^j001(1)^i$; i ≥ 0;j ≥ i}

4.4 Nevypouštějící a redukované gramatiky

Bezkontextové gramatiky mohou mít své speciální tvary. Tyto speciální tvary nemusejí porušovat třídu jazyků, které rozpoznávají obecné BKG a mohou mít

některé výhodné vlastnosti. Například prázdné slovo v pravidlech může být někdy na obtíž a působit komplikace při převodech. Na druhou stranu v praxi jsou e-pravidla někdy užitečná (usnadňují zápis a pochopitelnost gramatiky), proto se nedá obecně říct, že by jejich výskyt byl nežádoucí. Ukážeme si, že lze formulovat tvar bez e-pravidel a každou gramatiku lze do tohoto tvaru převést. Princip spočívá v tom, že odhalíme ty neterminály, které se přepisují na e a ty pak v ostatních pravidlech vynecháme (všechny kombinace i s původním pravidlem).

Definice 37: Bezkontextová gramatika se nazývá *nevypouštějící,* jestliže neobsahuje žádné pravidlo typu $X \rightarrow e$, kde X je neterminál.

Věta 26: Ke každé bezkontextové gramatice G lze zkonstruovat nevypouštějící gramatiku G′ takovou, že $L(G')=L(G)-\{e\}$.

Důkaz:

Nevypouštějící gramatika

Mějme $G=(\Pi,\Sigma,S,P)$. Zkonstruujme množinu U; $U=\{X \in \Pi | X \Rightarrow_G^* e\}$. U lze zkonstruuovat např. následovně. Položme $U_1=\{X \in \Pi | (X \rightarrow e) \in P\}$, obecně $U_{i+1}=U_i \cup \{X \in \Pi | X \rightarrow \alpha$, kde $\alpha \in U_i^*$, je pravidlo v $P\}(i \geq 1)$.
Zřejmě je $U_1 \subseteq U_2 \subseteq U_3 \subseteq ... \subseteq \Pi$. Pro nějaké n je $U_n=U_{n+1}$ a podle konstrukce je pak zřejmé, že $U_n=U_{n+k}$ pro libovolné $k \geq 0$. Tedy $U=U_n$.
Zkonstruujme gramatiku $G_1=(\Pi,\Sigma,S,P_1)$ následovně: v P_1 jsou právě taková pravidla $X \rightarrow \alpha$, pro která $\alpha \neq e$, přičemž pro každé takové $X \rightarrow \alpha$ existuje v P pravidlo $X \rightarrow \beta$, kde α vznikne z β vynecháním některých (třeba žádných) výskytů neterminálů z množiny U.
Chceme ukázat, že $L(G_1)=L(G)-\{e\}$.

Ukažme nejdříve $L(G_1) \subseteq L(G)$. Jestliže G_1 vygeneruje w, pak w je generováno i gramatikou G. Nová pravidla můžeme simulovat starými, přičemž používáme generování prázdného slova. Jelikož $e \notin L(G_1)$, pak je zřejmé, že $L(G_1) \subseteq L(G)-\{e\}$.
Nyní předpokládáme, že $w \neq e$ je odvozeno v gramatice G. Fakt, že w lze odvodit i v G_1 je zřejmý z toho, že výskyty neterminálů, které se v odvození podle G přepíší na prázdné slovo, může odvození v G_1 rovnou vynechat.

Věta 27: Ke každé bezkontextové gramatice $G=(\Pi,\Sigma,S,P)$ existuje ekvivalentní gramatika $G'=(\Pi \cup \{S'\},\Sigma,S',P')$ taková, že e se může vyskytovat na pravé straně jedině v pravidle $S' \rightarrow e$, přičemž S′ se nevyskytuje na pravé straně žádného pravidla P′.

Důkaz:
Jestliže $e \notin L(G)$, pak je tvrzení zřejmé z předchozí věty. Jestliže $e \in L(G)$, postupujeme následovně:
Vezměme $G_1=(\Pi,\Sigma,S,P_1)$ tak jako v důkazu předchozí věty. Nyní stačí položit $G'=(\Pi \cup \{S'\},\Sigma,S',P_1 \cup \{S' \rightarrow S, S' \rightarrow e\})$.

Dalším speciálním tvarem je redukovaná gramatika. Stejně jako u automatů, mohou i v gramatice existovat zbytečné neterminály (stavy). Jde v zásadě o dva případy. Buď některý neterminál nemůže vygenerovat žádné slovo

například se cyklí sám v sobě nebo se na některý neterminál nedá vůbec dostat z počátečního S. Opět lze každou gramatiku na tento tvar převést. V první fázi hledáme ty „neproduktivní“ neterminály (vyjdeme od těch, které se přímo přepisují na terminální slovo) a v druhé ty „nedosažitelné“ (princip je podobný hledání nedosažitelných stavů automatu).

Definice 38: Bezkontextová gramatika G=(Π,Σ,S,P) se nazývá *redukovaná*, jestliže platí následující dvě podmínky:
1. Pro každé $X \in \Pi$ existuje $w \in \Sigma^*$ takové, že $X \Rightarrow_G^* w$.
2. Pro každé $X \in \Pi$ existují $\alpha,\beta \in (\Pi \cup \Sigma)^*$ takové, že $S \Rightarrow_G^* \alpha X \beta$.

Věta 28: Ke každé bezkontextové gramatice G takové, že $L(G) \neq \emptyset$, lze zkonstruovat ekvivalentní redukovanou gramatiku.

Důkaz:
Nechť G=(Π,Σ,S,P).

Redukovaná gramatika

1. Zkonstruujeme množinu U; $U=\{X \in \Pi | X \Rightarrow_G^* w$ pro nějaké $w \in \Sigma^*\}$. U lze zkonstruovat např. takto: položme $U_0=\Sigma$, $U_{i+1}=U_i \cup \{X \in \Pi | (X \rightarrow \alpha) \in P$ pro nějaké $\alpha \in U_i^*\}$, $i \geq 0$; $U_0 \subseteq U_1 \subseteq U_2 \subseteq U_3 \subseteq ... \subseteq \Pi \cup \Sigma$. Když $U_n=U_{n+1}$ pak $\forall k \geq 0\ U_n=U_{n+k}$. Stačí vzít $U=U_n - \Sigma$.
Z pravidel gramatiky G vyhodíme všechna pravidla, obsahující nějaký neterminál z Π– U. Tím obdržíme gramatiku G′ takovou, že L(G)=L(G′). Navíc G′ splňuje vlastnost 1. z definice redukované gramatiky.
2. Zkonstruujeme množinu V; $V=\{X \in U | \exists \alpha,\beta \in (U \cup \Sigma)^*$ tak, že $S \Rightarrow_G^* \alpha X \beta\}$. Z pravidel gramatiky G′ odstraníme všechna pravidla obsahující nějaký neterminál z U – V. Tím dostaneme gramatiku G″ takovou, že L(G)=L(G″). Navíc G″ splňuje podmínku 2. z definice redukované gramatiky a přitom se neporušila platnost bodu 1.
(G″ splňuje 1. i 2. a je tedy redukovaná.)

Z následujících tvrzení vyplývá, že lze zjistit zda gramatika generuje prázdný jazyk. Pokud ji převedeme na redukovanou, zjistíme jednoduše, zda obsahuje nějaký neterminál nebo ne.

Věta 29: Existuje algoritmus, který pro libovolnou bezkontextovou gramatiku G rozhodne, zda $L(G)=\emptyset$.

Rozhodnutelnost $L(G)=\emptyset$

Důkaz:
Důsledek části 1. předchozího důkazu.

Řešený příklad 36:

Sestrojte bezkontextovou gramatiku G_1 tak, aby $L(G_1)=L(G)-\{e\}$ a aby G_1 byla nevypouštějící.
a)G:
$S \rightarrow ABC$, $A \rightarrow 011A$, $A \rightarrow e$, $B \rightarrow 10$, $C \rightarrow 1C0$, $C \rightarrow 0$.

Řešení: $U_1=\{A\}$, $U_2=\{A\}$, $U=\{A\}$.
Pak G_1 bude vypadat takto:
$S\rightarrow ABC$, $S\rightarrow BC$, $A\rightarrow 011A$, $A\rightarrow 011$,$B\rightarrow 10$, $C\rightarrow 1C0$, $C\rightarrow 0$.

b)G:
$S\rightarrow AB$, $A\rightarrow 011A$, $A\rightarrow e$, $B\rightarrow e$,$B\rightarrow 110B1111$.

Řešení: $U_1=\{A,B\}$, $U_2=\{A,B,S\}$, $U_3=\{A,B,S\}$, $U=\{A,B,S\}$.
Pak G_1 bude vypadat takto:
$S\rightarrow AB$, $S\rightarrow B$, $S\rightarrow A$, $A\rightarrow 011A$, $A\rightarrow 011$, $B\rightarrow 110B1111$, $B\rightarrow 1101111$.

c)G:
$S\rightarrow AB$, $A\rightarrow abbb$, $A\rightarrow aAb$, $B\rightarrow abc$,$B\rightarrow aBa$, $B\rightarrow bBb$.
Řešení: Gramatika je nevypouštějící. $G_1=G$.

d)G:
$S\rightarrow A$, $S\rightarrow B$, $A\rightarrow abaC$, $B\rightarrow e$, $B\rightarrow D$, $C\rightarrow bC$, $C\rightarrow e$, $D\rightarrow bbD$,$D\rightarrow aE$, $E\rightarrow e$, $E\rightarrow D$.

Řešení: $U_1=\{B,C,E\}$, $U_2=\{B,C,E,S\}$, $U_3=\{B,C,E,S\}$, $U=\{B,C,E,S\}$.
Pak G_1 bude vypadat takto:
$S\rightarrow A$, $S\rightarrow B$, $A\rightarrow abaC$, $A\rightarrow aba$, $B\rightarrow D$, $C\rightarrow bC$, $C\rightarrow b$, $D\rightarrow bbD$,$D\rightarrow aE$, $D\rightarrow a$, $E\rightarrow D$.

Řešený příklad 37:

Sestrojte ekvivalentní redukovanou bezkontextovou gramatiku G_r k bezkontextové gramatice G.
a) G:$S\rightarrow ABC$, $S\rightarrow a$, $A\rightarrow aA$,$A\rightarrow aB$, $B\rightarrow bBb$, $B\rightarrow Bb$, $C\rightarrow cAB$, $C\rightarrow c$.

Řešení:
$U_0=\{a,b,c\}$, $U_1=\{a,b,c,S,C\}$, $U_2=\{a,b,c,S,C\}$, $U=\{S,C\}$
G_t:$S\rightarrow a$, $C\rightarrow c$.
$V=\{S\}$
G_r:$S\rightarrow a$.
(A platí:$L(G)=L(G_t)=L(G_r)=\{a\}$.)

b) G:$S\rightarrow XY$, $S\rightarrow YZ$, $X\rightarrow aX$,$X\rightarrow e$, $Y\rightarrow bYb$, $Y\rightarrow X$, $Z\rightarrow bZ$, $M\rightarrow b$, $M\rightarrow bMc$.

Řešení:
$U_0=\{a,b,c\}$, $U_1=\{a,b,c,X,M\}$, $U_2=\{a,b,c,X,M,Y\}$, $U_3=\{a,b,c,X,M,Y,S\}$, $U_4=U_3$, $U=\{X,M,Y,S\}$
G_t:$S\rightarrow XY$, $X\rightarrow aX$,$X\rightarrow e$, $Y\rightarrow bYb$, $Y\rightarrow X$, $M\rightarrow b$, $M\rightarrow bMc$.
$V=\{S,X,Y\}$
G_r:$S\rightarrow XY$, $X\rightarrow aX$,$X\rightarrow e$, $Y\rightarrow bYb$, $Y\rightarrow X$.
(A platí:$L(G)=L(G_t)=L(G_r)=\{a^m b^j a^k b^j; m,j,k \geq 0\}$.)

4.5 Kanonická odvození, jednoznačné gramatiky

Odvození v lineární textové podobě není jediná možnost, jak ho reprezentovat. Další možnost je vytvořit strom odvození. Odvození však u některých gramatik pro stejné slovo může být různé. Mluvíme pak o nejednoznačných gramatikách.

Často je výhodné omezit se na tzv. *kanonické derivace*, tj. levé nebo pravé derivace. Nejde o nic jiného než o konvenci, kterou dodržujeme během celého odvození v gramatice. Buď se rozhodneme, že vždy přepíšeme neterminál nejvíce vpravo v odvozovaném slově nebo ten nejvíce vlevo.

Definice 39: Odvození v bezkontextové gramatice se nazývá *levé odvození* (levá derivace), jestliže se v něm přepisuje vždy nejlevější neterminál. Odvození v bezkontextové gramatice se nazývá *pravé odvození* (pravá derivace), jestliže se v něm přepisuje vždy nejpravější neterminál.

Levé a pravé odvození (kanonická)

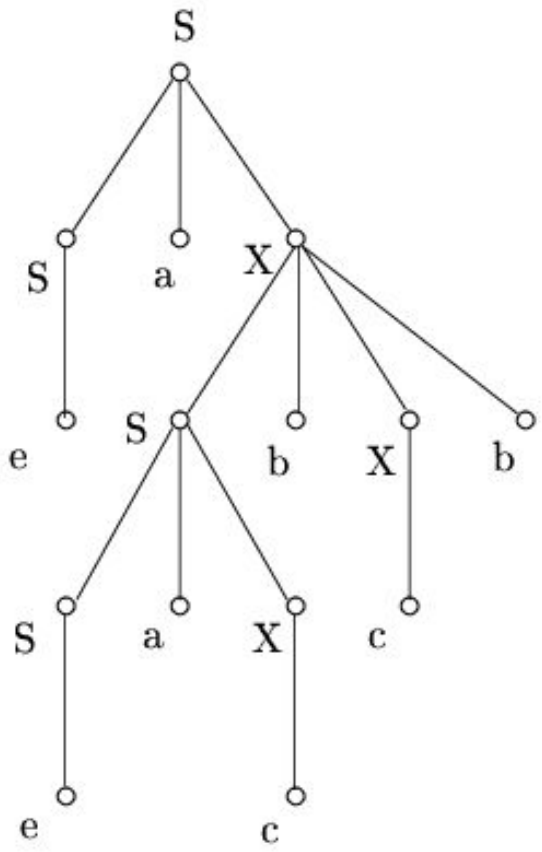
Podrobněji: Nechť G=(Π,Σ,S,P) je bezkontextová gramatika. Řekneme, že α se *přepíše levým přepsáním* na β ($\alpha,\beta \in (\Pi\cup\Sigma)^*$), jestliže existuje $X\rightarrow\gamma \in P$ takové, že $\alpha = uX\delta$, $\beta = u\gamma\delta$ ($u \in \Sigma^*$, $\delta \in (\Pi\cup\Sigma)^*$).
O odvození $\alpha_0\Rightarrow\alpha_1\Rightarrow\alpha_2\Rightarrow...\Rightarrow\alpha_n$ řekneme, že je to *levá derivace* jestliže α_i se přepíše na α_{i+1} levým přepsáním pro všechna i=0,1,2... n−1.
Podobně pravá derivace:
Nechť G=(Π,Σ,S,P) je bezkontextová gramatika. Řekneme, že α se *přepíše pravým přepsáním* na β ($\alpha,\beta \in (\Pi\cup\Sigma)^*$), jestliže existuje $X\rightarrow\gamma \in P$ takové, že $\alpha = \delta Xu$, $\beta = \delta\gamma u$ ($u \in \Sigma^*$, $\delta \in (\Pi\cup\Sigma)^*$).
O odvození $\alpha_0\Rightarrow\alpha_1\Rightarrow\alpha_2\Rightarrow...\Rightarrow\alpha_n$ řekneme, že je to *pravá derivace* jestliže α_i se přepíše na α_{i+1} pravým přepsáním pro všechna i=0,1,2... n−1.

Věta 30: Nechť G=(Π,Σ,S,P) je bezkontextová gramatika. Jestliže $X\Rightarrow_G^* w$ ($X \in \Pi, w \in \Sigma^*$), pak w je z X odvoditelné nějakým levým (pravým) odvozením.

Derivační strom

Poznámka: Ke každému odvození slova z jazyka generovaného gramatikou, existuje derivační strom daného odvození. Nebudeme uvádět přesnou definici derivačního stromu - zůstane na intuitivní úrovni. Kořenem stromu je počáteční neterminál, potomky každého uzlu jsou symboly z řetězce, na který se přepsal rodič. Listy stromu jsou pak terminální symboly (resp. prázdný řetězec).

Derivační strom na obr. je derivačním stromem např. odvození S⇒ SaX⇒ aX⇒ aSbXb⇒aSaXbXb⇒ aaXbXb⇒ aacbXb⇒ aacbcb v gramatice G obsahující určitě alespoň pravidla S→ SaX|e a pravidla X→ SbXb|c, ale tento derivační strom je taktéž derivačním stromem odvození S⇒ SaX⇒ SaSbXb⇒ aSbXb⇒aSaXbXb⇒ aaXbXb⇒ aacbXb⇒ aacbcb



Definice 40: Bezkontextová gramatika je *nejednoznačná,* jestliže pro některé slovo w ∈ L(G) existují dvě různé levé derivace (dva různé derivační stromy). V opačném případě je gramatika *jednoznačná*.

Nejednoznačná gramatika a jazyk

Poznámka: Bezkontextový jazyk, který nelze nagenerovat jednoznačnou gramatikou se nazývá ***vnitřně nejednoznačný*.**

4.6 Věta o vkládání (pumping lemma)

V této podkapitole si ukážeme, jak dokazovat, že jazyky nejsou bezkontextové. Je to podobné jako u regulárních jazyků, kde nám Nerodova věta dává

možnost, jak sporem ukázat, že jazyk není regulární. Stejně tak existuje tvrzení – pumping lemma, které formuluje vlastnost, kterou mají BKJ. Jazyky, které nejsou bezkontextové pak tuto vlastnost splňovat nebudou. Například pro jazyk $L=\{0^n1^n2^n;\ n \geq 0\}$ není možné sestrojit bezkontextovou gramatiku, tedy není regulární. Díky tomuto lemma to však umíme i dokázat.

Pumping lemma (věta o vkládání, uvwxy – teorém)

Věta 31: (lemma o vkládání, pumping lemma, uvwxy - teorém)
Nechť L je bezkontextový jazyk, pak existují přirozená čísla p,q taková, že každé slovo z ∈ L, které je delší než p (|z | > p), se dá psát ve tvaru z=uvwxy přičemž platí následující tři podmínky:
1. vx ≠ e (alespoň jedno ze slov v,x je neprázdné)
2. |vwx| ≤ q
3. $uv^iwx^iy \in L$ pro všechna i ≥ 0.

Pumping lemma nám dává nástroj na odhalování, že jazyk není bezkontextový. Pokud by byl bezkontextový, pak pro něj musí platit podmínka 3., která říká, že lze nalézt takové úseky slova, že když část v a x budeme pumpovat, budou vznikat slova z tohoto jazyka. Alespoň jedna část v nebo x musí být přitom neprázdná (podle 1). Zároveň toto platí pro slova od určité velikosti (konečná množina slov se může i v bezkontextovém jazyce vymykat tomuto pravidlu).

Z praktického hlediska nám říká, že jazyk obsahující závorkové struktury (např. známé z matematiky), nemůže být regulární, ale minimálně bezkontextový. Závorky (levé a pravé – jako symboly v a x) musí být nutně párovány ($\mathbf{v^i}$ **vs.** $\mathbf{x^i}$), což lze generovat právě pomocí bezkontextových jazyků.

Řešený příklad 38:
Vezměme jazyk $L=\{0^n1^n2^n;\ n \geq 0\}$ a dokažme, že tento jazyk nemůže být bezkontextový:

Aplikace, význam pumping lemmy

Pokud má být jazyk bezkontextový, platí pro něj pumping lemma. Tedy můžeme najít úseky v a x, které pumpováním vytvářejí slova z jazyka. Rozeberme možnosti, jak mohou tyto úseky vypadat:

v = 01 nebo v = 12 nebo x = 01 nebo x = 12, pak by vznikaly slova ...0101... nebo ...1212..., která jistě nejsou z jazyka, tedy v a u musí obsahovat slova ze stejných symbolů

v = 0, x = 0 (v = 1,x = 1) (v = 2,x = 2), pak ale budou vznikat slova z různým počtem 0,1,2

stejný případ je pokud v = 0, x = 1 nebo v = 0, x = 2 nebo v = 1, x = 2.

Nelze tedy naplnit podmínky lemmy a z toho plyne, že jazyk nemůže být bezkontextový.

Nejdůležitější probrané pojmy:

- bezkontextová gramatika, bezkontextový jazyk
- odvození
- regulární gramatika, lineární gramatika
- nevypouštějící gramatika
- redukovaná gramatika
- pumping lemma

Úkoly k textu:

1. Sestrojte DKA rozpoznávající jazyk L={w ∈ {a,b}*|w končí symbolem ‚a' nebo obsahuje ‚bab'} a převeďte ho na regulární gramatiku.
2. Lze každý ZNKA převést na regulární gramatiku? Zdůvodněte proč.

Korespondenční úkol:

Část 1:
Vyberte si dva redukované automaty z tohoto textu a převeďte je na regulární gramatiky.

Část 2:
Navrhněte dvě gramatiky s nejméně pěti neterminály. Převeďte je na redukovanou a nevypouštějící formu. Gramatiky musí mít taková pravidla, aby se při vytváření redukované gramatiky alespoň jeden neterminál zredukoval při každé fázi.
Určete jaký jazyk tuto gramatiky generují.

5 Zásobníkové automaty

Cíl:

Po prostudování této kapitoly pochopíte:

- co je zásobníkový automat
- jaký je jeho vztah k bezkontextovým jazykům
- co je pumping lemma (lemma o vkládání)

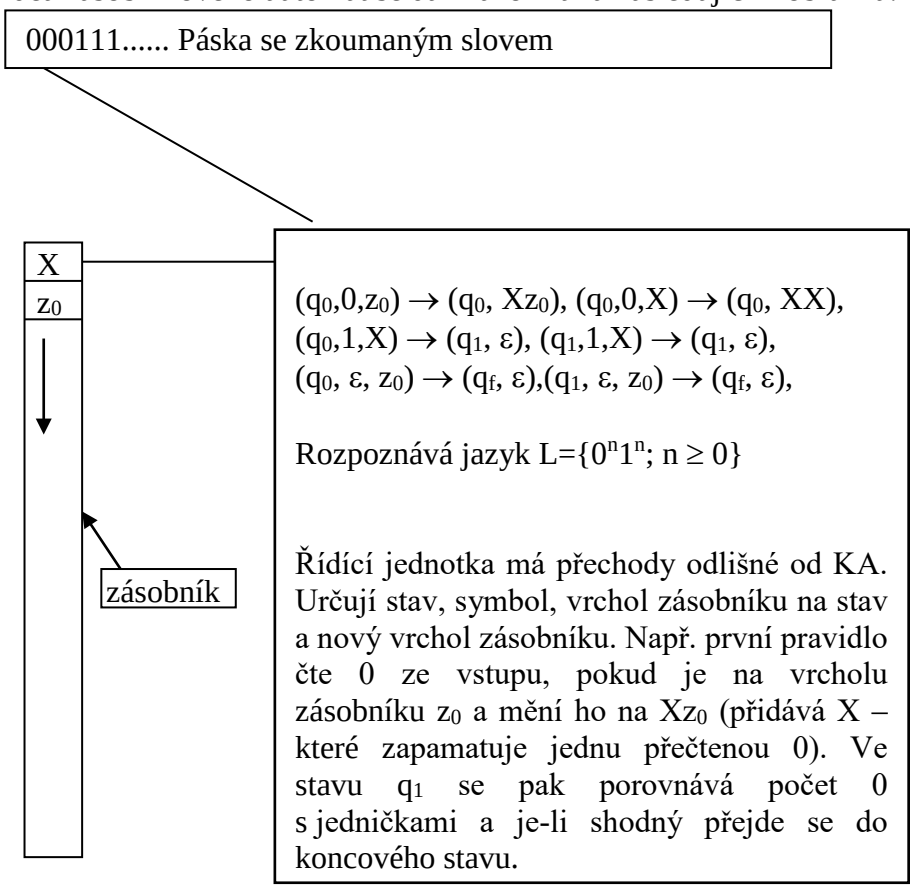
Naučíte se:

- vytvářet zásobníkové automaty pro zadané jazyky

Průvodce studiem

Zásobníkový automat je stroj, který stejně jako konečný automat má nějakou řídící jednotku, která je vždy v nějakém ze svých stavů, a který čte ze vstupní pásky slovo (nad nějakou abecedou) a po jeho přečtení rozhodne, zda slovo patří či nepatří do jazyka, který zásobníkový automat rozpoznává. Avšak narozdíl od konečných automatů, zásobníkový automat využívá navíc zásobníku, neboli jakési paměti typu LIFO. Tedy může ukládat a vybírat symboly na vrchol zásobníku, který si lze představit jako naskládané talíře – nelze je brát odkudkoliv – pouze z vrcholu.

Idea zásobníkového automat se dá znázornit na následujícím obrázku.



5.1 Zásobníkový automat a vztah k BKJ

Definice 41: Zásobníkovým automatem nazveme sedmici (systém určený sedmi parametry)
***M* = (*Q*, Σ, Γ, δ, q_0,Z_0,*F*), kde Q je konečná neprázdná množina stavů, Σ je konečná neprázdná množina vstupních symbolů (abeceda), Γ je konečná neprázdná množina zásobníkových symbolů, $q_0 \in Q$ je počáteční stav, $Z_0 \in \Gamma$ je počáteční zásobníkový symbol, $F \subseteq Q$ je množina koncových stavů a δ je zobrazení množiny $Q\times(\Sigma\cup\{e\})\times\Gamma$ do množiny konečných podmnožin množiny $Q \times\Gamma^*$ (přechodová funkce). ($\delta: Q\times(\Sigma\cup\{e\})\times\Gamma \rightarrow P(Q\times\Gamma^*)$)**

Zásobníkový automat

Z definice je patrné, že takto definovaný zásobníkový automat je nedeterministický.

Neformálně význam δ (tj. předpisu chování ZA *M*):
Je-li $\delta(q,a,X) = \{(q_1,\alpha_1),(q_2,\alpha_2), ...,(q_n,\alpha_n)\}$; $q \in Q$, $a \in (\Sigma\cup\{e\})$, $q_i \in Q, \alpha_i \in \Gamma^*$, $i \in \{1, 2, ..., n\}$,
$X \in \Gamma$, potom když *M* má čtecí hlavu na symbolu a, (konečná ŘJ) je ve stavu q a na vrcholu zásobníku je symbol X, může si *M* vybrat jedno i z {1, 2, ..., n } a posunout čtecí hlavu o jeden symbol vpravo, změnit stav řídící jednotky na q_i a symbol X v zásobníku nahradit řetězcem α_i. Speciálně je-li a=e, může *M* provést tzv. e-krok, při kterém nečte a hlava se tudíž neposunuje. Říkáme také, že *M* provedl instrukci (q,a,X) $[(\delta) \| (\rightarrow)]$ (q_i,α_i).
Důležitá je i skutečnost, že mohou existovat $q \in Q$, $a \in (\Sigma\cup\{e\})$, $X \in \Gamma$ tak, že $\delta(q,a,X)=\varnothing$ (v jistých situacích tedy nemůže automat pokračovat ve výpočtu). Při definici konkrétní přechodové funkce budeme definici obrazu pro takovéto vzory ((q,a,X)) vynechávat.

Definice 42: Mějme ZA *M* = (*Q*,Σ,Γ,δ,q_0,Z_0,*F*). Situací (konfigurací) zásobníkového automatu *M* nazveme trojici (q, w, α), kde $q \in Q$, $w \in \Sigma^*$ a $\alpha \in \Gamma^*$. q je stav ŘJ, w je slovo (ta část slova) na vstupní pásce, která zbývá přečíst, α je obsah zásobníku. (Nejlevější symbol v α představuje *vrchol zásobníku*). Jestliže $(q', \alpha) \in \delta(q,a,X)$, pak pro lib. $w \in \Sigma^*$, $\beta \in \Gamma^*$ vede situace $(q, aw, X\beta)$ bezprostředně k situaci $(q',w,\alpha\beta)$, symbolicky značíme:
$(q,aw,X\beta) \Rightarrow (q',w,\alpha\beta)$
Nechť E a E′ jsou situace ZA *M*, pak řekneme, že E vede k situaci E′, značíme $E \Rightarrow^* E'$, jestliže existují situace $E_1, E_2, ..., E_n$ tak, že $E=E_1 \Rightarrow E_2 \Rightarrow ... \Rightarrow E_n=E'$. Je-li potřeba, značíme o jaký ZA se jedná:
$\Rightarrow_M$ $\Rightarrow_M^*$

Konfigurace

Narozdíl od konečného automatu může ZA rozpoznávat slova nejen tím, že skončí v koncovém stavu, ale také tím, že vyprázdní celý svůj zásobník.

Například ilustrace na počátku kapitoly rozpoznává daný jazyk jak prázdným zásobníkem, tak i koncovým stavem.

Rozpoznávání jazyka zásobníkovým automatem budeme definovat dvěma způsoby:
1) přijímání koncovým stavem: slovo w je přijato ZA, jestliže existuje možnost, že po zpracování (přečtení) slova w se automat ocitne v koncovém stavu.
2) přijímání prázdným zásobníkem: slovo w je přijato ZA, jestliže existuje možnost, že po zpracování slova w se ZA ocitne v situaci s prázdným zásobníkem.

Rozpoznávání - koncovým stavem a prázdným zásobníkem

Definice 43: Mějme ZA $M = (Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$. Definujme
$L_{KS}(M)=\{w \in \Sigma^* | (q_0, w, Z_0) \Rightarrow_M^* (q,e,\alpha)$ pro nějaké $q \in F$ a $\alpha \in \Gamma^*\}$.
$L_{PZ}(M)=\{w \in \Sigma^* | (q_0, w, Z_0) \Rightarrow_M^* (q,e,e)$ pro libovolné $q \in Q\}$.

Deterministický ZA

Definice 44: ZA $M = (Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$ nazveme deterministický (DZA), jestliže platí následující dvě podmínky:
1. $\delta(q, a, X)$ je nejvýše jednoprvková množina pro lib. $q \in Q$, $a \in (\Sigma \cup \{e\})$, $X \in \Gamma$.
2. Jestliže $\delta(q,e,X) \neq \emptyset$ pro něj. $q \in Q$, $X \in \Gamma$, pak $\delta(q,a,X)=\emptyset$ pro lib. $a \in \Sigma$.

Definice 45: Jazyky rozpoznatelné DZA koncovým stavem nazveme deterministické (třídu těchto jazyků označíme Det). Jazyky rozpoznatelné DZA prázdným zásobníkem nazveme bezprefixové deterministické (třídu těchto jazyků označíme BDet).

Poznámka: Dá se ukázat, že Det je vlastní podtřída třídy bezkontextových jazyků.

(Např. jazyk $\{ww^R | w \in \{a,b\}^*\}$ není deterministický.)
(Srovnejte se situací u konečných automatů).

Řešený příklad 39:

Sestrojte zásobníkový automat, který rozpoznává jazyk $L=\{w(w)^R;\ w \in \{0,1\}^*\}$ (prázdným zásobníkem).

Hledaný automat $M=(\{p,q\},\{0,1\},\{A,B,C\},\delta,p,A,\varnothing)$ má přechodovou funkci δ definovánu takto:

$\delta(p,0,A)=\{(p,BA)\}$,
$\delta(p,1,A)=\{(p,CA)\}$,
$\delta(p,0,B)=\{(p,BB),(q,e)\}$,
$\delta(p,0,C)=\{(p,BC)\}$,
$\delta(p,1,B)=\{(p,CB)\}$,
$\delta(p,1,C)=\{(p,CC),(q,e)\}$,
$\delta(q,0,B)=\{(q,e)\}$,
$\delta(q,1,C)=\{(q,e)\}$,
$\delta(p,e,A)=\{(q,e)\}$,
$\delta(q,e,A)=\{(q,e)\}$.

Automat pracuje tak, že za každý symbol ze slova w přidá na zásobník zástupce, který pak porovná v zrcadlovém slově. Jelikož zásobník odebírá z vrcholu symboly rovněž zrcadlově, rozpozná právě slova z daného jazyka. Ale například jazyk $L=\{ww;\ w \in \{0,1\}^*\}$ (zdvojené slovo) už není možné rozpoznat ZA!

Řešený příklad 40:

Sestrojte zásobníkový automat, který rozpoznává jazyk $L=\{wc(w)^R;\ w \in \{0,1\}^*\}$ (prázdným zásobníkem).

Hledaný automat $M=(\{p,q\},\{0,1,c\},\{A,B,C\},\delta,p,A,\varnothing)$ má přechodovou funkci δ definovánu takto:

$\delta(p,0,A)=\{(p,BA)\}$,
$\delta(p,1,A)=\{(p,CA)\}$,
$\delta(p,0,B)=\{(p,BB)\}$,
$\delta(p,0,C)=\{(p,BC)\}$,
$\delta(p,1,B)=\{(p,CB)\}$,
$\delta(p,1,C)=\{(p,CC)\}$,
$\delta(p,c,A)=\{(q,e)\}$,
$\delta(p,c,B)=\{(q,B)\}$,
$\delta(p,c,C)=\{(q,C)\}$,
$\delta(q,0,B)=\{(q,e)\}$,
$\delta(q,1,C)=\{(q,e)\}$,
$\delta(q,e,A)=\{(q,e)\}$.

Poznámka: Tento automat je deterministický. Toto je příklad jazyka, ke kterému lze sestrojit DZA. Nicméně jsou i jazyky, ke kterým nelze DZA sestrojit (viz příklad předchozí).

Následující věta nám říká, že rozpoznávání KS a PZ jsou dvě ekvivalentní podmínky. Ke každému ZA KS lze sestrojit ekvivalentní ZA PZ a naopak. U PZ stačí doplnit instrukci, která při prázdném zásobníku automat dostane do koncového stavu. U KS je třeba doplnit více instrukcí, které v koncovém stavu (který změníme na nekoncový), vyprázdní postupně celý zásobník.

Věta 32: Mějme libovolný jazyk L. Pak $L=L_{KS}(M_1)$ pro nějaký ZA M_1, právě když $L=L_{PZ}(M_2)$ pro nějaký ZA M_2.

Ekvivalence rozpoznávání

Nyní formulujeme a dokážeme velmi důležité tvrzení, že jazyky rozpoznávané ZA jsou právě jazyky bezkontextové. Jde o stejný typ tvrzení jako, když jazyky generované regulárními gramatikami byly právě rozpoznatelné konečnými automaty.
Lze také jednoduše sestrojit ke každé gramatice ZA, který bude rozpoznávat generovaný jazyk a to pomocí simulace odvození v gramatice na zásobníku. Zpětně lze každý automat reprezentovat pomocí BKG – složitěji pomocí postupné simulace přechodů mezi stavy ZA

Věta 33: Ke každému bezkontextovému jazyku L existuje ZA M takový, že $L=L_{PZ}(M)$. Navíc M má jediný stav.

Důkaz:

Vztah jazyků rozpoznatelných ZA a BKJ

Mějme bezkontextovou gramatiku $G=(\Pi,\Sigma,S,P)$.
Sestrojíme ZA M tak, že $L(G)=L_{PZ}(M)$.
Položíme $M = (\{p\},\Sigma,\Pi\cup\Sigma,\delta,p,S,\varnothing)$.
Pro δ platí:
$\delta(p,e,X)=\{(p,\alpha)|(X\rightarrow \alpha) \in P\}; \forall X \in \Pi$
$\delta(p,a,a)=\{(p,e)\}; \forall a \in \Sigma$

Takto sestrojený ZA má dva typy pravidel – buď přepisuje neterminál na řetězec nebo srovnává terminální symboly. Pokud symboly nesedí, pak se automat zasekne. Obecně je automat nedeterministický – tedy musí si najít správnou cestu.

Věta 34: K libovolnému ZA M s jedním stavem, lze zkonstruovat bezkontextovou gramatiku G tak, že $L_{PZ}(M)=L(G)$.

Věta 35: K libovolnému ZA M lze zkonstruovat ZA M' s jedním stavem takový, že $L_{PZ}(M)=L_{PZ}(M')$.

Řešený příklad 41:

Mějme zásobníkový automat $M = (Q=\{p,q\},\Sigma = \{0,1\},\Gamma = \{A,B\},\delta,p,A,\varnothing)$
δ:
$\delta(p,0,A)=\{(p,BA)(q,A)\}$
$\delta(p,0,B)=\{(p,BB)\}$
$\delta(p,1,B)=\{(p,e)\}$
$\delta(q,0,A)=\{(q,A)(q,e)\}$
$\delta(q,e,A)=\{(p,BB)\}$
Zkonstruujte M' s jedním stavem tak, aby $L_{PZ}(M)=L_{PZ}(M')$.
Řešení:
$\delta'(p,1,\langle p,B,p\rangle) \ni (p,e)$
$\delta'(p,0,\langle q,A,q\rangle) \ni (p,e)$
$\delta'(p,0,\langle p,A,p\rangle) \ni (p,\langle q,A,p\rangle)$
$\delta'(p,0,\langle p,A,q\rangle) \ni (p,\langle q,A,q\rangle)$
$\delta'(p,0,\langle q,A,p\rangle) \ni (p,\langle q,A,p\rangle)$
$\delta'(p,0,\langle q,A,q\rangle) \ni (p,\langle q,A,q\rangle)$
$\delta'(p,0,\langle p,A,p\rangle) \ni (p,\langle p,B,p\rangle\langle p,A,p\rangle)$
$\delta'(p,0,\langle p,A,p\rangle) \ni (p,\langle p,B,q\rangle\langle q,A,p\rangle)$
$\delta'(p,0,\langle p,A,q\rangle) \ni (p,\langle p,B,p\rangle\langle p,A,q\rangle)$
$\delta'(p,0,\langle p,A,q\rangle) \ni (p,\langle p,B,q\rangle\langle q,A,q\rangle)$
$\delta'(p,0,\langle p,B,p\rangle) \ni (p,\langle p,B,p\rangle\langle p,B,p\rangle)$
$\delta'(p,0,\langle p,B,p\rangle) \ni (p,\langle p,B,q\rangle\langle q,B,p\rangle)$
$\delta'(p,0,\langle p,B,q\rangle) \ni (p,\langle p,B,p\rangle\langle p,B,q\rangle)$
$\delta'(p,0,\langle p,B,q\rangle) \ni (p,\langle p,B,q\rangle\langle q,B,q\rangle)$
$\delta'(p,e,\langle q,A,p\rangle) \ni (p,\langle p,B,p\rangle\langle p,B,p\rangle)$
$\delta'(p,e,\langle q,A,p\rangle) \ni (p,\langle p,B,q\rangle\langle q,B,p\rangle)$
$\delta'(p,e,\langle q,A,q\rangle) \ni (p,\langle p,B,p\rangle\langle p,B,q\rangle)$
$\delta'(p,e,\langle q,A,q\rangle) \ni (p,\langle p,B,q\rangle\langle q,B,q\rangle)$
$\delta'(p,e,R)=\{(p,\langle p,A,p\rangle),(p,\langle p,A,q\rangle)\}$

Přechodová funkce δ' zkonstruovaného ZA M' s jedním stavem
$\delta'(p,e,R)=\{(p,\langle p,A,p\rangle),(p,\langle p,A,q\rangle)\}$
$\delta'(p,0,\langle p,A,p\rangle)=\{(p,\langle q,A,p\rangle),(p,\langle p,B,p\rangle\langle p,A,p\rangle),(p,\langle p,B,q\rangle\langle q,A,p\rangle)\}$
$\delta'(p,0,\langle p,A,q\rangle)=\{(p,\langle q,A,q\rangle),(p,\langle p,B,p\rangle\langle p,A,q\rangle),(p,\langle p,B,q\rangle\langle q,A,q\rangle)\}$
$\delta'(p,0,\langle q,A,p\rangle)=\{(p,\langle q,A,p\rangle)\}$
$\delta'(p,0,\langle q,A,q\rangle)=\{(p,e),(p,\langle q,A,q\rangle)\}$
$\delta'(p,1,\langle p,B,p\rangle)=\{(p,e)\}$
$\delta'(p,0,\langle p,B,p\rangle)=\{(p,\langle p,B,p\rangle\langle p,B,p\rangle),(p,\langle p,B,q\rangle\langle q,B,p\rangle)\}$
$\delta'(p,0,\langle p,B,q\rangle)=\{(p,\langle p,B,p\rangle\langle p,B,q\rangle),(p,\langle p,B,q\rangle\langle q,B,q\rangle)\}$
$\delta'(p,e,\langle q,A,p\rangle)=\{(p,\langle p,B,p\rangle\langle p,B,p\rangle),(p,\langle p,B,q\rangle\langle q,B,p\rangle)\}$
$\delta'(p,e,\langle q,A,q\rangle)=\{(p,\langle p,B,p\rangle\langle p,B,q\rangle),(p,\langle p,B,q\rangle\langle q,B,q\rangle)\}$

Důsledky vztahů mezi ZA a BKG

Věta 36: (důsledek) Pro libovolný jazyk L jsou následující podmínky ekvivalentní:
1) L je bezkontextový
2) L je rozpoznatelný ZA koncovým stavem
3) L je rozpoznatelný ZA prázdným zásobníkem
4) L je rozpoznatelný ZA s jedním stavem prázdným zásobníkem

Stejně jako u regulárních jazyků lze sledovat, zda je třída BKJ uzavřena na množinové a jiné operace. V tomto případě to nebude platit pro všechny operace. Dále se naučíte Parikhovu větu, která je alternativní možností, jak dokazovat, že jazyky nejsou bezkontextové.

5.2 Uzávěrové vlastnosti třídy BKJ

Třída bezkontextových jazyků je uzavřena především vůči sjednocení, zřetězení a iteraci. Je velice jednoduché sestrojit k gramatikám jejich sjednocení a další operace.

Mějme $G_1=(\Pi_1,\Sigma,S_1,P_1)$ a $G=(\Pi_2,\Sigma,S_2,P_2)$ zkonstruovat gramatiky k jazykům $L_1=L(G_1)$ a $L_1=L(G_2)$ po aplikaci uzávěrových operací lze následovně:

$L_1 \cup L_2$: G: $S \rightarrow S_1$, $S \rightarrow S_2$, + P_1 + P_2 *(tedy vygeneruje slovo podle G_1 nebo G_2)*

$L_1 \cdot L_2$: G: $S \rightarrow S_1S_2$, + P_1 + P_2 *(tedy vygeneruje slovo podle G_1 a za ním podle G_2)*

L_1*: G: $S \rightarrow S_1S$, $S \rightarrow \varepsilon$, + P_1 *(tedy vygeneruje libovolněkrát slovo podle G_1)*

Věta 37: Třída bezkontextových jazyků je uzavřena vůči: sjednocení, zřetězení, iteraci, zrcadlovému obrazu, homomorfismu a substituci. (Ale je také uzavřena např. vůči průniku s regulárním jazykem a vůči kvocientu podle regulárního jazyka).

Uzávěrové vlastnosti

Průnik a doplněk však nemohou být sestrojeny, protože třída BKJ není uzavřena vůči těmto operacím. Existují totiž BKJ, jejichž průnik není BKJ. Příkladem budiž jazyk:

$L_1=\{0^n1^n2^m; n,m \geq 0\}$a $L_2=\{0^m1^n2^n; n,m \geq 0\}$, pak
$L_1 \cap L_2 = \{0^n1^n2^n; n \geq 0\}$, který ovšem jak už víme, není bezkontextový.

Věta 38: Třída bezkontextových jazyků není uzavřena vůči průniku a doplňku.

Nejdůležitější probrané pojmy:

- zásobníkový automat
- jazyky rozpoznatelné ZA
- vztah k bezkontextovým jazykům
- důsledky těchto vztahů
- Uzávěrové vlastnosti třídy BKJ

Takový jazyk existovat nemůže, neboť známe postup jak každý NKA převést na DKA.

Úkol k textu:

Vezměte si libovolné dva BKJ z tohoto textu a sestrojte gramatiku pro jejich sjednocení a zřetězení.

Kontrolní otázka:

Existují jazyky rozpoznatelné ZA, které nejsou rozpoznatelné deterministickým ZA?

Řešení:

Takový jazyk existuje (uvedený v předchozím textu).

Korespondenční úkol:

Část 1:
Vyberte si dva bezkontextové jazyky, ke kterým nebyl v tomto textu sestrojen ZA a sestrojte jej. Pokud to jde, sestrojte DZA.

6 Chomského hierarchie

Cíl:

Po prostudování této kapitoly pochopíte:

- hierarchizaci jazyků v jejich obecnosti
- které jazyky jsou složitější než regulární a bezkontextové
- jak pracují automaty pro složitější jazyky

Naučíte se:

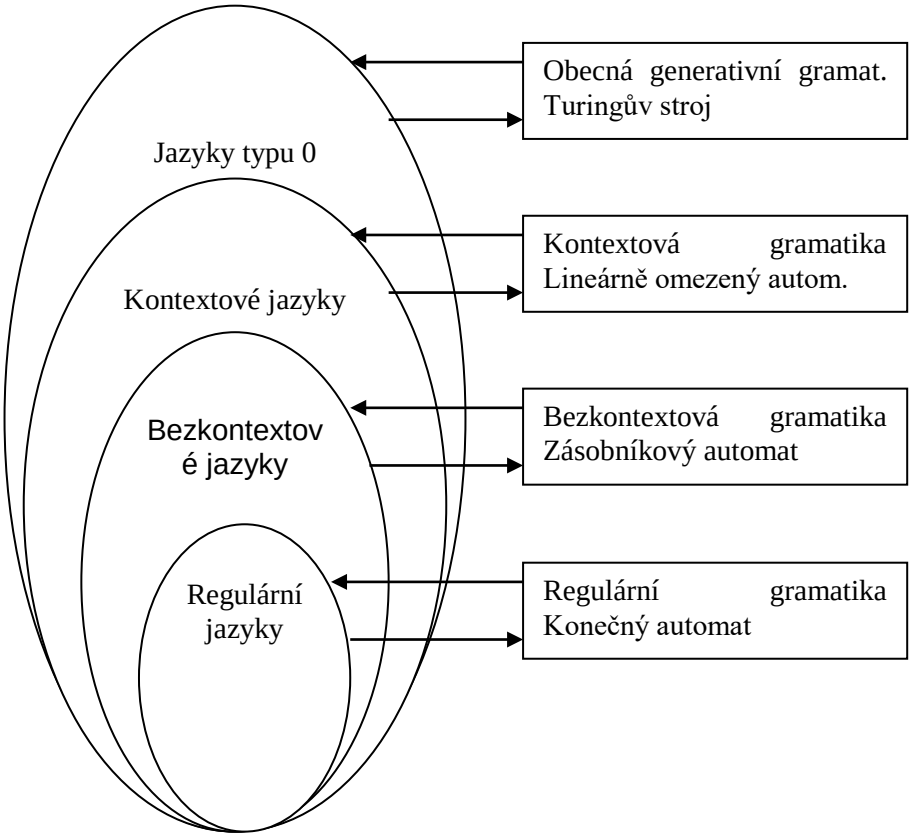
- klasifikovat jazyky z hlediska Chomského hierarchie

Průvodce studiem

Během vašeho studia teorie formálních jazyků jste se seznámili především se dvěmi třídami jazyků – regulárními a bezkontextovými. Existují ale samozřejmě i vyšší třídy jazyků (složitější). Vzpomeňte si na obecný pojem gramatiky. Právě tyto obecné gramatiky generují nejvyšší třídu jazyků (tzv. jazyky typu 0) podle Chomského hierachie. Právě podle již zmiňovaného Noama Chomského se tato klasifikace jazyků, podle toho jaké typy gramatik je generují, nazývá.

Na obrázku můžete toto rozdělení vidět. Chomského hierarchie obsahuje 4 třídy jazyků, které lze generovat generativními gramatikami. Samozřejmě, že s použitím generativních gramatik nelze vytvořit všechny jazyky – tyto jazyky jsou pak nad touto hierarchií. Pro teoretické výsledky teorie vyčíslitelnosti je důležitá třída jazyků typy 0 a kontextové jazyky (typu 1). Jazyky kontextové mají navíc význam pro umělou inteligenci, konkrétně analýzu přirozeného jazyka. Pro aplikované oblasti informatiky mají význam především jazyky bezkontextové (typu 2) a regulární (typu 3) a to při definování struktur programovacích a jiných jazyků používaných v praxi. Kromě gramatiky je důležitý zmíněný duální pojem automatu, který rozpoznává slova jazyka. Na obrázku jsou také ke každé třídě připojeny příslušné duální pojmy gramatiky - automatu.
V Chomského hierarchii je možné dále rozlišovat podtřídy podle toho zda jazyky lze analyzovat pomocí deterministického nebo nedeterministického automatu. Zvláště důležité to je pro třídu bezkontextových jazyků, které korespondují s používanými programovacími jazyky. Deterministické jazyky (rozpoznatelné deterministickými zásobníkovými automaty) jsou ve svých speciálních formách jako LL nebo LR jazyky efektivně analyzovatelné. Existují i alternativní hierarchie jazyků založené na odlišných přístupech ke generování jazyků, z nichž zřejmě nejznámější jsou Lindenmayerovy systémy využívané například v biologii pro simulaci chování živých organismů. Teorie jazyků je důležitou součástí informatiky a její poznatky se aplikují nejen v informatice samotné.

6.1 Obecná generativní gramatika a Chomského hierarchie



Chomského hierarchie

Definice 46: Generativní gramatika je čtveřice G=(Π,Σ,S,P), kde všechny parametry mají tentýž význam jako u bezkontextových gramatik s tím, že přepisovací pravidla jsou obecně tvaru $\alpha\rightarrow\beta$, kde $\alpha,\beta \in (\Pi\cup\Sigma)^*$, přičemž α obsahuje alespoň jeden neterminál.

Řekneme, že γ se přímo přepíše na δ a značíme $\gamma\Rightarrow\delta(\gamma,\delta \in (\Pi\cup\Sigma)^*)$, jestliže lze psát $\gamma = \gamma_1\alpha\gamma_2$, $\delta = \gamma_1\beta\gamma_2$, kde $(\alpha\rightarrow\beta) \in P$.

Relace $\Rightarrow^*$ je reflexivní a tranzitivní uzávěr relace $\Rightarrow$.

Jazyk generovaný gramatikou G je $L(G)=\{w \in \Sigma^* | S\Rightarrow^* w\}$.

Generativní gramatika

Nejstarší a nejznámější hierarchie gramatik podle tvarů přepisovacích pravidel je tzv. Chomského hierarchie.

Definice 47: Generativní gramatika G=(Π,Σ,S,P) je
0) typu 0, jestliže na pravidla neklademe žádná omezení
1) typu 1, neboli kontextová gramatika, jestliže všechna pravidla jsou ve tvaru $\alpha X\beta\rightarrow\alpha\gamma\beta$, kde $|\gamma| \geq 1$, $\alpha,\beta,\gamma \in (\Pi\cup\Sigma)^*$, $X \in \Pi$. Jedinou vyjimkou je pravidlo typu $S\rightarrow$ e, které se v gramatice objevit může, v tom případě se ale S nesmí objevit na pravé straně žádného pravidla.

2) typu 2, neboli bezkontextová gramatika (dřívější definice)
3) typu 3, neboli regulární gramatika (dřívější definice)

Věta 39: **Nechť L_i označuje třídu jazyků typu i. Pak $L_3 \subset L_2 \subset L_1 \subset L_0$.**

Důkaz:
$L_3 \subset L_2$, $L_1 \subset L_0$ triviálně platí.
$L_2 \subset L_1$ řeší se pomocí nevypouštějících bezkontextových gramatik.
Všechny inkluze jsou vlastní.
Např. $\{a^n b^n\} \in (L_2 - L_3)$.
Dále $\{a^n b^n c^n\} \in (L_1 - L_2)$. (viz následující příklad).
Inkluzi $L_1 \subset L_0$ nyní řešit nebudeme.

Třída kontextových jazyků, jak ji vidíte v definici obsahuje také jazyk, o kterém jsme dříve dokázali, že není bezkontextový. Kontextové gramatiky přepisují neterminály také v **kontextu** dalších slov. Nejlépe je to vidět na gramatice pro zmíněný jazyk.

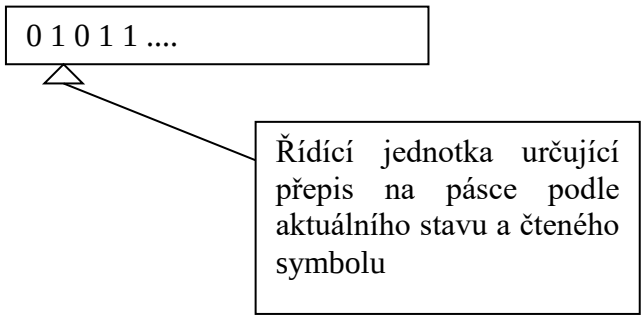
Řešený příklad 42:

Příklad: Gramatika pro $L=\{a^n b^n c^n\}$
G:
$S \rightarrow aSBC$
$S \rightarrow e$
$CB \rightarrow BC$
$aB \rightarrow ab$
$bB \rightarrow bb$
$bC \rightarrow bc$
$cC \rightarrow cc$
Není těžké ověřit, že $L(G)=L=(\{a^n b^n c^n\})$.
G se dá převést na ekvivalentní kontextovou gramatiku G′:
pravidlo $CB \rightarrow BC$ se nahradí trojicí pravidel
$CB \rightarrow CB'$
$CB' \rightarrow BB'$
$BB' \rightarrow BC$.

6.2 Turingův stroj

Na úrovni nejvyšší tedy u jazyků typu 0 je akceptorem takového jazyka Turingův stroj. Budete se jím detailně zabývat v teorii vyčíslitelnosti a složitosti. Nyní si ho ukažme pouze jako ideu. V roce 1936 Alan Turing, který je pro teoretickou informatiku klíčovou postavou, formuloval svou ideu formalizace pojmu algoritmus ve formě Turingova stroje (TS). Tato formalizace má svůj velmi jednoduchý princip mechanismu se vstupní potenciálně nekonečnou páskou s danou abecedou a čtecí hlavou, která může

zapisovat i číst na pásce a pohybovat se po jednom políčku. Schéma tohoto stroje lze vidět na obrázku. Lineárně omezený automat se liší jen v tom, že páska pro něj není nekonečná, ale je omezena na k – násobek velikosti vstupního slova. Právě to pak způsobí, že není schopen rozpoznávat jazyky typu 0.



Turingův stroj

Tento velice jednoduchý formalismus s velkou výpočetní silou umožnil formulovat pro informatiku klíčové pojmy jako jsou rozhodnutelnost a částečná rozhodnutelnost problémů (příp. lze tyto pojmy aplikovat na funkce, množiny či jazyky). Podařilo se dokázat vlastnosti některých problémů (nejznámějším nerozhodnutelným problémem je problém zastavení). Myšlenky důkazů těchto faktů jsou poměrně jednoduché, i když netriviální a lze je najít v literatuře [Ja97a] a [Ch84]. Dalšími důležitými výsledky jsou vztahy mezi jazyky typu 0 a rekurzivně spočetnými jazyky, které spadají také do TFJA. Pro zájemce lze doporučit distanční studijní oporu pro tento kurz [Pa02].

Nejdůležitější probrané pojmy:

- Chomského hierarchie
- Generativní gramatika
- Turingův stroj

Úkol k textu:

Sestrojte ke každé třídě jazyků Chomského hierarchie pět jazyků, které do ní patří (s výjimkou typu 0).

Chomského hierarchie

7 Základy syntaktické analýzy

Cíl:

Po prostudování této kapitoly pochopíte:

- Co je syntaktická analýza (SA)
- Kde se používá v reálných aplikacích
- Proč k zápisu syntaxe používáme bezkontextové jazyky

Naučíte se:

- Vytvořit jednoduchý (naivní) model syntaktického analyzátoru

Průvodce studiem

Při studiu základů teorie formálních jazyků a zejména dvou nejjednodušších tříd jazyků – regulárních a bezkontextových – jste se setkali s duálním konceptem gramatiky a automatu k danému jazyku. Gramatika umožňuje generovat daný jazyk (tedy jednotlivé prvky jazyka) a automat naopak rozpoznávat, zda testovaný prvek patří do jazyka. Z tohoto teoretického pohledu může někdy zůstat v pozadí fakt, že tento duální koncept znáte přímo ze své praxe informatiků.

Pravděpodobně nejbližším vám bude příklad z oblasti programování a tedy programovacích jazyků. Pokud se učíte používat daný programovací jazyk, učíte se zejména správně zapisovat programy dle jeho specifikace (odhlédneme-li nyní od toho, že chcete aby program dělal to co požadujete – to je vyšší stupeň). Učíte se tedy správně zapisovat **syntaxi** daného programovacího jazyka (tedy jeho strukturu z hlediska jazykových vyjadřovacích prostředků). Například u jazyka Pascal víte, že musí obsahovat nejprve deklarace typů, proměnných, dále deklarace funkcí a procedur a nakonec samotnou výkonnou část s popisem algoritmu –programu. Nebo na mnohem nižší úrovni víte, že aritmetický výraz vložený do přiřazovacího příkazu se může skládat s podvýrazů vzájemně spojených operátory sčítaní, odčítání apod. Přičemž nejjednodušším operandem může být kupříkladu celé číslo a důležité je, že tyto výrazy se mohou do sebe vzájemně vnořovat, čímž můžete vytvářet potenciálně nekonečně složité vnořené výrazy.

Syntaxe

Toto je vlastně malá část oné syntaxe jazyka, kterou musíte zvládnout. Když uděláme analogii s vašimi teoretickými poznatky z předchozího studia, učíte se vlastně **gramatiku** daného jazyka. Co však s oním duálním konceptem automatu? I on je vaší práci zcela přirozeně přítomen. Po správném napsaní programu samozřejmě vaše práce nekončí. Musíte si zdrojový kód programu pomocí zvoleného **překladače** (kompilátoru) přeložit do formy spustitelného nebo jiného cílového kódu. Součástí každého překladače musí být (mimo jiných mnoha dalších kroků) kontrola, zda je váš program správně zapsán podle specifikace jazyka. Tuto kontrolu musí provést jistá část překladače – algoritmu, která z teoretického hlediska funguje jako **automat**. Dá vám

Gramatika

Překladač

Syntaktický analyzátor

odpověď, zda je váš program správně napsán – tedy zda zdrojový kód patří do jazyka Pascal. Samozřejmě u pokročilých překladačů dostanete daleko více informací, včetně typu případné chyby a místa, kde chyba vznikla. V nejjednodušším případě pouhé kontroly typu ANO/NE (program je správně/není správně syntakticky zapsán) se jedná o takzvanou **syntaktickou analýzu** (dále budeme zkracovat **SA**). V anglicky psaných zdrojích se setkáte spíše s jednoslovným označením „**parsing**“. O daném postupu - algoritmu jak tuto SA provést, pak hovoříme jako **syntaktickém analyzátoru** (anglicky „**parser**“).
Z vašich znalostí rovněž vyplývá, že už znáte poměrně naivní metody, jak tyto analyzátory sestrojit. Jsou jimi například zásobníkové automaty. Problém však je (stejně jako i v jiných problémech informatiky), jak tyto analyzátory sestrojovat tak, aby byly dostatečně efektivní („rychlé“). Je jasné, že překladač Pascalu, který by váš program kompiloval celé hodiny by asi neměl pro vás žádný užitek. Podobně jako u jiných problémů, proto půjde především o to, jak najít efektivní analyzátory. Odpovědí bude jisté zjednodušení a okleštění příliš obecných bezkontextových gramatik a především tím se budeme v celém kurzu zabývat.

7.1 Bezkontextová gramatika a zápis syntaxe jazyka

Bezkontextová gramatika (BKG) je jedním z velmi vhodných způsobu zápisu syntaxe jazyků. Syntaxí zde rozumíme jejich jazykovou strukturu. Umožňuje totiž vyjádřit většinu technik, které například u programovacích jazyků používáme. Jde o alternativu několika možností, opakování stejného jazykového výrazu a hlavně vnořování celých rozvětvených struktur mezi sebou. Poslední zmiňovaná technika je právě onou technikou, kterou neumíme vyjádřit pomocí jazyků regulárních, ale teprve pomocí bezkontextových jazyků. Zkusme si představit velmi omezenou část nějakého programovacího jazyka – například strukturu aritmetického výrazu. Principiálně je většina jiných struktur velmi podobných (např. sekvence příkazů je analogická opakovanému sčítání podvýrazů!).

Řešený příklad 43:

Sestrojme BKG pro jazyk složený z aritmetických výrazů s operandem x, operacemi +,* a umožňující vnořovat další podvýrazy stejného typu pomocí symbolů závorek (,). Kupříkladu se může jednat o výraz:
x * (x + x + x)

Gramatiku sestrojíme hierarchicky – tedy aby byla rozlišena priorita operátorů a využijeme „rekurzivní“ vlastnosti přepisu neterminálu, abychom docílili možnosti generovat opakovaně sčítání a násobení.

G = ({S, A, B},{x, *, +, (,)}, S, P)

P:

$S \rightarrow_1 A + S$, $S \rightarrow_2 A$ *(opakovaný přepis na S nám umožní generovat libovolně mnoho sčítání struktury A)*
$A \rightarrow_3 B * A$, $A \rightarrow_4 B$ *(opakovaný přepis na A nám umožní generovat libovolně mnoho násobení struktury B)*
$B \rightarrow_5 (S)$, $B \rightarrow_6 x$ *(rekurzivním přepisem na S můžeme vnořit libovolně mnoho podvýrazů zcela stejné struktury jako výraz sám do závorek anebo ukončit generování operandem x)*
(Pozn.: index u symbolu $\rightarrow$ určuje pomocné číslo pravidla)

V této gramatice pak lze snadno generovat například výše uvedený výraz
x * (x + x + x):

$S \Rightarrow_2 A \Rightarrow_3 B * A \Rightarrow_6 x * A \Rightarrow_4 x * B \Rightarrow_5 x * (S) \Rightarrow_1 x * (A + S)$
$\Rightarrow_4 x * (B + S) \Rightarrow_6 x * (x + S) \Rightarrow_1 x * (x + A + S)$
$\Rightarrow_4 x * (x + B + S) \Rightarrow_6 x * (x + x + S) \Rightarrow_2 x * (x + x + A)$
$\Rightarrow_4 x * (x + x + B) \Rightarrow_6 x * (x + x + x)$

(Pozn.: index u symbolu $\Rightarrow$ určuje pomocné číslo pravidla)

Vidíte, že daná, poměrně jednoduchá gramatika, dokáže generovat relativně složitou strukturu, jakou je aritmetický výraz z hlediska hierarchie vnoření. Zároveň jsme díky indexům získali informaci o způsobu konstrukce výrazu – sekvence 23645146146246. Tím, že jsme navíc provedli kanonickou derivaci – levé odvození (vždy přepisujeme nejlevější neterminál) směřujeme k jednoznačnému postupu – algoritmu, jak generovat konkrétní výraz. Spolu s intuitivním pravidlem pro výběr varianty 1 nebo 2 resp. 3 nebo 4, které vybírá dle toho, zda je ještě přítomen v požadovaném výrazu další operátor stejného typu nebo ne, nám dává toto levé odvození deterministickou možnost krok po kroku derivovat právě požadovaný výraz. Samozřejmě je ještě potřeba správně vybrat pravidlo 5 nebo 6, ale to lze učinit jednoduše dle toho, zda se vyskytuje jako následující požadovaný znak x nebo (.

Zmiňovaný **determinismus** – tedy schopnost jednoznačně určit, které pravidlo máme použít – není samozřejmě automatický. Jsou gramatiky, kde jej nebudeme schopni splnit, což má poměrně značný dopad na efektivitu takového procesu. Kdybyste nevěděli, které pravidlo si vybrat, pokud máte více možností, museli byste zkoušet v podstatě všechny možnosti, které existují a čekat, zda dojdete k požadovanému výrazu. To obecně vede k takzvané „**kombinatorické explozi**“, což je vytváření obrovského množství možností geometrickou řadou. Takováto exploze samozřejmě značně omezuje použití daného postupu pro reálné aplikace. Je tedy jasné, že vhodný tvar výchozí gramatiky je pro reálné aplikace zásadní. A taktéž vám asi začíná být zřejmé, že i pro některé bezkontextové jazyky není vůbec možné deterministické a zároveň efektivní postupy najít.

7.2 Backusova-Naurova forma

Backusova-Naurova forma

Dalším přehledným a hlavně v praxi ještě více využívaným způsobem zápisu syntaxe jazyka je takzvaná **Backusova-Naurova forma (BNF)**. Jde o zápis podobný bezkontextové gramatice, ale přitom bližší spíše programátorům, resp. praxi.

BNF obsahuje podobně jako BKG neterminály, které se uvádějí do úhlových závorek a přepisují skrze symbol := na řetězce terminálních a neterminálních symbolů. Jde tedy o pravidla tvaru:

$<X> := \alpha_1 \mid \ldots \mid \alpha_n$

Pro přehlednější zápis je však ještě lepší modifikace BNF zvaná EBNF (Extended BNF) – rozšířená BNF, která zjednodušuje zápis opakovaně používaných, příp. podmíněně vyskytujících se výrazů. Umožňuje následující zápisy:

$\{\alpha\}$ – znamená, že výraz se vyskytuje v libovolném počtu (ekvivalent operace iterace)
$\{\alpha\}_n^m$ - znamená, že výraz se vyskytuje v počtu nejméně n a nejvýše m (ekvivalent operace mocniny od n do m)
$[\alpha]$ – znamená, že výraz se může a nemusí na daném místě vyskytnout - je to ekvivalentní zápisu $\{\alpha\}_0^1$

Řešený příklad 44:

Gramatika z předchozího příkladu by v BNF mohla být zapsána například takto:

<aritmetický výraz+> := <aritmetický výraz*>{+<aritmetický výraz*>}
<aritmetický výraz*> := <operand/podvýraz>{*<operand/podvýraz>}
<operand/podvýraz> := (<aritmetický výraz+>) | x

BNF umožňuje přehledný zápis a navíc i jednoduchý přechod k některým typům SA, které však budeme probírat spíše ve vyšším kurzu Překladače. V závěru textu se ale této metodě SA alespoň okrajově budeme věnovat. S pomocí BNF je zapsána například celá gramatika jazyka Pascal v učebnici [Ji88]. Příkladem může být deklarace podmíněného příkazu:

<podmíněný příkaz> := if <booleovský výraz> then <příkaz>|
if <booleovský výraz> then <příkaz> else <příkaz>

7.3 Syntaktická analýza v reálných aplikacích

Použití syntaktické analýzy v reálných aplikacích už trochu vyplývá z předchozích podkapitol. Jedním ze stěžejních použití je využití SA jako součásti překladače. Překladač je algoritmus (program), který k libovolnému kódu ve zdrojovém jazyce (zdrojový kód) vytvoří kód v cílovém jazyce (cílový kód). Tento proces se skládá z mnoha částí a zejména v počáteční fázi překladu hraje SA významnou roli. Počáteční fáze překladu integruje zejména tři druhy analýzy kódu:

1. Lexikální analýza
2. Syntaktická analýza
3. Sémantická analýza

První část tedy lexikální analýza shlukuje symboly do takzvaných lexikálních elementů. Příkladem takového elementu v programovacím jazyce může být například identifikátor. Typicky lexikální analýza nepřesahuje složitostí úroveň regulárních jazyků. Tu pak obstarává SA, která již pracuje s připravenými lexikálními elementy a vytváří jistou reprezentaci derivačního stromu podle konkrétní gramatiky. Sémantická analýza pak řeší problematiku kontroly určitých vazeb programu jako jsou vazby typů proměnných apod.

Samozřejmě, že překladač nemusí provádět pouze překlad zdrojového kódu v nějakém programovacím jazyce. Existují překladače zdrojových kódů textů v programech pro podporu sazby textu jako je např. TeX, kde popisujete text pomocí „příkazů“. Nebo lze provádět překlad mezi různými formáty např. RichTextFormat vs. TeX.

7.4 Syntaktická analýza „shora dolů“

Analýza „shora dolů“

Zásadní rozdělení přístupů v SA spočívá ve způsobu konstrukce derivačního stromu odvození pro daný jazyk (gramatiku). Prvním přístupem je analýza principem „**shora dolů**“ (anglicky top-down parsing). Tento princip vychází při analýze z myšlenky postupné dopředného odvozování na zásobníku od počátečního neterminálu S a srovnávání analyzovaného slova po terminálních symbolech, které se objeví na zásobníku až dojdeme do situace, kdy nám nezbude již nic ke srovnání a to jak v původním slově, tak v přepisovaných řetězcích od S na zásobníku. Tyto kroky, kdy přepisujeme od S nazýváme **expanze** (jelikož neterminály rozšiřujeme na řetězce podle pravidel).
Podívejme se na příklad takovéto analýzy.

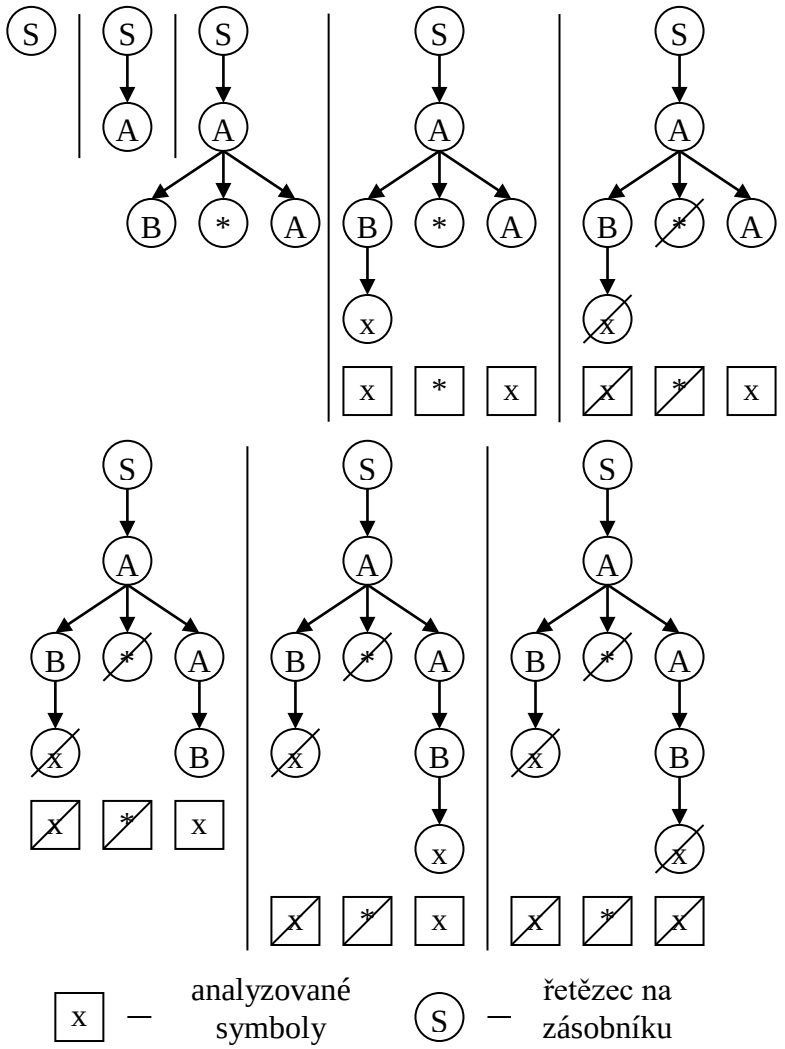
Řešený příklad 45:

Mějme gramatiku dle příkladu 1.
$G = (\{S, A, B\}, \{x, *, +, (,)\}, S, P)$
P: $S \rightarrow_1 A + S$, $S \rightarrow_2 A$
$A \rightarrow_3 B * A$, $A \rightarrow_4 B$

$B \rightarrow_5 (S), B \rightarrow_6 x$

Postupné expanze a srovnání od S můžeme lineárně a graficky znázornit, přičemž odvození zůstává stejné (každý krok odpovídá kroku ve znázornění, podtrženým písmem a symbolem ≈ zobrazujeme srovnávané terminální symboly). Slovo, které chceme rozpoznat zvolíme pro ilustraci jednoduché:
x * x

$S \Rightarrow_2 A \Rightarrow_3 B * A \Rightarrow_6 x * A \approx \underline{x} * A \approx \underline{x *} A \Rightarrow_4 x * B \Rightarrow_6 \underline{x *} x \approx \underline{x * x}$



7.5 Syntaktická analýza „zdola nahoru“

Druhým přístupem je analýza principem „**zdola nahoru**“ (anglicky bottom-up parsing). Tento princip vychází při analýze z myšlenky postupné zpětné odvozování tím, že postupně vkládáme symboly do zásobníku a pokud se nám vyskytne v zásobníku řetězec, který se vyskytuje u některého z pravidel na opačné (pravé straně), tak provedeme zpětně odvození na daný neterminál.

Princip je tedy zcela opačný – řetězce zjednodušujeme na neterminály. Tomuto opaku expanze se říká **redukce**. Slovo je přijato, pokud dojdeme k situaci, že slovo je celé přečteno a na zásobníku zbyl pouze neterminál S. Podívejme se na příklad takovéto analýzy.

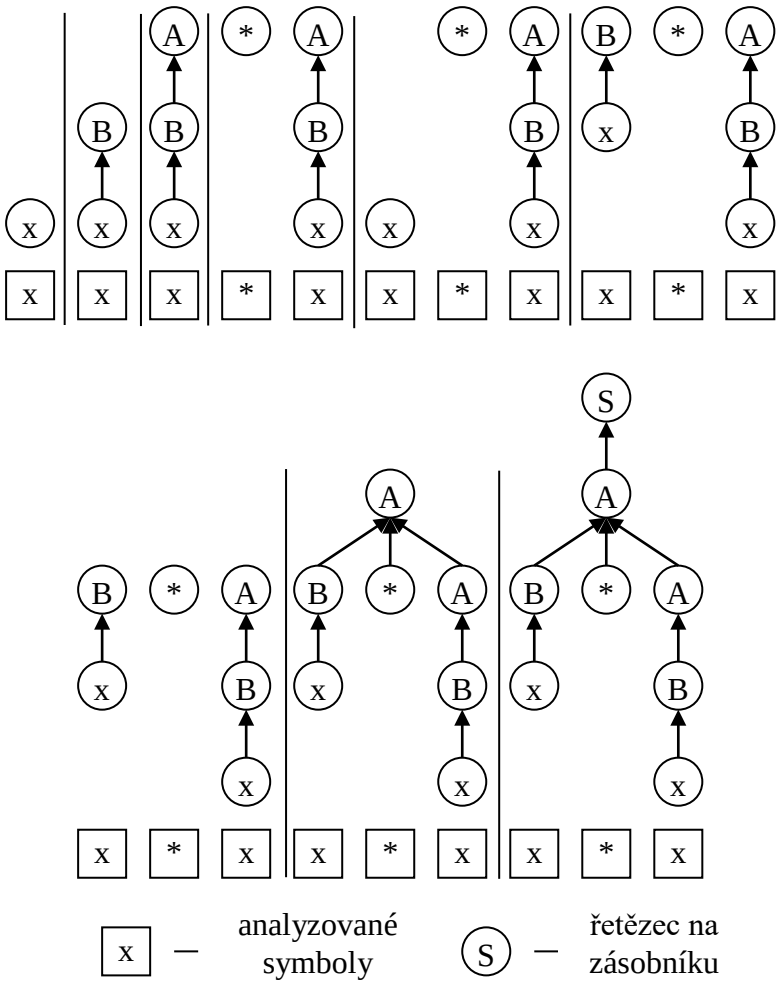
Analýza „zdola nahoru“

Řešený příklad 46:

Mějme gramatiku dle příkladu 1.

Postupné redukce a vkládání do zásobníku můžeme lineárně a graficky znázornit, přičemž odvození zůstává stejné (každý krok odpovídá kroku ve znázornění, podtrženým písmem a symbolem ≈ zobrazujeme vkládané terminální symboly). Slovo, které chceme rozpoznat zvolíme pro ilustraci jednoduché:
x * x (Pozor! V tomto případě musíme slovo číst v obráceném pořadí, pokud chceme opět dostat levé odvození.)

$\underline{x} \Leftarrow_6 B \Leftarrow_4 A \approx \underline{*} A \approx \underline{x *} A \Leftarrow_6 B \underline{*} A \Leftarrow_3 A \Leftarrow_2 S$



Oba principy analýzy bychom mohli zkoumat z pohledu jejich intuitivnosti i efektivity. Pravděpodobně intuitivnější se vám bude zdát SA „shora dolů“, neboť hierarchicky prochází jednotlivé struktury od nejsložitějších k nejjednodušším. Naopak analýza „zdola nahoru“ hledá možné „střípky skládanky“ a postupuje tím méně přehledně k nejvyššímu celku v hierarchii.

V textu si ukážeme oba přístupy na konkrétních typech BKG. Bude se jednat o takzvané **LL a LR gramatiky**. Zkratky LL a LR vyjadřují jednak způsob čtení analyzovaného slova („left-to-right“ – zleva doprava nebo „right-to-left“

naopak) a druhé písmeno pak vyjadřuje typ derivace, který dostaneme analýzou takových gramatik (left – levá derivace, right – pravá derivace).

I způsob využití v praxi souvisí s výše zmiňovanou „intuitivností“. Pokud budete chtít konstruovat analyzátory (překladače) spíše vlastními silami („ručně“), pak použijete přehlednější LL gramatiky (jsou ovšem slabší, pokud jde o vyjadřovací schopnosti – neumí vyjádřit některé jazyky, které LR gramatiky umí). V případě, že budete chtít použít již hotové automatizované nástroje (existuje jich mnoho a budeme se jimi zabývat ve vyšším kurzu překladače), použijete spíše LR gramatiky. Tyto nástroje vám umožní vygenerovat zcela automaticky analyzátor pro gramatiku. Ovšem tyto typy analyzátorů jsou méně přehledné a jejich algoritmická tvorba je mnohem náročnější na pochopení.

Nejdůležitější probrané pojmy:
- syntaxe jazyka a jeho gramatika
- překladač a automat
- syntaktická analýza a syntaktický analyzátor
- determinismus
- Backusova-Naurova forma
- SA „shora dolů“ a „zdola nahoru“

Úkoly k textu:

Sestrojte BKG a Backusovu-Naurovu formu pro jazyk výrokových formulí s jediným atomem x a operacemi konjunkce, disjunkce a implikace (s rozlišením priority) a dále s možností vnořit místo atomu podformuli uzavřenou do závorek.
Ke gramatice a vybrané formuli (co nejjednodušší) z úkolu 1. proveďte SA „shora dolů“ a „zdola nahoru“.

8 Syntaktická analýza shora dolů

Cíl:

Po prostudování této kapitoly pochopíte:

- Jaká omezení mají nejjednodušší typy LL gramatik
- Funkci rozkladové tabulky v SA

Naučíte se:

- Vytvářet rozkladovou tabulku pro SLL(1) gramatiku
- Provádět SA pomocí této rozkladové tabulky

Průvodce studiem

Studium této kapitoly by pro vás mělo být mnohem zajímavější než u teoretických kapitol. Ukážeme si některé postupy, které se uplatňují při tvorbě překladačů. Doporučuji sledovat pozorně příklady a teprve poté nezbytnou teorii. Věnujte této kapitole cca 6 hodin.

Při deterministické syntaktické analýze se v zásadě mohou využívat dva typy informací:

1. Informace o nepřečtené části analyzovaného (vstupního) řetězce
2. Informace o dosavadním průběhu SA

Samozřejmě, že čím méně takovýchto pomocných informací budeme potřebovat, tím jednodušší a přímočařejší SA bude. U LL gramatik se bude využívat především informace o k symbolech, které se v řetězci vyskytují na následujících k pozicích a pouze u obecnějších typů LL gramatik bude potřeba mít k dispozici i informace typu 2.

Z praktického hlediska je to velmi výhodné. Vraťme se opět k programovacím jazykům. Z hlediska konstrukce analyzátoru je velmi pohodlné (a tím i algoritmicky málo složité), když budeme zdrojový kód číst pouze dopředu bez návratů a navíc si nebudeme muset uchovávat nějaké další nadbytečné informace.

8.1 Model analýzy „shora dolů“ a jednoznačnost

Při studiu modelu analýzy typu „shora dolů“ jste viděli, že klíčovým problémem je rozhodnutí, jaké pravidlo použít, pokud máme u jednoho neterminálu více možností na co jej expandovat. Speciální typy gramatik pro tento typ analýzy proto mají omezení, které má především zabránit vzniku nejednoznačnosti při použití přepisovacího pravidla u stejného neterminálu. Podívejme se na následující velmi jednoduchou gramatiku:

Řešený příklad 47:

$G = (\{S, A\},\{a,b,c\}, S, P)$

$P: S \rightarrow_1 aASc,\ S \rightarrow_2 b$

$A \rightarrow_3 a,\ A \rightarrow_4 cSAb$

Lze vygenerovat například slovo: acbabbc
$S \Rightarrow_1 aASc \Rightarrow_4 acSAbSc \Rightarrow_2 acbAbSc \Rightarrow_3 acbabSc \Rightarrow_2 acbabbc$

U této gramatiky je už na první pohled zřejmé, že má pro každý neterminál dvě pravidla. Teoreticky zde tedy hrozí nejednoznačnost při expanzi. Při bližším zkoumání, jakým terminálním symbolem pravidla začínají, ale zjistíme, že každé z pravidel pro určitý neterminál začíná různým symbolem. Nabízí se tedy možnost rozhodnout se podle toho, jaký symbol v analyzovaném slově následuje. U této gramatiky je to poměrně jasné, avšak to jen díky dvěma faktům:

Kolize

1. Každé pravidlo začíná terminálem – tedy přímo „vidíme“ na jaký symbol máme přepis provést a tedy i vidíme, zda nedochází k přepisu na stejný terminál u dvou pravidel pro stejný neterminál (tzv. **kolize**).
2. Gramatika vůbec neobsahuje pravidlo s e (epsilonem) na pravé straně. Právě e-pravidla by celou situaci ještě mnohem více zkomplikovala, neboť způsobují, že příslušné neterminály mohou (ale nemusí) v odvození „mizet“. To pak v odhalování kolizí způsobuje další nepřehlednost.

Jak si později ukážeme je možné jednoznačnou SA provádět i bez splnění těchto podmínek, avšak bude to obtížnější.

8.2 Jednoduché LL(1) gramatiky a rozkladové tabulky

Gramatika z předchozího řešeného případu splňuje podmínky, které předpokládáme u nejjednoduššího typu LL gramatik. Jde o takzvanou **jednoduchou LL(1) gramatiku neboli SLL(1)** (Simple LL). Její základní omezení spočívá v tom, že vždy přepisuje neterminál na řetězec začínající terminálem a dvě pravidla pro jeden neterminál musí začínat různými terminály.

Jednoduchá LL(1) gramatika

Definice 48: Bezkontextová gramatika $G=(\Pi,\Sigma,S,P)$ je jednoduchá LL(1) gramatika nebo SLL(1) gramatika, pokud platí:

1. **$(X \rightarrow a\alpha) \in P$, kde a je terminál a α je řetězec složený z terminálů a neterminálů.**
2. **Když platí $(X \rightarrow a\alpha) \in P$ a $(X \rightarrow b\beta) \in P$, pak $a \neq b$.**

Číslo „1“ v názvu SLL(1) určuje počet symbolů, který musíme dopředu znát v analyzovaném slově, abychom byli schopni určit jaké pravidlo použít. Vidíte, že u každého pravidla jsme to schopni určit na základě pouhého jednoho znaku, neboť podmínka 2. vylučuje existenci dvou pravidel pro stejný neterminál začínající stejným znakem.

Pro každou takovou gramatiku (i obecnější typy LL gramatik) je možné sestrojit takzvanou rozkladovou tabulku, která určuje pro každý neterminál a příslušný následující symbol podle jakého pravidla máme provést expanzi.

Algoritmus pro vytvoření takovéto rozkladové tabulky pracuje podle jednoduchého principu – na příslušný řádek (odpovídající neterminálu) a příslušný sloupec (odpovídající vstupnímu symbolu na pravé straně pravidla) se vloží řetězec, který je u zkoumaného pravidla na pravé straně.

Pro gramatiku z předchozího příkladu by rozkladová tabulka vypadala následovně.

Řešený příklad 48:

Mějme gramatiku:
G = ({S, A},{a,b,c}, S, P)
P: S $\rightarrow_1$ aASc, S $\rightarrow_2$ b
A $\rightarrow_3$ a, A $\rightarrow_4$ cSAb

Rozkladová tabulka

M	a	b	c
S	aASc, 1	b, 2	
A	a, 3		cSAb, 4

Máme-li sestrojenou rozkladovou tabulku, můžeme pomocí algoritmu pro syntaktickou analýzu provést rozpoznání daného slova. Tento algoritmus postupně čte vstupní slovo, pracuje s pamětí typu zásobník a provádí 4 možné operace – expanze podle pravidel, porovnání stejných symbolů na zásobníku i ve vstupní řetězci, přijetí slova (pokud se vyprázdní zásobník a slovo je přečteno) a chyba, pokud se dostaneme do situace, pro kterou není v rozkladové tabulce definovaná akce.

Následující formalizace uvedeného postupu je obecným postupem pro LL(1) gramatiky (tedy i pro vyšší typy obecnějších gramatik, které budeme probírat později).

Algoritmus 1: Syntaktická analýza pro LL(1) gramatiky.

Vstup: rozkladová tabulka M pro SLL(1) gramatiku, q-gramatiku nebo LL(1) gramatiku G=(Π,Σ,S,P), vstupní řetězec w $\in \Sigma^*$.

Algoritmus syntaktické analýzy

Výstup: levý rozklad (derivace) vstupního řetězce v případě, že w $\in$ L(G), jinak chybová signalizace.

Postup:

- Algoritmus čte vstupní řetězec, používá zásobník a vytváří výstupní řetězec složený z indexů pravidel.
- Konfigurace je trojice (x, α, π), kde x $\in \Sigma^*$, $\alpha \in (\Pi \cup \Sigma)^*$ a $\pi \in N^*$, kde x je dosud nepřečtená část slova, α je obsah zásobníku a π je posloupnost čísel pravidel reprezentující levý rozklad podle G.

- Počáteční konfigurace je (w, S, e) a algoritmus provádí přechody mezi konfiguracemi podle následujících kroků 1. a 2., dokud nenastane situace 3. nebo 4.
  1. **Expanze:** (ax, Aα, π) ⇒ (ax, βα, πi), pokud A ∈ Π, M(A, a) = β, i. Symbol A se na vrcholu zásobníku nahradí řetězcem β a číslo i je připojeno k posloupnosti reprezentující levý rozklad.
  2. **Porovnání:** (ax, aα, π) ⇒ (x, α, π), pokud a ∈ Σ^*. Totožné symboly na vrcholu zásobníku a ve vstupním řetězci se smažou resp. přečtou ze vstupu.
  3. **Přijetí:** konfigurace (e, e, π) znamená, že řetězec je rozpoznán, analýza končí a π obsahuje posloupnost pravidel reprezentující levou derivaci řetězce podle G.
  4. **Chyba:** ve všech ostatních případech analýza končí s chybovou signalizací.

Pokusme se nyní provést SA slova vygenerovaného v gramatice z předchozího příkladu, ke které použijeme rozkladovou tabulku výše uvedenou.

Řešený příklad 49:

Mějme řetězec acbabbc, pak následující přechod konfigurací reprezentuje SA.

(acbabbc, S, e) ⇒ (acbabbc, aASc, 1) ⇒ (cbabbc, ASc, 1) ⇒
(cbabbc, cSAbSc, 14) ⇒ (babbc, SAbSc, 14) ⇒ (babbc, bAbSc, 142) ⇒
(abbc, AbSc, 142) ⇒ (abbc,abSc, 1423) ⇒ (bbc, bSc, 1423)
⇒ (bc, Sc, 1423) ⇒ (bc, bc, 14232) ⇒ (c, c, 14232) ⇒ (e, e, 14232)

8.3 Tvorba rozkladové tabulky

Vlastní algoritmus pro vytvoření rozkladové tabulky lze formalizovat následovně. Tento postup je ovšem použitelný pouze pro SLL(1) gramatiky.

Algoritmus 2: Vytvoření rozkladové tabulky pro SLL(1) gramatiku.

Vstup: SLL(1) gramatika G=(Π,Σ,S,P).

Vytvoření rozkladové tabulky

Výstup: rozkladová tabulka M pro G.

Postup:
- Rozkladová tabulka je definována na kartézském součinu Π × Σ.
  1. Pokud A→ aα je i-té pravidlo v P, pak M(A, a) = aα, i.
  2. M(X, a) = chyba v ostatních případech (tyto přechody není třeba vypisovat – jako jejich ekvivalent slouží prázdný přechod).

Aplikujme nyní tyto postupy na trochu praktičtější problém. S pomocí velmi omezených prostředků, které nám dává SLL(1) gramatika se nám zápis i poměrně jednoduchých úloh bude konstruovat poměrně těžko. Nemáme totiž k dispozici klíčový prostředek, kterým je e-pravidlo (umožňuje provádět iteraci stejných výrazů) a zároveň nesmí dojít k situaci, kdy nějaké pravidlo začíná stejným terminálem. Tato kombinace činí z této úlohy v kontrastu s již řešenými podobnými gramatikami bez tohoto omezení poměrně složitý a méně přehledný problém.

Řešený příklad 50:

Sestrojme gramatiku, která bude popisovat blok v programovacím jazyce C, který bude složen ze sekvence abstraktních příkazů p nebo vnořených bloků (uzavřených do složených závorek – {,}). Aby bylo možné vůbec sestrojit SLL(1) gramatiku, musíme provést jisté omezení ukončení sekvence příkazů a bloků. V SLL(1) gramatice nemůžeme použít e-pravidlo a tedy je nutno odlišit situaci, kdy nějaký příkaz následuje a kdy ne. Upravíme si tedy deklaraci tak, že všechny příkazy jsou odděleny středníkem (;) vyjma posledního.

G = ({S, P, R},{p, {, }, ;}, S, P)
P: $S \rightarrow_1$ {P, $P \rightarrow_2$ pR, $P \rightarrow_3$ {PR, $R \rightarrow_4$;P, $R \rightarrow_5$ }

Konstrukce této gramatiky se opírá o následující pravidla:
- Neterminál S vytváří uzavření do závorek, ovšem musíme k ukončení použít jiných prostředků než u neomezených BKG, je zde totiž problém, jak rozlišit, že už jde o koncový příkaz nebo ještě blok pokračuje dalším.
- Sekvence příkazů je generována neterminálem P, rozlišují se dvě možnosti – buď jde o příkaz p nebo o vnořený blok začínající závorkou, oba konstrukty je možno ukončit pomocí R
- R vyžaduje rozlišení, zda ještě následuje další příkaz oddělený středníkem nebo jde o konec bloku uvozený uzavírací závorkou
- Jen díky tomuto poměrně nepřehlednému přístupu jsme dokázali sestrojit SLL(1) gramatiku, což navozuje myšlenku, že tyto gramatiky pro praktické úlohy jsou příliš omezené

Můžeme nyní odvodit ukázkový blok jazyka C:

$S \Rightarrow_1$ {P $\Rightarrow_2$ {pR $\Rightarrow_4$ {p;P $\Rightarrow_3$ {p;{PR$\Rightarrow_2$ {p;{pRR $\Rightarrow_5$ {p;{p}R
$\Rightarrow_4$ {p;{p};P} $\Rightarrow_4$ {p;{p};pR $\Rightarrow_5$ {p;{p};p}

Vidíte, že díky nelogickému rozdělení na část počáteční a koncovou je odvození složitější a nepřehlednější než u obecné BKG. Nyní proveďme kontrolu zda jde opravdu o SLL(1) gramatiku a pokud ano, tak sestrojíme rozkladovou tabulku a provedeme analýzu bloku, který jsme právě vygenerovali.

Kontrola podmínek pro SLL(1) gramatiku.
1. Všechna pravidla začínají terminálem.

2. U S je pouze jedno pravidlo a tudíž konflikt nehrozí. Neterminál P se přepisuje buď na řetězec začínající p nebo {, což opět není konflikt a neterminál R se přepisuje na řetězec začínající ; nebo }, což opět jsou různé symboly.

Gramatika tedy je SLL(1).

Rozkladovou tabulku sestrojíme dle algoritmu.

M	p	;	{	}
S			{P, 1	
P	pR, 2		{PR, 3	
R		;P, 4		}, 5

Nyní můžeme provést analýzu slova {p;{p};p}:

({p;{p};p}, S, e) ⇒ ({p;{p};p}, {P, 1) ⇒ (p;{p};p}, P, 1) ⇒
(p;{p};p}, pR, 12) ⇒ (;{p};p}, R, 12) ⇒ (;{p};p}, ;P , 124) ⇒
({p};p}, P, 124) ⇒ ({p};p}, {PR, 1243) ⇒ (p};p}, PR, 1243) ⇒
(p};p}, pRR, 12432) ⇒ (};p}, RR, 12432) ⇒ (};p}, }R, 124325) ⇒
(;p}, R, 124325) ⇒ (;p}, ;P, 1243254) ⇒ (p}, P, 1243254) ⇒
(p}, pR, 12432542) ⇒ (}, R, 12432542) ⇒ (}, }, 124325425)
⇒ (e, e, 124325425)

Slovo bylo rozpoznáno a levá derivace je reprezentována čísly použitých pravidel: 124325425

Výhodou SLL(1) gramatiky je samozřejmě velmi jednoduchá konstrukce rozkladové tabulky oproti složitějším gramatikám, které budeme probírat v následujících kapitolách. S pomocí této tabulky už je analýza zcela deterministická a mohl by ji velmi jednoduše provádět například počítačový program.

8.4 Q-gramatika a funkce FOLLOW

V předcházející podkapitole jsme se seznámili s nejjednodušším typem LL(1) gramatiky, který neumožňuje použití **epsilon pravidel**. Jak už bylo řečeno, jde o velmi omezující kritérium, neboť to kupříkladu neumožňuje zapsat žádnou gramatiku, ve které se vyskytuje prázdné slovo. To by ještě nebyl zásadní problém (lze se omezit i na takové gramatiky a samotné prázdné slovo řešit jinak – nesystémově). Bez epsilon pravidla nemůžeme přehledně popsat syntaktické struktury, které jsou obvyklé v problémových úlohách (viz předchozí kapitola). V následujícím textu si tedy ukážeme o něco obecnější gramatiky, které e-pravidlo připouštějí.

Takzvané q-gramatiky jsou vlastně SLL(1) gramatiky s rozšířením umožňujícím použít epsilon pravidlo. Toto použití ovšem není zcela

automatické. Epsilon pravidlo může totiž způsobit ne zcela transparentní kolizi mezi tím, čím může pro určitý neterminál řetězec začínat a tím, co může následovat v generovaném slově v případě, že by se epsilon pravidlo aplikovalo (tudíž by se neterminál vymazal a následovat mohou **všechny řetězce**, které lze odvodit **bezprostředně za tímto neterminálem**).

To bude vyžadovat definici speciální funkce **FOLLOW**, která obsahuje všechny takové symboly, které následují. Zjištění, o které symboly jde, není triviální postup, avšak algoritmus lze zapsat několika pravidly.

Uvažujme následující gramatiku:

Řešený příklad 51:

G = ({S, A},{a, b, c}, S, P)
P: $S \rightarrow_1 aAS$, $S \rightarrow_2 b$, $A \rightarrow_3 cAS$, $A \rightarrow_4 e$

Tato gramatika není SLL(1), protože obsahuje epsilon pravidlo. Aby bylo možné provádět opět deterministickou SA podle stejného principu jako u SLL(1) gramatiky, musíme mít k dispozici rozkladovou tabulku. Vytvoření položek pro pravidla 1. – 3. se zdá velmi jednoduché a je totožné jako u SLL(1). Ale problematické je pravidlo 4. Kdy máme provést přepis podle něj a nezpůsobuje nám kolizi s jiným pravidlem pro neterminál A? Jak to poznáme?
Na tyto otázky existuje odpověď, pokud si uvědomíme, co bude znamenat aplikace tohoto pravidla. Pokud někde aplikujeme pravidlo 4., neterminál A v daném řetězci „zmizí“ a jako následující terminální symbol dostaneme množinu těch terminálů, které následují za A. Co do takovéto množiny patří. Uvažujme, kde se na pravé straně v pravidlech vyskytuje A. Jde o dva výskyty:

$S \rightarrow_1 aAS$ a $A \rightarrow_3 cAS$

V obou těchto případech vidíme, že pokud A „zmizí“ dostane se na jeho místo ten symbol, kterým „začíná“ neterminál S. U této gramatiky je pak zcela zřejmé, že S může díky pravidlům 1. a 2. začínat jedině symbolem a nebo b. Logicky se tedy tato pravidla aplikují, pokud se v generovaném/analyzovaném slově vyskytnou tyto symboly. Proto je zařadíme do příslušných sloupců pro neterminál A v rozkladové tabulce. Zároveň je jasné, že kdyby kterýkoliv z řetězců na pravé straně pravidla pro A začínal symbolem a nebo b, tak by nebyla tabulka jednoznačná, neboť se mohl provést buď přepis na tyto řetězce nebo na epsilon. Zde ovšem žádná nejednoznačnost nevzniká, neboť symboly a, b nekolidují s c. Tabulka ještě navíc musí obsahovat nový sloupec e (epsilon) a to z důvodu možnosti přepisu na zásobníku i v případě, že celé slovo už je přečteno a my můžeme ještě aplikovat epsilon pravidla (ty nevygenerují žádné symboly a tudíž nedojde k nesrovnalosti s obsahem zásobníku a analyzovaného řetězce).
Rozkladová tabulka tedy vypadá takto:

M	a	b	c	e
S	aAS, 1	b, 2		
A	e, 4	e, 4	cAS, 3	

Podle této tabulky můžeme analyzovat slovo aacbb:
(aacbb, S, e) ⇒ (aacbb, aAS, 1) ⇒ (acbb, AS, 1) ⇒ (acbb, S, 14)
⇒ (acbb, aAS, 141)
⇒ (cbb, AS, 141) ⇒ (cbb, cASS, 1413) ⇒ (bb, ASS, 1413)
⇒ (bb, SS, 14134) ⇒ (bb, bS, 141342) ⇒ (b, S, 141342)
⇒ (b, b, 1413422) ⇒ (e, e, 1413422)

Pozn. Každá aplikace pravidla 4 vyžadovala rozhodnutí, zda neterminál A vypustit. Je jasné, že kdyby například pro neterminál A a symbol b bylo definováno pravidlo, nedokázali bychom se rozhodnout zda použít ono konkrétní pravidlo nebo nejprve vypustit A a teprve následně b vygenerovat pomocí neterminálu nebo řetězce, který následuje! Podle symbolu, který se vykytuje jako následující v analyzovaném slově můžeme určit, které pravidlo by se použilo za předpokladu, že A se vypustí pomocí epsilon pravidla. V takovém případě, že tato možnost v kroku, který by následoval, existuje, můžeme díky epsilon pravidlu „uvolnit místo“ pro pozdější přepis na symbol následující ve slově. Proto se u q-gramatiky vyžaduje, aby množina FOLLOW pro neterminál, který přepisuje na epsilon, byla disjunktní s množinou symbolů, kterými začínají pravidla pro tento neterminál.

Definice 49: Máme neterminál X v G = (Π,Σ,P,S), pak platí:
FOLLOW(X) = {a| S⇒* αXβ, β ⇒*aγ, γ∈(Π ∪ Σ)* } ∪ {e| S⇒* αX}

Funkce FOLLOW

Hodnotou funkce FOLLOW pro daný neterminál X je množina všech symbolů, které mohu následovat za X, včetně e (epsilon), pokud za X už nemusí následovat žádný symbol (může být na konci řetězce v odvození).

Příkladem FOLLOW může být hodnota FOLLOW(A) = {a, b} pro gramatiku z předchozího příkladu.

Definice 50: BKG se nazývá q-gramatika, jestliže platí:

1. **Pravá strana pravidla je buď prázdná (epsilon) nebo začíná terminálem.**
2. **Každá dvě pravidla přepisující stejný neterminál X se liší terminálem, kterým začíná pravá strana (pokud X→aα, X→bβ jsou různá pravidla, pak a≠b).**
3. **Jestliže existuje pravidlo X→e, pak terminály, kterými začínají pravé strany ostatních pravidel X→aα, nesmí patřit do FOLLOW(X), a∉FOLLOW(X).**

Q-gramatika

8.5 Výpočet funkce FOLLOW

Výpočet funkce FOLLOW lze realizovat algoritmem, který pracuje podle následujících pravidel (jedná se o takzvaný tečkový algoritmus):

- tečka před symbolem v řetězci nám určuje místo, kde chceme určit terminální symbol, který následuje
- určíme si na začátku, které neterminály se mohou přepsat (libovolným počtem kroků) na e
- rozšiřujeme množinu pravidel s tečkou, tím, že přidáváme pravidla, která se vyskytují již v této množině a splňují daná kritéria

Výpočet FOLLOW

Algoritmus 3: Výpočet FOLLOW(A)
Vstup: BKG gramatika G=(Π,Σ,S,P) a neterminální symbol A
Výstup:FOLLOW(A)

Metoda:

1. Položíme N_e rovno množině všech prvků, které se dají přepsat (i rekurzivně) na prázdný řetězec (k tomu lze použít algoritmus z prvního dílu opory [Ha03], který se aplikoval při převodu BKG na nevypouštějící gramatiku).
2. Krok 2a: F = { A → A. }, do F umístíme fiktivní pravidlo, které reprezentuje situaci, kdy tečka vyjadřuje, že chceme zjistit všechny symboly, které se vyskytují za A.

   Krok 2b:

   Je-li v F pravidlo B → γ., kde γ je neprázdný řetězec, dáme do F všechna pravidla, ve kterých je na pravé straně B a tečku umístíme za B. V podstatě budeme dále určovat čím začíná řetězec za tečkou. Přidáváme tak vlastně všechny výskyty inkriminovaného neterminálu v celé gramatice, což je cílem.

   Krok 2c:

   Je-li v F pravidlo C → α.Bβ, přidáme do F všechna pravidla s B na levé straně, tečku umístíme na začátek. Tím, vlastně postupně rozvíjíme všechny neterminály na řetězce, na které se mohou přepsat.

   Krok 2d:

   Je-li v F pravidlo, kde je za tečkou neterminální symbol, který patří do N_e, do F vložíme toto pravidlo ještě jednou, ale tečku posuneme o jeden symbol doprava (to je případ, že tento neterminál se může přepsat na epsilon a tedy vypustit a je logické, že nás tedy zajímají i symboly bezprostředně za ním).

   Krok 2e:

   Kroky 2b, 2c, 2d opakujeme, dokud do F můžeme přidávat další položky.

   Krok 3:

Do FOLLOW(A) dáme všechny terminální symboly, před kterými je tečka. Je-li v F pravidlo S $\rightarrow \alpha.$, kde S je startovací symbol, přidáme do FOLLOW(A) i symbol e (epsilon) – tato situace reprezentuje možnost, že za zkoumaným neterminálem už není žádný symbol, tj. S se přepisuje na řetězec, kterým zkoumané místo končí.

Aplikujme nyní algoritmus na gramatice z předchozího příkladu.

Řešený příklad 52:

G = ({S, A},{a, b, c}, S, P)
P: S $\rightarrow_1$ aAS, S $\rightarrow_2$ b, A $\rightarrow_3$ cAS, A $\rightarrow_4$ e

Určíme FOLLOW(A).

Krok 1: Spočítáme N_e = {A} – zjevně S se nemůže přepsat na e, neboť vždy obsahuje přepis na alespoň jeden terminál.

Krok 2a: F = { A $\rightarrow$ A. }
Krok 2b: F = { A $\rightarrow$ A. , S $\rightarrow_1$ aA.S, A $\rightarrow_3$ cA.S }
Krok 2c: F = { A $\rightarrow$ A. , S $\rightarrow_1$ aA.S, A $\rightarrow_3$ cA.S, S $\rightarrow_1$ .aAS, S $\rightarrow_2$ .b }
Krok 2d: F = { A $\rightarrow$ A. , S $\rightarrow_1$ aA.S, A $\rightarrow_3$ cA.S, S $\rightarrow_1$ .aAS, S $\rightarrow_2$ .b } – nelze nic přidat, neboť tečka není nikde před A.
Krok 2e: F = { A $\rightarrow$ A. , S $\rightarrow_1$ aA.S, A $\rightarrow_3$ cA.S, S $\rightarrow_1$ .aAS, S $\rightarrow_2$ .b } – konec, protože už není možno v následujícím kroku přidat žádnou novou položku kroky 2b.,2c. ani 2d.
Krok 3: FOLLOW(A) = {a,b}

Výpočet funkce FOLLOW použijeme ve dvou případech:

1. Je nutný pro zjištění, zda gramatika je q-gramatika (viz podmínka 3 definice).
2. Umožní do rozkladové tabulky vložit buňky, odpovídající epsilon pravidlu pro daný neterminál.

8.6 Tvorba rozkladové tabulky

Vytvoření rozkladové tabulky pro q-gramatiku je složitější než pro SLL(1). Složitost spočívá v nutnosti správně zaplnit buňky odpovídající přepisu podle

pravidel s epsilon, což vyžaduje spočítat funkce FOLLOW pro všechny neterminály, které přepisují na e. Pak toto pravidlo vložíme do sloupců odpovídajících symbolům v množině FOLLOW, resp. epsilon, pokud FOLLOW e obsahuje.

Algoritmus 4: Vytvoření rozkladové tabulky pro q-gramatiku.

Vstup: q-gramatika G=(Π,Σ,S,P).

Výstup: rozkladová tabulka M pro G.

Postup:
- Rozkladová tabulka je definována na kartézském součinu $\Pi \times (\Sigma \cup e)$.
  1. Pokud $A \rightarrow a\alpha$ je i-té pravidlo v P, pak $M(A, a) = a\alpha$, i.
  2. Pokud $A \rightarrow e$ je i-té pravidlo v P, pak $M(A, b) = e$, i pro všechny $b \in$ FOLLOW(A).
  3. M(X, a) = chyba v ostatních případech (tyto přechody není třeba vypisovat – jako jejich ekvivalent slouží prázdný přechod).

Aplikujme nyní probrané postupy na příkladu analogickém jako v předchozí kapitole (s mírnou modifikací).

Řešený příklad 53:

Sestrojme gramatiku, která bude popisovat blok v programovacím jazyce C, který bude složen ze sekvence abstraktních příkazů p nebo vnořených bloků (uzavřených do složených závorek – {,}). Na rozdíl od konstruované q-gramatiky se již nemusíme omezovat v deklaraci jazyka. Můžeme dovolit použití epsilon a tudíž ukončení lze realizovat bez explicitního neterminálu a navíc není třeba vyžadovat, aby poslední příkaz bloku neobsahoval ;. Také je možné, aby příkaz byl prázdný – tedy pouze středník, resp. blok může být prázdný.

G = ({S, P, R},{p, {, }, ;}, S, P)
P: $S \rightarrow_1 \{P\}$, $P \rightarrow_2 p;P$, $P \rightarrow_3 \{P\}P$, $P \rightarrow_4 ;P$, $P \rightarrow_5 e$

Pozn. Neterminál P vyjadřuje všechny možnosti, jak může vypadat sekvence příkazů.

Můžeme nyní odvodit ukázkový blok jazyka C:

$S \Rightarrow_1 \{P\} \Rightarrow_2 \{p;P\} \Rightarrow_3 \{p;\{P\}P\} \Rightarrow_2 \{p;\{p;P\}P\} \Rightarrow_5 \{p;\{p;\}P\}$
$\Rightarrow_4 \{p;\{p\};P\} \Rightarrow_2 \{p;\{p\};p;P\} \Rightarrow_5 \{p;\{p\};p;\}$

Tato gramatika je schopna mnohem přehledněji generovat bloky a sekvence příkazů a navíc lépe odpovídá normě jazyka C.

Kontrola podmínek pro q-gramatiku. Potřebujeme vyčíslit FOLLOW(P):

Krok 1: Spočítáme N_e = {P} – zjevně S se nemůže přepsat na e, neboť vždy obsahuje přepis na alespoň jeden terminál.

Krok 2a: F = { P → P. }
Krok 2b: F = { P → P. , S $\rightarrow_1$ {P.}, P $\rightarrow_2$ p;P. , P $\rightarrow_3$ {P.}P, P $\rightarrow_3$ {P}P.,
P $\rightarrow_4$;P. }
Krok 2c: nic nového se nepřidá
Krok 2d: nic nového se nepřidá
Krok 2e: : F = { P → P. , S $\rightarrow_1$ {P.}, P $\rightarrow_2$ p;P. , P $\rightarrow_3$ {P.}P, P $\rightarrow_3$ {P}P.,
P $\rightarrow_4$;P. }
– konec, protože už není možno v následujícím kroku přidat žádnou novou položku kroky 2b.,2c. ani 2d.
Krok 3: FOLLOW(A) = { '}' } – obsahuje pouze symbol }

1. Všechna pravidla začínají terminálem nebo epsilonem.
2. U S je pouze jedno pravidlo a tudíž konflikt nehrozí. Neterminál P se přepisuje buď na řetězec začínající p, { nebo ; , což opět není konflikt.
3. Nesmí být konflikt mezi množinou FOLLOW(P) a všemi terminály, kterými začínají pravidla 2. – 4. Konflikt není, protože ve FOLLOW(P) je jen symbol } a ten u žádného z těchto pravidel na začátku není.

Gramatika tedy je q-gramatika.

Rozkladovou tabulku sestrojíme dle algoritmu.

M	p	;	{	}	e
S			{P}, 1		
P	p;P, 2	;P, 4	{P}P, 3	e, 5	

Nyní můžeme provést analýzu slova {p;{p;};p;}:

({p;{p;};p;}, S, e) ⇒ ({p;{p;};p;}, {P}, 1) ⇒ (p;{p;};p;}, P}, 1)
⇒ (p;{p;};p;}, p;P}, 12) ⇒ (;{p;};p;}, ;P}, 12) ⇒ ({p;};p;}, P}, 12)
⇒ ({p;};p;}, {P}P}, 123) ⇒ (p;};p;}, P}P}, 123)
⇒ (p;};p;}, p;P}P}, 1232) ⇒ (;};p;}, ;P}P}, 1232)
⇒ (};p;}, P}P}, 1232) ⇒ (};p;}, }P}, 12325) ⇒ (;p;}, P}, 12325)
⇒ (;p;}, P}, 12325) ⇒ (;p;}, ;P}, 123254) ⇒ (p;}, P}, 123254)
⇒ (p;}, p;P}, 1232542) ⇒ (;}, ;P}, 1232542) ⇒ (}, P}, 1232542)
⇒ (}, }, 12325425) ⇒ (e, e, 12325425)

Slovo bylo rozpoznáno a levá derivace je reprezentována čísly použitých pravidel: 112325425; každá z aplikací pravidla 5. vyjadřuje situaci, kdy došlo k umazání/ukončení P na konci bloku.

Nejdůležitější probrané pojmy:
- analýza „shora dolů“ a jednoznačnost
- LL gramatiky
- SLL(1) gramatika
- rozkladová tabulka
- algoritmus SA pro LL(1) gramatiky

Úkoly k textu:

Sestrojte SLL(1) gramatiku pro jazyk aritmetických výrazů s jediným operandem x s operací sčítaní a dále s možností vnořit místo operandu x podvýraz uzavřený do závorek o stejné struktuře.
K SLL(1) gramatice z úkolu 1. sestrojte rozkladovou tabulku a proveďte analýzu jednoduchého výrazu (s alespoň dvěmi spojkami a jedním vnořeným výrazem).

Nejdůležitější probrané pojmy:
- q-gramatika a její rozkladová tabulka
- Funkce FOLLOW

Úkoly k textu:

Sestrojte q-gramatiku pro jazyk aritmetických výrazů s jediným operandem x s operací sčítaní, násobení a dále s možností vnořit místo operandu x podvýraz uzavřený do závorek o stejné struktuře.

Ke q-gramatice z úkolu 1. sestrojte rozkladovou tabulku a proveďte analýzu jednoduchého výrazu (s alespoň dvěmi spojkami a jedním vnořeným výrazem).

9 Silné a slabé LL(k) gramatiky

Cíl:

Po prostudování této kapitoly pochopíte:

- co je LL(1) gramatika
- jakou funkci a jaká omezení přináší neterminál na začátku pravidla
- co je funkce FIRST a k čemu slouží

Naučíte se:

- vytvářet LL(1) gramatiky pro problémové úlohy
- vytvářet rozkladové tabulky pro LL(1) gramatiky
- vyčíslit funkci FIRST
- provádět SA pro LL(1) gramatiky

Průvodce studiem

Studium této kapitoly by pro vás mělo být mnohem zajímavější než u teoretických kapitol. Ukážeme si některé postupy, které se uplatňují při tvorbě překladačů. Doporučuji sledovat pozorně příklady a teprve poté nezbytnou teorii. Věnujte této kapitole cca 6 hodin.

Q-gramatiky mají již poměrně vysokou expresivitu, jak jsme viděli na problémových úlohách. Mají však ještě jednu nevýhodu, která se plně projevuje až u složitějších (rozsáhlejších gramatik). Touto nevýhodou je nutnost, aby každé pravidlo začínalo terminálem, což může vést také k jisté nepřehlednosti a nesystematičnosti gramatiky. Již dopředu (před studiem obecných vlastností LL gramatik) lze ale říci, že tento požadavek je už opravdu spíše otázkou „komfortnosti“ návrhu a zápisu gramatiku, protože libovolnou LL(1) gramatiku lze jednoduchým algoritmem převést na q-gramatiku (na rozdíl od vztahu q-gramatik a SLL(1) gramatik).

9.1 Funkce FIRST

LL(1) gramatika umožňuje, aby pravidlo začínalo neterminálem. Přesto stále musíme trvat na požadavku, aby i mezi takovými pravidly pro jeden neterminál nevznikala kolize. Jak ale takovou situaci odhalit a následně tvořit rozkladovou tabulku? Je k tomu potřeba opět jistá funkce, která se nazývá **FIRST**. Vyjadřuje množinu pro daný řetězec, která obsahuje všechny terminální symboly resp. epsilon, kterými může řetězec začínat. Způsob jejího formálního výpočtu je podobný jako u FOLLOW – postupně procházíme pravidla a přidáváme na základě jistých pravidel do množiny položky s tečkou a na konci tyto položky projdeme a zjistíme terminály, které jsou bezprostředně za tečkou. Uvažujme následující příklad.

Řešený příklad 54:

Silné a slabé LL(k) gramatiky

Mějme gramatiku pro tvorbu aritmetických výrazů s operandem x, operací sčítání a vnořenými podvýrazy se závorkami.

G = ({S, A, B},{x, +, (,)}, S, P)
P: S $\rightarrow_1$ BA, A $\rightarrow_2$ + BA, A $\rightarrow_3$ e, B $\rightarrow_4$ x, B $\rightarrow_5$ (S)

Pozn. V této gramatice B reprezentuje operandy a umožňuje generovat libovolně mnoho operandů spojených symbolem +.

Tato gramatika není zjevně ani SLL(1) ani q-gramatika, neboť pravidlo 1. nezačíná terminálem. Přesto pro ni lze poměrně analogicky sestrojit rozkladovou tabulku, pokud se nám podaří zjistit čím začíná řetězec BA. řetězec začíná na B a tedy pohledem na pravidla pro B zjišťujeme, že B může začínat jedinými dvěma terminály – x a (. Sestrojení rozkladové tabulky by se pak ubíralo stejnými pravidly, jako u q-gramatiky. Musíme vyčíslit FOLLOW(A), protože A se přepisuje na epsilon. FOLLOW(A)={), e}, protože za A následuje uzavírací závorka a navíc se může A vyskytnout zcela na konci slova (viz pravidlo 1.).

Rozkladová tabulka by pak vypadalo takto:

M	x	+	(	)	e
S	BA, 1		BA, 1		
A		+ BA, 2		e, 3	e, 3
B	x, 4		(S), 5		

Funkce FIRST

Definice 51: Máme řetězec $\alpha \in (\Pi \cup \Sigma)^*$ v G = (Π,Σ,P,S), pak platí: FIRST(α) = {a| $\alpha \Rightarrow^* a\beta$, $a \in \Sigma$, $\beta \in (\Pi \cup \Sigma)^*$ } ∪ {e| $\alpha \Rightarrow^* e$}

Funkce FIRST pro daný řetězec je daná množinou terminálů, kterými může řetězec po odvození začínat, resp. epsilon, pokud se může derivovat na prázdný řetězec.

Výpočet FIRST lze realizovat opět jednoduchým algoritmem, který vychází z „tečkové“ notace, kde tečka vyjadřuje místo, které chceme prozkoumat.

Výpočet funkce FIRST

Algoritmus 5: Výpočet funkce FIRST
Vstup: BKG gramatika G=(Π,Σ,S,P) a řetězec $\alpha = X_1X_2\ldots X_n$
Výstup:FIRST(α)

Metoda:

Krok 1a: Položíme množinu F={. $X_1X_2\ldots X_n$ }

Krok 1b:

Je-li v F pravidlo $A \rightarrow \beta.A\gamma$, kde $A \in \Pi$, přidáme do F všechna pravidla $A \rightarrow .\delta$, kde $\delta \in (\Pi \cup \Sigma)^*$

Tento krok reprezentuje výpis všech možností přepisu neterminálu, kterým může začínat řetězec – je před ním tečka, čímž získáme nové položky pro prozkoumání.

Krok 1c:

Je-li v F pravidlo $B \rightarrow \delta.$, kde $\delta \in (\Pi \cup \Sigma)^*$, přidáme do F pravidla z původní množiny F, ve kterých se vyskytoval symbol B, ale tečku umístíme až za něj. To se stane ve dvou případech:

- $\delta = e$ – a to tedy tečka můžeme pod neterminálem „podplavat“, neboť neterminál může být vypuštěn,
- $\delta \neq e$ – to stane pokud se všechny neterminály v řetězci mohly vypustit a tím pádem nastává stejný efekt jako v prvním případě.

Krok 1d:

Kroky 1b a 1c se opakují tak dlouho, dokud lze do F přidávat nové položky.

Krok 2:

FIRST(α) bude obsahovat všechny terminální symboly z F, které jsou bezprostředně za tečkou. Zároveň přidáme e, je-li tečka na konci pravidla.

Aplikujme tento algoritmus na předchozí gramatiku.

Řešený příklad 55:

G = ({S, A, B},{x, +, (,)}, S, P)
P: $S \rightarrow_1 BA$, $A \rightarrow_2 + BA$, $A \rightarrow_3 e$, $B \rightarrow_4 x$, $B \rightarrow_5 (S)$

Vypočtěme FIRST(BA).

Krok 1a: F={.BA}
Krok 1b: F={.BA, $B \rightarrow_4 .x$, $B \rightarrow_5 .(S)$ }
Krok 1c: nelze nic nového přidat.
Krok 1d: v dalším kroku by nic nového nebylo přidáno.
Krok 2: FIRST(BA)={x, (}

9.2 LL(1) gramatika

I LL(1) gramatika musí splňovat pravidlo, aby v její rozkladové tabulce nedošlo k žádné kolizi. Lze to formulovat slovně tak, že:

1. Pokud existují dvě pravidla pro jeden neterminál, řetězce na pravé straně musí začínat různými terminálními symboly (nemusí jít o explicitně zapsané terminální symboly!) Říkáme, že nesmí nastat tzv. **FIRST-FIRST kolize**.
2. Pokud se nějaký neterminál přepisuje na epsilon, pak všechna pravidla pro tento neterminál musí začínat jiným symbolem než který za neterminálem může následovat. (podmínka analogická jako u q-gramatiky). Nesmí nastat tzv. **FIRST-FOLLOW kolize**.

FIRST-FIRST kolize

FIRST-FOLLOW kolize

Přesně to lze elegantně formulovat pomocí FIRST a FOLLOW v definici.

LL(1) gramatika

Definice 52: BKG G=(Π,Σ,S,P) se nazývá LL(1) gramatika, jestliže platí pro každé A ∈ Π, kde v P jsou různá pravidla A→ α, A→ β :

1. **FIRST(α) ∩ FIRST(β) = ∅.**
2. **Pokud z řetězce α je možné generovat prázdný řetězec a z řetězce β není možné generovat prázdný řetězec, pak FOLLOW(A) ∩ FIRST(β) = ∅.**

Pozn. Obě podmínky lze ještě integrovat do sebe tak, že se vyjádří společnou ekvivalentní podmínkou:
FIRST(αFOLLOW(A)) ∩ FIRST(βFOLLOW(A)) = ∅.

9.3 Tvorba rozkladové tabulky

Pokud máme vytvořeny množiny FIRST všech řetězců, které se vyskytují na pravé straně pravidel, a množiny FOLLOW pro neterminály, které se přepisují na epsilon, je poměrně snadné vytvořit rozkladovou tabulku. Zjištění množin FIRST pro řetězce, které začínají terminálem je triviální a tudíž není nutné vždy aplikovat důsledně algoritmus na výpočet FIRST. U složitějších gramatik a řetězců, kde je na začátku neterminál je rozhdně bezpečnější provést výpočet dle algoritmu než odhadnou množinu pouhým pohledem na přepisovací pravidla. Ještě více to platí o FOLLOW, jejíž výpočet je o něco složitější.
Tvorba rozkladové tabulky se opírá o následující dvě pravidla:

1. Pro pravidla, která přepisují na neprázdné řetězce spočítáme FIRST těchto řetězců a do sloupců příslušných sloupců toto pravidlo vložíme. (výjimkou je sloupec epsilon, kam nepřidáváme nic – to je situace která se může vyskytnout na konci řetězce a tu řeší epsilon pravidla)
2. Pro pravidla, která přepisují na prázdné řetězce spočítáme FOLLOW neterminálu, který přepisují a vložíme toto pravidlo do příslušných sloupců symbolů resp. epsilonu pro nalezené prvky FOLLOW.

Algoritmus 6: Vytvoření rozkladové tabulky pro LL(1) gramatiku.

Vstup: LL(1) gramatika G=(Π,Σ,S,P).

Rozkladová tabulka pro LL(1) gramatiku

Výstup: rozkladová tabulka M pro G.

Postup:

- Rozkladová tabulka je definována na kartézském součinu Π × (Σ ∪ e).
  1. Pokud A→ α je i-té pravidlo v P, pak M(A, a) = α, i pro všechny a∈FIRST(α) – {e}.
  2. Pokud A→ α je i-té pravidlo v P a e∈FIRST(α), pak M(A, b) = α,i pro všechny b ∈ FOLLOW(A).
  3. M(X, a) = chyba v ostatních případech (tyto přechody není třeba vypisovat – jako jejich ekvivalent slouží prázdný přechod).

Aplikujme nyní všechny postupy na problémové úloze.

Řešený příklad 56:

Sestrojme gramatiku pro generování aritmetických výrazů s operacemi sčítaní, násobení a operandem x, umožňující navíc vnořovat podvýrazy stejného typu pomocí závorek.

G = ({S, A, B},{x, *, +, (,)}, S, P)
P: S $\rightarrow_1$ AP,
P $\rightarrow_2$ + AP, P $\rightarrow_3$ e
A $\rightarrow_4$ BR,
R $\rightarrow_5$ * BR, R $\rightarrow_6$ e,
B $\rightarrow_7$ (S), B $\rightarrow_8$ x

V této gramatice se na rozdíl od Řešený příklad 43: musí využít jiný způsob na vytváření opakovaného generování sčítaní a násobení. Nelze použít rekurzivní volání přímo S resp. A, protože by tím vznikla kolize u dvou pravidel, které by obě začínaly stejným řetězcem A resp. B. Proto se musí zavést nové neterminály P resp. R, které vlastně umožňují hrát roli jakéhosi vytýkaní neterimálu S resp. A, čímž odstraníme kolizi, tím, že se zbavíme dvou pravidel u S resp. A. U nově vzniklých neterminálů P resp. R kolize FIRST-FIRST nastat nemůže, neboť jedno z pravidel přepisuje na epsilon. Může však potenciálně nastat FIRST-FOLLOW kolize, díky ukončovacímu pravidlu s e. Jak ale ihned ukážeme, nedochází k ní ani v jednom případě.

Kontrola podmínek pro LL(1) gramatiku a vytvoření FIRST a FOLLOW množin, které budeme potřebovat i pro konstrukci rozkladové tabulky:

Nejprve určíme triviální množiny FIRST:
P2. FIRST(+AP) = {+}, P5. FIRST(*BR) = {*}, P7. FIRST((S)) = {(}, P8. FIRST(x) = {x}
Dále pomocí algoritmů určíme netriviální množiny FIRST:
P1. FIRST(AP)
Krok 1a: F={.AP}, krok 1b: F={.AP, A $\rightarrow_4$ .BR }, krok 1c.: nic, krok 1d: opakujeme 1b. F={.AP, A $\rightarrow_4$ .BR , B $\rightarrow_7$.(S), B $\rightarrow_8$ .x }, krok 1c.: nic, krok 1d.: už nic nového nemůžeme následným krokem přidat.
FIRST(AP)={(, x}

P4. FIRST(BR)
Krok 1a: F={.BR}, krok 1b: F={.BR, B $\rightarrow_7$.(S), B $\rightarrow_8$ .x }, krok 1c.: nic, krok 1d: už nic nového nemůžeme následným krokem přidat.
FIRST(BR)={(, x}

A nakonec zbývá nejsložitější výpočet FOLLOW.

P3. FOLLOW(P)
Krok 1: N_e={P, R}
Krok 2a: F={P $\rightarrow$ P.},

Krok 2b: F={P → P., S $\rightarrow_1$ AP., P $\rightarrow_2$ + AP.} – přidaly se pravidla, kde se vyskytuje P,
Krok 2c: F={P → P., S $\rightarrow_1$ AP., P $\rightarrow_2$ + AP.} – nic se nepřidá
Krok 2d: F={P → P., S $\rightarrow_1$ AP., P $\rightarrow_2$ + AP.} – nic se nepřidá
Krok 2e: opakujeme od 2b.
Krok 2b: F={P → P., S $\rightarrow_1$ AP., P $\rightarrow_2$ + AP., B $\rightarrow_7$ (S.)} – přidaly se pravidla, kde se vyskytuje S,
Krok 2c, 2d: F={P → P., S $\rightarrow_1$ AP., P $\rightarrow_2$ + AP., B $\rightarrow_7$ (S.)} – nic se nepřidá
Krok 2e: v následujícím průchodu už se nic nepřidá.

FOLLOW(P)={), e} – epsilon patří do množiny, neboť se vyskytla položka, kde se S přepisuje na řetězec s tečkou na konci

P6. FOLLOW(R)
Krok 1: N_e={P, R}
Krok 2a: F={R → R.},
Krok 2b: F={R → R., A $\rightarrow_4$ BR., R $\rightarrow_5$ * BR.,} – přidaly se pravidla, kde se vyskytuje R,
Krok 2c: F={R → R., A $\rightarrow_4$ BR., R $\rightarrow_5$ * BR.} – nic se nepřidá
Krok 2d: F={R → R., A $\rightarrow_4$ BR., R $\rightarrow_5$ * BR.} – nic se nepřidá
Krok 2e: opakujeme od 2b.
Krok 2b: F={R → R., A $\rightarrow_4$ BR., R $\rightarrow_5$ * BR., S $\rightarrow_1$ A.P, P $\rightarrow_2$ + A.P }– přidaly se pravidla, kde se vyskytuje A,
Krok 2c: F={R → R., A $\rightarrow_4$ BR., R $\rightarrow_5$ * BR., S $\rightarrow_1$ A.P, P $\rightarrow_2$ + A.P,
P $\rightarrow_2$.+ AP, P $\rightarrow_3$ .e }– přidaly se pravidla,která přepisují P,
Krok 2d: F={R → R., A $\rightarrow_4$ BR., R $\rightarrow_5$ * BR., S $\rightarrow_1$ A.P, P $\rightarrow_2$ + A.P,
P $\rightarrow_2$.+ AP, P $\rightarrow_3$ .e, S $\rightarrow_1$ AP., P $\rightarrow_2$ + AP. }– přidaly se pravidla, kde se tečka přesunula přes P, neboť je v N_e,
Krok 2e: opakujeme od 2b.
Krok 2b: F={R → R., A $\rightarrow_4$ BR., R $\rightarrow_5$ * BR., S $\rightarrow_1$ A.P, P $\rightarrow_2$ + A.P,
P $\rightarrow_2$.+ AP, P $\rightarrow_3$ .e, S $\rightarrow_1$ AP., P $\rightarrow_2$ + AP., B $\rightarrow_7$ (S.) } – vyskytla se nová položka s tečkou na konci pro S a P
Kroky 2c. a 2d.: nic nového se nepřidá
Krok 2e: v následujícím průchodu už se nic nepřidá.

FOLLOW(R)={+,), e} – epsilon patří do množiny, neboť se vyskytla položka, kde se S přepisuje na řetězec s tečkou na konci

Nyní zkontroluje podmínky pro LL(1).

1. Jediný neterminál, kterého hrozí FIRST-FIRST kolize je B. Ovšem pro pravidlo 7. a 8. FIRST((S)) = {(} a FIRST(x) = {x}, a tedy podmínka je splněna.
2. FIRST-FOLLOW kolize hrozí jednak u P a R.

FIRST(+AP) = {+} a FOLLOW(P)={), e} – kolize nenastala.
FIRST(*BR) = {*} a FOLLOW(R)={+,), e} – kolize nenastala.

Gramatika tedy je LL(1).

Nyní sestrojíme rozkladovou tabulku.

M	x	+	*	(	)	e
S	AP, 1			AP, 1		
P		+ AP, 2			e, 3	e, 3
A	BR, 4			BR, 4		
B	x, 8			(S), 7		
R		e, 6	*BR, 5		e, 6	e, 6

Analyzujeme ukázkový výraz (x + x) * x.
((x + x) * x , S, e) ⇒ ((x + x) * x , AP, 1) ⇒ ((x + x) * x , BRP, 14) ⇒
((x + x) * x , (S)RP, 147) ⇒ (x + x) * x , S)RP, 147) ⇒
(x + x) * x , AP)RP, 1471) ⇒ (x + x) * x , BRP)RP, 14714) ⇒
(x + x) * x , xRP)RP, 147148) ⇒ (+ x) * x , RP)RP, 147148) ⇒
(+ x) * x , P)RP, 1471486) ⇒ (+ x) * x , +AP)RP, 14714862) ⇒
(x) * x , AP)RP, 14714862) ⇒ (x) * x , BRP)RP, 147148624) ⇒
(x) * x , xRP)RP, 1471486248) ⇒ ( ) * x , RP)RP, 1471486248) ⇒
( ) * x , P)RP, 14714862486) ⇒ ( ) * x ,)RP, 147148624863) ⇒
(* x , RP, 147148624863) ⇒ (* x , *BRP, 147148624863) ⇒
(x , BRP, 147148624863) ⇒ (x , xRP, 1471486248638) ⇒
(e , RP, 1471486248638) ⇒ (e , P, 14714862486386) ⇒
(e , e, 147148624863863)

Slovo bylo rozpoznáno a výsledná sekvence 147148624863863 reprezentuje levé odvození.

9.4 LL(k) gramatiky

V předchozích kapitolách jsme se zabývali gramatikami, pro které lze poměrně jednoduše provádět SA pouze s informací, jaký následuje v analyzovaném slově první symbol. To je samozřejmě velice pohodlné a jak uvidíme dají se do tohoto tvaru převést gramatiky pro praktické problémy (programovací jazyky, výrazy atd.). Přesto existují i LL gramatiky, které nelze analyzovat s informací o jediném symbolu, ale s informací o k-symbolech následujících ve slově (říká se jim silné LL(k) gramatiky). A dokonce existují pro každé takové k i gramatiky, kde nám nestačí znát pouze informaci o následujících k-symbolech, ale musíme znát a provádět rozhodnutí na základě dosavadního průběhu samotné analýzy.

Abychom mohli rozšíření na k-symbolů následujících ve slově provést, musíme pochopitelně rozšířit funkce FIRST a FOLLOW.

Definice 53: Máme řetězec $\alpha \in (\Pi \cup \Sigma)^*$ a neterminál A v G = (Π,Σ,P,S), pak platí:

$FIRST_k(\alpha) = \{x \mid \alpha \Rightarrow^* x\beta, x \in \Sigma^*, |x| = k, \beta \in (\Pi \cup \Sigma)^* \} \cup \{x \mid \alpha \Rightarrow^* x, x \in \Sigma^*, |x| < k\}$
$FOLLOW_k(A) = \{x \mid S \Rightarrow^* wAx\beta, x \in \Sigma^*, |x| = k, w,\beta \in (\Pi \cup \Sigma)^* \} \cup \{x \mid S \Rightarrow^* wAx, x \in \Sigma^*, |x| < k, w,\beta \in (\Pi \cup \Sigma)^* \}$

Množina pro funkci **$FIRST_k(\alpha)$** tedy obsahuje jednak všechny řetězce o velikosti k, které se mohou vyskytovat na začátku řetězce α a jednak všechny řetězce o velikosti menší než k, pokud se takový řetězec vyskytuje na konci slova (už za ním nic nenásleduje). Může tedy jít o poměrně velkou a náročně se hledající množinu všech kombinací symbolů splňujích tyto dvě podmínky.
Množina pro funkci **$FOLLOW_k(A)$** obsahuje jednak všechny řetězce o velikosti k, které se mohou vyskytovat bezprostředně za A a jednak všechny řetězce o velikosti menší než k, pokud se takový řetězec vyskytuje na konci slova.
Algoritmy pro výpočet těchto funkcí nebudeme explicitně uvádět. Postačí slovní vyjádření pomocí modifikací algoritmů pro FIRST a FOLLOW. Jelikož chceme získat nejen symboly na první pozici, ale potenciálně na k pozicích, obohatíme algoritmy o další opakující se krok.

Modifikace algoritmů FIRST a FOLLOW pro výpočet $FIRST_k$ a $FOLLOW_k$.

V algoritmech se místo obyčejné tečky, bude používat teček s indexy. Všechny tečky bez indexu se považují za tečku s indexem 1. Provádíme-li operace přidání položky do F, index tečky se kopíruje.
Do algoritmů pro výpočet FIRST a FOLLOW přidáme následující kroky:

- Pokud se v položce v množině F vyskytuje tečka s indexem i před terminálním symbolem, umístíme další tečku s indexem i+1 za tento terminální symbol.
- Nevytváříme nikdy položky s tečkou s indexem vyšším než k.

Výsledná množina se modifikuje tak, že do ní budou patřit všechna terminální slova o délce maximálně k vyskytující se za tečkou s indexem 1 s vyloučením všech teček z těchto slov.
Dále definujeme funkce $FIRST_k$ a $FOLLOW_k$ pro celé množiny řetězců resp. symbolů.

Definice 54: Máme podmnožinu řetězců $R \subseteq (\Pi \cup \Sigma)^*$ v $G=(\Pi,\Sigma,P,S)$, pak platí:
$FIRST_k(R) = \{x \mid x \in FIRST_k(\alpha)$ pro nějaké $\alpha \in R\}$
$FOLLOW_k(R) = \{x \mid x \in FOLLOW_k(\alpha)$ pro nějaké $\alpha \in R\}$

9.5 Silné LL(k) gramatiky a jejich SA

Silná LL(k) gramatika

Silná LL(k) gramatika musí splňovat podobná kritéria jako LL(1) gramatika, avšak rozšířená na k-symbolů.

Silné a slabé LL(k) gramatiky

Definice 55: BKG G=(Π,Σ,S,P) se nazývá silná LL(k) gramatika, jestliže platí pro každé A ∈ Π, kde v P jsou různá pravidla A→ α, A→ β : $FIRST_k(\alpha FOLLOW_k(A)) \cap FIRST_k(\beta FOLLOW_k(A)) = \varnothing$.

Pro tyto gramatiky pak můžeme i použít velmi podobný algoritmus na vytvoření rozkladové tabulky (hlavní rozdíl spočívá ve struktuře, kde mohou být nejen jednotlivé symboly ve sloupcích, ale celé řetězce). Pro jednodušší definici zavedeme tedy množinu všech řetězců s omezenou délkou $\Sigma^{*k} = \{x | x \in \Sigma^*, |x| <= k\}$.

Rozkladová tabulka pro silnou LL(k) gramatiku

Algoritmus 7: Vytvoření rozkladové tabulky pro silnou LL(k) gramatiku.

Vstup: LL(k) gramatika G=(Π,Σ,S,P).

Výstup: rozkladová tabulka M pro G definovaná na $\Pi \times \Sigma^{*k}$.

Postup:

- Rozkladová tabulka je definována na kartézském součinu $\Pi \times (\Sigma \cup e)$.
  1. Pokud $A \rightarrow \alpha$ je i-té pravidlo v P, pak $M(A, x) = \alpha$, i pro všechny $x \in FIRST_k(\alpha)$, kde $|x| = k$.
  2. Pokud $A \rightarrow \alpha$ je i-té pravidlo v P a $y \in FIRST_k(\alpha)$ a $|y| < k$, pak $M(A, z) = \alpha$,i pro všechny $z \in FIRST_k(\alpha FOLLOW_k(A))$.
  3. M(X, y) = chyba v ostatních případech (tyto přechody není třeba vypisovat – jako jejich ekvivalent slouží prázdný přechod).

První podmínka tedy definuje všechny přechody pro řetězce o velikosti k. V druhé pak doplňujeme přechody pro řetězce o délce menší než k – samozřejmě včetně epsilon pravidel.

Abychom mohli provádět syntaktickou analýzu, je potřeba ještě nadefinovat modifikovaný algoritmus SA.

SA pro silnou LL(k) gramatiku

Algoritmus 8: Syntaktická analýza pro silné LL(k) gramatiky

Vstup: rozkladová tabulka M pro silnou LL(k) gramatiku G=(Π,Σ,S,P), vstupní řetězec $w \in \Sigma^*$.

Výstup: levý rozklad (derivace) vstupního řetězce v případě, že $w \in L(G)$, jinak chybová signalizace.

Postup:

- Algoritmus čte vstupní řetězec, používá zásobník a má k dispozici slovo $u = FIRST_k(x)$, kde x je dosud nepřečtená část řetězce.
- Počáteční situace je (w, S, e).
- Vykonávají se přechody podle 1. a 2. dokud nenastane situace 3. nebo 4.
  1. **Expanze:** $(x, A\alpha, \pi) \Rightarrow (x, \beta\alpha, \pi i)$, pokud $A \in \Pi$, $M(A, u) = \beta$, i. Symbol A se na vrcholu zásobníku nahradí řetězcem β a číslo i je připojeno k posloupnosti reprezentující levý rozklad.

2. **Porovnání:** (ax, aα, π) ⇒ (x, α, π), pokud a ∈ Σ. Totožné symboly na vrcholu zásobníku a ve vstupním řetězci se smažou resp. přečtou ze vstupu.
3. **Přijetí:** konfigurace (e, e, π) znamená, že řetězec je rozpoznán, analýza končí a π obsahuje posloupnost pravidel reprezentující levou derivaci řetězce podle G.
4. **Chyba:** ve všech ostatních případech analýza končí s chybovou signalizací.

Řešený příklad 57:

Mějme gramatiku:

G = ({S, A},{a,b}, S, P)
P: S $\rightarrow_1$ e, S $\rightarrow_2$ abA,
A $\rightarrow_3$ Saa, A $\rightarrow_4$ b

Ověřme nejprve, zda neplatí podmínky pro LL(1) gramatiku.

Musíme tedy určit FOLLOW(S)={a, e}. Ale zároveň platí, že FIRST(abA)={a} – pravidla obsahují FIRST-FOLLOW kolizi. Není to tedy určitě silná LL(1) gramatika. Zkusme tedy nyní ověřit zda je silná LL(2).

Určíme $FOLLOW_2(S)$.

$N_e = \{S\}$
$F=\{S\rightarrow S._1\}$
$F=\{S\rightarrow S._1, A\rightarrow S._1aa\}$
$F=\{S\rightarrow S._1, A\rightarrow S._1aa, A\rightarrow S._1a._2a\ \}$
$FOLLOW_2(S)=\{aa, e\}$

Dále potřebujeme určit $FOLLOW_2(A)$.

$N_e = \{S\}$
$F=\{A\rightarrow A._1\}$
$F=\{A\rightarrow A._1, S\rightarrow abA._1\}$
$F=\{A\rightarrow A._1, S\rightarrow abA._1, A\rightarrow S._1aa\ \}$
$F=\{A\rightarrow A._1, S\rightarrow abA._1, A\rightarrow S._1aa, A\rightarrow S._1a._2a\ \}$

$FOLLOW_2(A)=\{aa, e\}$

Nyní prověříme podmínku pro první dvě pravidla:

$FIRST_2(eFOLLOW_2(S))\cap FIRST_2(abAFOLLOW_2(S))=$
$FOLLOW_2(S)\cap FIRST_2(ab) = \{aa, e\}\cap\{ab\}=\varnothing.$

$FIRST_2(SaaFOLLOW_2(A))\cap FIRST_2(bFOLLOW_2(A))=$
$FIRST_2\ (Saa)\cap FIRST_2(bFOLLOW_2(A)) = \{ab, aa\}\cap\{b, ba\}=\varnothing.$

Je to tedy silná LL(2) gramatika.

Můžeme sestrojit rozkladovou tabulku.

M	aa	ab	a	ba	bb	b	e
S	e, 1	abA, 2					e, 1
A	Saa, 3	Saa, 3		b, 4		b, 4	

Nyní zanalyzujme slovo ababbaa s vyznačením řetězce u tučným písmem (k nebo méně symbolů z dosud nepřečteného slova pro LL(2)).

(**ab**abbaa, S, e) ⇒ (**ab**abbaa, abA, 2) ⇒ (**ba**bbaa, bA, 2) ⇒ (**ab**baa, A, 2)
⇒ (**ab**baa, Saa, 23) ⇒ (**ab**baa, abAaa, 232) ⇒ (**bb**aa, bAaa, 232)
⇒ (**ba**a, Aaa, 232) ⇒ (**ba**a, baa, 2324) ⇒ (**aa**, aa, 2324) ⇒ (**a**, a, 2324)
⇒ (e, e, 2324)

Slovo bylo rozpoznáno a levá derivace je reprezentována řetězcem 2324.

9.6 Slabé LL(k) gramatiky

Kromě toho, že existují poměrně jednoduše analyzovatelné silné LL(k) gramatiky, jsou zde také **gramatiky slabé LL(k).** Jejich syntaktická analýza už není možná pouze s informací o k následujících symbolech v řetězci, ale vyžadují také informaci o dosavadním průběhu analýzy. To celou SA velmi komplikuje a vyžaduje to nejen existenci rozkladové tabulky, ale také informace a průběhu SA. Tu reprezentuje takzvaný položkový automat.

Uvažujme jednoduchý příklad, který osvětlí problém SA slabých LL(k) gramatik.

Řešený příklad 58:

Mějme gramatiku:

G = ({S, A},{a,b}, S, P)
P: S $\rightarrow_1$ aAaa, S $\rightarrow_2$ bAba,
A $\rightarrow_3$ b, A $\rightarrow_4$ e

Pokud máme v této gramatice provést expanzi symbolu A a řetězec, který následuje je ba, neumíme se rozhodnout, zda použít pravidlo 3 nebo 4, protože výsledek je na k-symbolů stejný. Jediná možnost, jak toho rozhodnutí provést je vědět, zda jsme v předchozí analýze provedli vygenerování a podle pravidla 1 nebo b podle pravidla 2. To však už vyžaduje „pamatovat si“, co se stalo v předchozím průběhu analýzy. Pokud byl přečetný symbol a, tak se použije pravidlo 3 a pokud b, tak se použije pravidlo 4.
Tato gramatika tedy nemůže být silná LL(2), protože platí:
$FIRST_2(bFOLLOW_2(A)) \cap FIRST_2(FOLLOW_2(A))=\{ba\}$.

Přesto jde o takzvanou slabou LL(2) gramatiku podle následující definice.

(Slabá) LL(k) gramatika

Definice 56: BKG G=(Π,Σ,S,P) se nazývá (slabá) LL(k) gramatika, pro k>0, jestliže platí pro dvě levé derivace:

$$S \Rightarrow^* wA\alpha \Rightarrow^* w\beta\alpha \Rightarrow^* wx$$
$$S \Rightarrow^* wA\alpha \Rightarrow^* w\gamma\alpha \Rightarrow^* wy$$

takové, že $FIRST_k(x) = FIRST_k(y)$, platí $\beta = \gamma$.

Jinak řečeno, gramatika G je obecná LL(k), pokud pro vygenerování řetězce začínajícího stejnými k symboly, můžeme v gramatice použít pouze jedno pravidlo (tedy podmínka jednoznačnosti expanze). Ovšem tato podmínka je **slabší** než podmínka pro silnou LL(k), protože nemusí jednoznačnost splňovat samotné pravidlo, ale může to záviset na celém odvození slova. Historie syntaktické analýzy je nejjednodušeji reprezentovaná obsahem zásobníku – tento řetězec se nazývá **perspektivní přípona**.

Definice 57: Mějme levou derivaci $S \Rightarrow^* wA\alpha \Rightarrow^* w\beta\alpha$ v G=(Π,Σ,S,P). řetězec γ nazýváme perspektivní příponou v G, pokud γ=S nebo γ je příponou řetězce $\beta\alpha$.

Úplná perspektivní přípona je taková, která začíná neterminálem. V okamžiku, kdy je v zásobníku úplná přípona, provede SA expanzi na základě této přípony.

Řešený příklad 59:

Mějme gramatiku z předchozího příkladu. Pak provedeme porovnání v SA řetězce bba a abaa, jejichž průběh se liší právě v perspektivní příponě na zásobníku.

Vstupní řetězec	Obsah zásobníku	Provedená operace
bba	S	Expanze na bAba
bba	bAba	Porovnání b
ba	Aba	Expanze na e
ba	ba	Porovnání b
a	a	Porovnání a
e	e	Přijetí
Vstupní řetězec	Obsah zásobníku	Provedená operace
abaa	S	Expanze na aAaa
abaa	aAaa	Porovnání a
baa	Aaa	Expanze na b
baa	baa	Porovnání b
aa	aa	Porovnání a
a	a	Porovnání a
e	e	Přijetí

Z uvedených tabulek můžeme vyčíst, že expanze na třetím řádku se provede podle toho, zda na zásobníku je řetězec Aba nebo Aaa. V prvním případě se expanduje na e, protože chceme vygenerovat ba a v druhém na b, protože

generujeme baa. Tabulka i algoritmus SA by se tedy museli rozhodovat na základě bohatší informace než je jen neterminál a řetězec následujících symbolů. V řádcích by byly nikoliv pouze neterminály, ale celé perspektivní přípony.

M	aa	ab	ba	bb
S	aAaa, 1	aAaa, 1		bAba, 2
Aaa	e, 4		b, 3	
Aba			e, 4	b, 3

Tento postup pro náš daný jednoduchý případ lze aplikovat. Problém je, že není obecný, neboť perspektivních přípon může být nekonečně mnoho. Nekonečná tabulka samozřejmě nebude umožňovat efektivní syntaktickou analýzu. Druhým problémem je, jak zjistit pro perspektivní přípony jim příslušné pravidla.

V tomto textu se již dále zabývat SA pro slabé LL(k) gramatiky nebudeme. Z praktického hlediska to není příliš zajímavé (aplikační problémové úlohy jako jsou programovací jazyky lze řešit zápisem pomocí silných LL(k), resp. LL(1) gramatik s využitím zástupných jednosymbolových zápisů (nebo dobrou lexikální analýzou při předzpracování). Navíc existuje algoritmus, který libovolnou slabou LL(k) gramatiku převede na silnou LL(k). Jelikož **jazyk perspektivních přípon tvoří regulární jazyk**, vede SA pro slabé LL(k) gramatiky na vytvoření tzv. **charakteristického konečného automatu**, který se pak používá při SA. Vytvoření automatu zase vyžaduje aplikaci algoritmu na vytvoření **souboru tzv. LL(k) položek** a teprve s těmito informacemi můžeme provádět deterministickou SA. Celý postup je oproti předchozím algoritmům poměrně složitý a zdlouhavý. Čtenáři s hlubším zájmem o tato spíše teoreticky zajímavá témata lze doporučit literaturu [Ch84], [Ce92].

Nejdůležitější probrané pojmy:

- silné LL(k) gramatiky a (slabé) LL(k) gramatiky
- Funkce $FIRST_k$ a $FOLLOW_k$
- SA pro silné LL(k) gramatiky

Úkol k textu:

Sestrojte BKG generující nekonečný jazyk, která je silná LL(3) a zároveň není silná LL(2) gramatika. Dokažte splnění těchto podmínek a následně vytvořte rozkladovou tabulku a analyzujte libovolné slovo délky 4.

Nejdůležitější probrané pojmy:

- LL(1) gramatika a její rozkladová tabulka
- Funkce FIRST
- FIRST-FIRST kolize
- FIRST-FOLLOW kolize

Korespondenční úkol:

Sestrojte bezkontextovou gramatiku a Backusovu-Naurovou formu pro jednoduchý programovací jazyk složený ze seznamu řádků s příkazy. Řádek je uvozen návěštím ve tvaru X: příkaz, kde X je přirozené číslo. Lze používat identifikátory, které začínají písmenem anglické abecedy a obsahují libovolně mnoho písmen a číslic. Příkazy které se mohou použít jsou následující:

- přiřazovací příkaz tvaru X = aritmetický výraz, kde X je identifikátor a aritmetický výraz je výraz obsahující operandy – identifikátory a dále přirozená čísla, operace sčítání (+), odčítání (-), násobení (*) a dělení (/) a také umožňují vnořovat podvýrazy pomocí závorek.
- Nepodmíněný skok tvaru > X, kde X je návěští řádku, na který se má skočit.
- Podmíněný skok tvaru ?X$Y, kde X je identifikátor a Y je návěští a význam tohoto příkazu je, že se skočí na Y pouze pokud hodnota identifikátoru X je nula.
- Příkaz ukončení běhu programu - !

Pozn. I když pro řešení tohoto to nepotřebujeme vědět, protože používáme pouze syntaxi jazyka, všechny operace „ořezávají“ výsledné číslo na nezáporné hodnoty.

Pro Vámi sestrojenou LL(1) gramatiku proveďte kontrolu, zda je LL(1) a následně sestrojte rozkladovou tabulku a analyzujte jednoduchý ukázkový program:

```
10:X=5
20:Y=1
30:Y=X*Y
40:X=X–1
50 ?X$70
60:>30
70:!
```

Silné a slabé LL(k) gramatiky

10 Syntaktická analýza zdola nahoru

Cíl:

Po prostudování této kapitoly pochopíte:
- Způsob analýzy „zdola nahoru“ pomocí rozkladové tabulky
- Princip LR gramatik
- Funkce BEFORE a EFF

Průvodce studiem

Studium této kapitoly by pro vás mělo být spíše doplňující. Na analýzu zdola nahoru se zatím nebudeme do hloubky zaměřovat. Doporučuji sledovat pozorně příklady a teprve poté nezbytnou teorii. Věnujte této kapitole cca 4 hodin.

V minulých kapitolách a vlastně v celém textu se věnujeme především analýze „shora dolů“. Tento způsob je vhodný zejména pro vlastní přímou implementaci, protože je poměrně jednoduchý a přehledný i pro neautomatizované zpracování (což platí zejména až do třídy LL(1) gramatik). Přesto alespoň v krátkosti zmíníme o druhém způsobu deterministických analýz v lineárním čase a ten je založen na SA pomocí rozkladových tabulek způsobem „zdola nahoru“. Tento způsob je v jistém ohledu více obecný (jak udivíme u vlastností LR jazyků) a využívá se ve větší míře pro automatické generování analyzátorů pomocí počítačových programů. Platí se za to ovšem mnohem složitější konstrukcí rozkladových tabulek, což v případě automatizované tvorby není překážkou.

10.1 Model analýzy „zdola nahoru“

Analýza „zdola nahoru“

Model analýzy **„zdola nahoru“** je založen na postupných redukcích řetězců zpětně podle pravidel gramatiky a přesunech symbolů do zásobníku. V jistém smyslu jde tedy o duální postup oproti analýze „shora dolů“, kde naopak provádíme expanzi v směru pravidel. Podobně jako pro analýzu „shora dolů“ je možné uvažovat o obecném principu SA pomocí zásobníkového automatu. Takto sestrojený zásobníkový automat (potenciálně nedeterministický) by pracoval podle následujících tří typů pravidel):

1. Přesun symbolu vstupního slova na zásobník.
2. Redukce řetězce na vrcholu zásobníku zpětně podle přepisovacího pravidla.
3. Přijetí v případě, že jsme přečetli celé slovo a na zásobníku jsme vytvořili S – počáteční neterminál.

Formálně lze uvedený postup popsat jako alternativu k důkazu **Chyba! Nenalezen zdroj odkazů.**

Mějme bezkontextovou gramatiku G=(Π,Σ,S,P).
Sestrojíme ZA *M* tak, že L(G)=L_{PZ}(*M*).
Položíme *M* = ({*p*},Σ,Π∪Σ∪{#},δ,*p*, #,∅).
Pro δ platí:
δ(p,a,e)={(p,a)}; ∀a ∈ Σ
δ(p,e,α)={(p,A)|(A→ α) ∈ P}; ∀X ∈ Π
δ(p,e, S#)={(p,e)}

Je však nutno rozšířit definici ZA tak, aby umožňoval přepisy celých řetězců na zásobníku, nikoliv pouze symboly!

Takto sestrojený ZA má tři typy pravidel – buď přepisuje řetězec na neterminál, ukládá terminální symboly na zásobník nebo v případě, že přečte celé slovo a na zásobníku zbývá právě S a počáteční symbol #, provede přijetí slova. Obecně je automat nedeterministický – tedy musí si najít správnou cestu. Pokusme se sestrojit takový automat pro gramatiku z příkladu 1.

Řešený příklad 60:

G = ({S, A, B},{x, *, +, (,)}, S, P)
P: S $\rightarrow_1$ A + S, S $\rightarrow_2$ A
A $\rightarrow_3$ B * A, A $\rightarrow_4$ B
B $\rightarrow_5$ (S), B $\rightarrow_6$ x

Zásobníkový automat sestrojený tímto principem bude následující:
M = ({*p*},Σ,Π∪Σ∪{#},δ,*p*, #,∅), kde
Σ ={x, *, +, (,)}, Π = {S, A, B, #}
δ :

1. δ(p,e, A + S)=(p, S)
2. δ(p,e,A)=(p, S)
3. δ(p,e,B * A)=(p, A)
4. δ(p,e,B)=(p, A)
5. δ(p,e, (S))=(p, B)
6. δ(p,e,x)=(p, B)

a dále pravidla pro přesun a ukončení
P1. δ(p,x,e)=(p,x), P2. δ(p,*,e)=(p,*), P3. δ(p,+,e)=(p,+),
P4. δ(p, (, e)=(p,( ), P5. δ(p,), e)=(p,)), U. δ(p,e,#S)=(p,e)

Tento zásobníkový automat můžeme dále využít pro rozpoznávání řetězce
např. x * x (pozor je potřeba vzít úvahu, že řetězec čteme jako zrcadlově obrácené slovo!).

(p, x * x, #) $\Rightarrow_{P1}$ (p, x * , x #) $\Rightarrow_6$ (p, x * , B #) $\Rightarrow_4$ (p, x * , A #)
$\Rightarrow_{P2}$ (p, x , * A #) $\Rightarrow_{P1}$ (p, e , x * A #) $\Rightarrow_6$ (p, e , B * A #)
$\Rightarrow_3$ (p, e , A #) $\Rightarrow_2$ (p, e , S #) $\Rightarrow_U$ (p, e , e)

Slovo jsme tedy úspěšně rozpoznali, ovšem analýza vykazuje opět velmi silné znaky nedeterminismu – v mnoha případech by bylo možné udělat buď přesun nebo redukci. Právě aby k tomuto nedeterminismu nedocházelo, je potřeba zavést speciální typy gramatik – LR gramatiky. Opět budeme moci rozlišit silné a slabé LR(k) podle toho, zda k jejich SA potřebujeme znát pouze určitou část následujícího analyzovaného slova nebo i informaci o dosavadním průběhu analýzy. V tomto textu se budeme zabývat jen silnými LR(k) gramatikami – slabé LR(k) gramatiky se potýkají se stejnými problémy jako slabé LL(k) gramatiky (nutnost tvorby položek apod.)

10.2 LR(k) gramatiky a jejich syntaktická analýza

Pro definici silné LR(k) gramatiky je potřeba mít k dispozici podobné funkce jako pro LL(k) gramatiky. Z principu SA pro LR gramatik však tyto funkce fungují odlišným způsobem. Jde o funkce BEFORE (analogie s FOLLOW), která určuje které symboly předcházejí určitému neterminálu a funkci EFF „e-free FIRST“, která má význam symbolů nacházejících se na začátku řetězce.

Funkce BEFORE a EFF

Definice 58: Máme neterminál X a řetězec $\alpha \in (\Pi \cup \Sigma)^*$ v G=(Π,Σ,P,S), pak platí:

$BEFORE(X) = \{Y \mid S \Rightarrow^* \alpha Y X \beta, Y \in (\Pi \cup \Sigma)^* \} \cup \{\# \mid S \Rightarrow^* X\beta\}$

$EFF_k(\alpha) = \{w \mid w \in FIRST_k(\alpha)$ a existuje pravá derivace $\alpha \Rightarrow^* \beta \Rightarrow^* wx$ taková, že pro β neplatí $\beta = Awx\}$

Do množiny EFF tedy patří všechny řetězce z FIRST, které byly derivované derivací $\alpha \Rightarrow^* \beta \Rightarrow^* wx$ tak, že první neterminál v β nebyl nahrazený e.

Silná LR(k) gramatika

Definice 59: BKG G=(Π,Σ,S,P) se nazývá silná LR(k) gramatika, pokud pro rozšířenou gramatiku G'=(Π∪{S'},Σ,S',P∪{S'→S}) platí pro každé dvojice pravidel v P':

1.

- **a. A→αX, B→βX,**
- **b. A→αX, B→e a X∈BEFORE(B)**
- **c. A→e, B→e, X∈BEFORE(B) a X∈BEFORE(A)**

Pak $FOLLOW_k(A) \cap FOLLOW_k(B) = \varnothing$.

2.

- **a. A→αX, B→αXγ,**
- **b. A→e, B→αXγ a X∈BEFORE(A)**
- **c. A→e, B→γ, X∈BEFORE(A) a X∈BEFORE(B)**

Pak $FOLLOW_k(A) \cap EFF_k(\gamma FOLLOW_k(B)) = \varnothing$.

Podmínka 1. zabezpečuje, že pro redukci je možné se rozhodnout o pravidle na základě řetězce na k symbolů (b. a c. jsou případy, kdy se jeden nebo oba řetězce vypustí a pak je nutno zkoumat jen situaci, kdy jsou předcházející řetězce totožné). Podmínka 2. určuje jednoznačnost provedení redukce nebo přesunu.

Pro tyto gramatiky lze s pomocí výše uvedených funkcí sestrojit rozkladovou tabulku, která bude mírně odlišné struktury než u LL(k) gramatik. Bude totiž obsahovat v řádcích neterminály i terminály, což vyplývá z faktu, že na zásobníku se mohou redukovat řetězce obou druhů symbolů. V jednotlivých buňkách pak budou symboly reprezentující redukce, přesuny a přijetí.

Algoritmus 9: Vytvoření rozkladové tabulky pro silnou LR(k) gramatiku.

Rozkladová tabulka pro silnou LR(k)gramatiku

Vstup: LR(k) gramatika $G=(\Pi,\Sigma,S,P)$.
Výstup: rozkladová tabulka M pro G definovaná na $(\Pi\cup\Sigma\cup\{\#\}) \times \Sigma^{*k}$.
Postup:
Nejprve vytvoříme novou ekvivalentní gramatiku:

$G'=(\Pi\cup\{S'\},\Sigma,S',P\cup\{S'\rightarrow S\})$

Rozkladovou tabulku sestrojíme podle následujících pravidel:

1. $M(X, u)$ = redukce (i), pokud $A\rightarrow \alpha X$ je i-té pravidlo v P a $u\in FOLLOW_k(A)$.
2. $M(X, u)$ = redukce (i), pokud $A\rightarrow e$ je i-té pravidlo v P, $X\in BEFORE(A)$ a $u\in FOLLOW_k(A)$.
3. $M(S, e)$ = přijetí.
4. $M(X, u)$ = přesun, pokud $B\rightarrow \beta X\gamma \in P$ a $u\in EFF_k(\gamma FOLLOW_k(B))$.
5. $M(X, y)$ = chyba v ostatních případech (tyto přechody není třeba vypisovat – jako jejich ekvivalent slouží prázdný přechod).

Podmínka 1. určuje redukce pokud je X na zásobníku následuje v řetězci právě ta kombinace symbolů, která může následovat za neterminálem v daném pravidle pro redukci (A). Podmínka 2. je obdobná, ale jelikož pravidlo může vypustit A, zajímá nás, co se může vyskytnout před ním. Podmínka 3. reprezentuje situaci, že jsme celý řetězec úspěšně zredukovali až na S a 5. je situace, kdy se analýza neúspěšně zastaví. Podmínka 4. popisuje přesuny, které se mohou vykonat v případech, kdy symboly ve slově se potenciálně shodují s tím, co následuje za X (tedy takové přesuny povedou v budoucnu možná k redukci – jinak to nemá smysl).

Abychom mohli provádět syntaktickou analýzu, je potřeba ještě nadefinovat modifikovaný algoritmus SA.

Algoritmus 10: Syntaktická analýza pro silné LR(k) gramatiky

SA pro silnou LR(k)gramatiku

Vstup: rozkladová tabulka M pro silnou LR(k) gramatiku $G=(\Pi,\Sigma,S,P)$, vstupní řetězec $w \in \Sigma^*$.

Výstup: pravý rozklad (derivace) vstupního řetězce v případě, že $w \in L(G)$, jinak chybová signalizace.

Postup:

- Algoritmus čte vstupní řetězec, používá zásobník a má k dispozici slovo u = $FIRST_k(x)$, kde x je dosud nepřečtená část řetězce, symbolem X označíme vrchol zásobníku. (vrchol zásobníku je na konci slova)
- Počáteční situace je (w, #, e).
- Vykonávají se přechody 1. a 2. dokud nenastane situace 3. nebo 4.
  1. **Redukce:** Pokud M(X, u) = redukce (i), vyloučíme ze zásobníku α, které je na pravé straně pravidla $A \rightarrow \alpha$. Pokud na zásobníku nebyl řetězec α, nastává chybová signalizace a analýza končí, jinak číslo i je připojeno k posloupnosti reprezentující pravý rozklad a do zásobníku zařadíme řetězec A.
  2. **Přesun**: Pokud M(X, u) = přesun, přečte se vstupní symbol a uloží se na vrchol zásobníku.
  3. **Přijetí:** Pokud M(X,u) = přijetí, řetězec je rozpoznán, analýza končí a π obsahuje posloupnost pravidel reprezentující pravou derivaci řetězce podle G.
  4. **Chyba:** ve všech ostatních případech analýza končí s chybovou signalizací.

Řešený příklad 61:

Mějme LR(1) gramatiku pro generování aritmetických výrazů se sčítáním, násobením, vnořenými výrazy se závorkami a operandem x:
G = ({S, A, B, C, D},{+,*,(,), x}, S, P)
P: $S \rightarrow_1 AB$, $A \rightarrow_2 S+$, $A \rightarrow_3 e$, $B \rightarrow_4 CD$, $C \rightarrow_5 B*$, $C \rightarrow_6 e$,
$D \rightarrow_7 (S)$, $D \rightarrow_8 x$

Gramatiku rozšíříme dále o pravidlo $S' \rightarrow_0 S$. Vytvoříme tabulku, kde použijeme následující zkratky: P – přesun, R(i) – redukce (i), A – přijetí.

M	x	+	*	(	)	e
S		P			P	A
A	R(6)			R(6)		
B		R(1)	P		R(1)	R(1)
C	P			P		
D		R(4)	R(4)		R(4)	R(4)
x		R(8)	R(8)		R(8)	R(8)
+	R(2)			R(2)		
*	R(5)			R(5)		
(	R(3)			R(3)		
)		R(7)	R(7)		R(7)	R(7)
#	R(3)			R(3)		

Při konstrukci této tabulky jsme použili hodnoty funkce BEFORE(A) = {#, (}, BEFORE(C) = {E‘}

Nyní zanalyzujme slovo x * x + x:

(x * x + x , # , e) ⇒ (x * x + x , #A , 3) ⇒ (x * x + x , #AC , 36) ⇒
(* x + x , #ACx , 36) ⇒ (* x + x , #ACD , 368) ⇒
(* x + x , #AB , 3684) ⇒ (x + x , #AB* , 3684) ⇒
(x + x , #AC , 36845) ⇒ (+ x , #ACx , 36845) ⇒
(+ x , #ACD , 368458) ⇒ (+ x , #AB , 3684584) ⇒
⇒ (+ x , #S , 36845841) ⇒
(x , #S+ , 36845841) ⇒ (x , #A , 368458412) ⇒
(x , #AC , 3684584126) ⇒ (e , #ACx , 3684584126) ⇒
(e , #ACD , 36845841268) ⇒ (e , #AB , 368458412684) ⇒
(e , #S , 3684584126841) ⇒ Přijetí

Slovo bylo rozpoznáno a pravá derivace je reprezentována řetězcem 3684584126841.

Kromě silných LR(k) gramatik existují ještě jejich další podtřídy a nadtřídy. Nebudeme uvádět jejich přesné definice, pouze si provedeme jejich stručný výčet.

1. **Slabé LR gramatiky** – jsou definovány analogickou podmínkou jako slabé LL gramatiky, tj. není kladeno omezení na pravidla, ale na jednoznačnost odvození. Z toho plyne mnohem obtížnější SA a konstrukce rozkladových tabulek podobně jako slabých LL(k) gramatik.
2. **LR(0) gramatiky** – využívají při SA informaci pouze o průběhu SA a nepotřebují znát žádnou část řetězce.
3. **Jednoduché LR(k) gramatiky** – tyto tzv. SLR(k) gramatiky umožňují díky omezení jednodušeji konstruovat rozkladové tabulky.

10.3 Vlastnosti LR jazyků

Podobně jako LL jazyků můžeme formulovat některé důležité vlastnosti.

LR jazyky

Definice 60: Bezkontextový jazyk L se nazývá LR(k) jazyk, pokud existuje LR(k) gramatika G, taková že L = L(G). Bezkontextový jazyk L se nazývá LR jazyk, pokud existuje LR(k) gramatika G pro nějaké k ≥ 0 taková, že L = L(G).

To jestli je nějaký jazyk LR resp. LR(k) jazyk tedy závisí na existenci příslušného typu gramatiky generující daný jazyk.

První vlastností opět zaručuje jednoznačnost.

Věta 40: Každá LR(k) gramatika je jednoznačná.

Věta 41: Každá LL(k) gramatika je LR(k) gramatika.

Zajímavou vlastností je, že každá LL(k) gramatika je LR(k) gramatika a naopak nikoliv. To znamená, že LR gramatiky jsou obecnější třídou gramatik než LL gramatiky – což jim dává větší expresivitu. Ovšem to je za cenu komplikovanější SA, resp. tvorby tabulek.

Nejdůležitější probrané pojmy:

- LR gramatiky a jazyky
- Funkce BEFORE a EFF
- Vlastnosti LR jazyků

11 LL a LR jazyky

Cíl:

Po prostudování této kapitoly pochopíte:
- vlastnosti LL gramatik a jazyků
- zobecnění principů LL jazyků

Naučíte se:
- transformovat gramatiky z předcházejících kapitol
- využít tyto transformace na problémových aplikačních úlohách pro snadnější manipulaci s gramatikami a jazyky

Průvodce studiem

Studium této kapitoly vám ukáže důležité postupy, které můžete využít i v praktickém návrhu gramatik pro překladače. Ukážeme si některé postupy, které se uplatňují při tvorbě překladačů. Doporučuji sledovat pozorně příklady a teprve poté nezbytnou teorii. Věnujte této kapitole cca 6 hodin.

I když jsme v předcházejících kapitolách kladli důraz především na algoritmy (postupy), jak provádět deterministickou – tedy implementovatelnou – SA, zavedli jsme rovněž mnoho nových tříd jazyků. Jednalo se postupně o třídu jazyků generovaných SLL(1) gramatikami, q-gramatikami, LL(1) gramatikami, silnými a slabými LL(k) gramatikami. Už v minulém díle opory jste poznali, že vlastnosti tříd jazyků jsou důležité, neboť vám dávají možnost poznat a uvědomit si principy a smysl jejich použití. Příkladem mohou být uzávěrové vlastnosti regulárních a bezkontextových jazyků nebo Chomského hierarchie jazyků. Podobné vlastnosti nyní budeme stručně zkoumat (pouze se slovní formulací myšlenky důkazu) u LL jazyků. Věnujte, prosím, této kapitole rovněž vlekou pozornost. I když se vám může zdát na první pohled méně důležitá, naopak její význam je pro pochopení a přehled o celém učivu klíčový.

11.1 Vlastnosti LL jazyků

Nejprve musíme přesně definovat, co to vlastně je LL jazyk, i když o tom asi již máte jistou představu.

Definice 61: Bezkontextový jazyk L se nazývá LL(k) jazyk, pokud existuje LL(k) gramatika G, taková že L = L(G). Bezkontextový jazyk L se nazývá LL jazyk, pokud existuje LL(k) gramatika G pro nějaké k ≥ 0 taková, že L = L(G).

LL jazyky

To jestli je nějaký jazyk LL resp. LL(k) jazyk tedy závisí na existenci příslušného typu gramatiky generující daný jazyk.

První vlastností, která je důležitá pro SA je jednoznačnost ve smyslu definice z prvního dílu opory.

Věta 42: Každá LL(k) gramatika je jednoznačná.

Jednoznačnost gramatiky je dána existencí pouze jedné levé derivace pro každé slovo jazyka. Jelikož pro každou LL(k) gramatiku platí podmínka Definice 56: musí každé odvození být jednoznačně určeno předponou řetězce na k-symbolů.

Při konstrukci LL(k) gramatiky je třeba dodržet základní podmínku a tou je vlastnost, že nemůže být zleva rekurzivní. Zleva rekurzivní je taková gramatika, která umožňuje z neterminálu A generovat řetězec, který začíná jím samým tedy opět A.

Věta 43: Žádná LL(k) gramatika není zleva rekurzivní.

Pokud by gramatika byla zleva rekurzivní, pak umožňuje generovat sekvence $A \Rightarrow^* A\alpha$. α se může přepsat buď na prázdné slovo a v tom případě tedy můžeme A odvodit různými derivacemi nebo na terminální slovo v a A na terminální slovo u, pak můžeme v různých odvozeních odvození generovat stále znovu slovo začínající na uv^{k+i} . To bylo ve sporu s jednoznačností gramatiky, protože každá LL(k) gramatika je jednoznačná. Je-li tedy gramatika zleva rekurzivní (což je z pravidel ihned zjistitelné) zjevně nemůže být LL(k).

Další vlastnost souvisí s pojmy algoritmické rozhodnutelnosti. Jde o vlastnosti, které blíže zkoumá teorie vyčíslitelnosti – čtenář se může seznámit s oporou [Pa02]. Pro naše účely tento pojem zjednodušíme do programátorské roviny (není problém si udělat paralelu mezi konkrétním programem v třeba v Pascalu a algoritmem zapsaným jiným způsobem). Představte si, že máte napsat program (algoritmus), který pro nějaké objekty řekne zda platí jistá vlastnost nebo ne. Musí to tedy fungovat pro jakýkoliv objekt dostanete na vstup a vždy musíte dostat na výstupu jasnou odpověď ano nebo ne. Pokud takový algoritmus existuje, říkáme že problém je rozhodnutelný. V opačném případě se jedná o problém nerozhodnutelný. Příkladem rozhodnutelného problému může být existence reálných kořenů pro kvadratickou rovnici. Jistě byste dokázali napsat velmi jednoduchý program, který by pro dané koeficienty kvadratické rovnice a,b,c (objekt) dokázal obecně spočítat determinant a pokud by byl nezáporný, vrátili byste odpověď ANO a v opačné případě NE.

Věta 44: Pro danou BKG a dané pevné k≥0 je rozhodnutelné, zda gramatika je LL(k) nebo ne.

Je zřejmé, že je jednoduché ověřit zda daná gramatika je silná LL(k). Stačí si spočítat příslušné množiny $FIRST_k$ a $FOLLOW_k$ a pak ověřit podmínky. Složitější je situace u slabých LL(k), ale i zde je možné vytvořit soubor LL(k) položek, vytvořit rozkladovou tabulku a pokud tato tabulka nikde neobsahuje dvě expanze pro jednu buňku, pak je LL(k).

Věta 45: Pro danou BKG je nerozhodnutelné, zda je LL(k) pro nějaké k≥0.

Postup řešení tohoto problému, který nás asi napadne jako první, je zkoušet zda gramatika je LL(1) a pokud ne, zkusit zda je LL(2) a tak dále… Pokud skutečně gramatika pro nějaké k je LL(k), pak to zjistíme (viz předchozí věta). Problém této myšlenkové konstrukce je, že nebude fungovat pokud gramatika není LL(k) pro žádné k. Pak se vlastně postup zacyklí do nekonečné smyčky a stále bude zvyšovat k do nekonečna. Postup už nám tedy nedá spolehlivě odpověď. Uvědomme si, že tato vlastnost je zásadní pro práci s LL tvary gramatik. Pokud dostaneme gramatiku, nejsme schopni jednoznačně pro každý případ ověřit, zda gramatika je vůbec LL(k) gramatika. Ještě horší je však vlastnost následující.

Věta 46: Pro danou BKG G, která není LL(k) pro dané pevné k, je nerozhodnutelné, zda k ní existuje ekvivalentní gramatika, která je LL(k).

Nemáme tedy jistotu, že obecnou BKG můžeme převést vždy na LL(k) gramatiku. Samozřejmě tento pesimistický teoretický výsledek nás ještě nemusí odradit od úsilí, převést obecnou BKG na LL(k) nebo dokonce LL(1) gramatiku. Už v předchozím textu jste viděli, že pro stejný nebo podobný problém lze někdy sestrojit různé typy gramatik. Pro převody na LL(k) gramatiky existuje několik technik, které jsme při konstrukci gramatik už v některých případech používali. Není samozřejmě s ohledem na předchozí věty zaručeno, že budou fungovat vždy, ale pro většinu praktických aplikačních úloh vedou k cíli.

Další zajímavou vlastností je, že u LL(1) gramatik je jedno zda aplikujeme podmínku silné nebo slabé LL(k) gramatiky – jde o ekvivalentní podmínky.

Věta 47: BKG je silnou LL(1) gramatikou právě tehdy, když je (slabou) LL(1) gramatikou.

Dále je jasné, že pokud gramatika splňuje podmínky pro LL(k) gramatiku, splňuje zároveň i podmínky pro LL(k+1) gramatiku. To znamená, že když je gramatika jednoznačná na k symbolů už je jednoznačná na libovolně mnoho symbolů, kterých je více než k. Zároveň platí, že každá silná LL(k) gramatika je i slabá LL(k), ale naopak ne (viz příklad slabé a silné LL(2) gramatiky) z předchozí kapitoly.

Věta 48: Pokud je BKG silná LL(k) gramatika, pak je i slabá LL(k).

Věta 49: Pokud je BKG LL(k) gramatika, pak je i LL(k+1).

Tato vlastnost vytváří hierarchii LL(k) jazyků, kde jsou do sebe postupně vnořeny třídy LL(1), LL(2), … LL(k), …jazyků navíc ještě každá třída se skládá z vnořené třídy silných LL(k) jazyků.

Poměrně elegantně lze zapsat schéma pro gramatiku, která je LL(k) a není LL(k-1).

Řešený příklad 62:

G = ({S, A},{a,b}, S, P)
P: $S \rightarrow aT$, $A \rightarrow c$, $A \rightarrow bB$, $T \rightarrow SA$, $T \rightarrow A$, $B \rightarrow b^{k-1}d$, $B \rightarrow e$

Tato gramatika je LL(k) lze rozhodnout na základě k následujících symbolů ve slově, zda použít pravidlo $B \rightarrow b^{k-1}d$, kde je na k-tém místě d nebo zda se použije pravidlo s epsilon a tím se dá možnost vygenrovat sekvenci symbolů b pomocí $A \rightarrow bB$. Ale na k-1 symbolů už nejsme schopni toto rozhodnutí vždy provést.

Velmi jednoduše lze také podat příklad jazyka, který není LL(k) pro žádné k. Jde o jazyk $L = \{a^n b^n | \; n \geq 0 \} \cup \{a^n c^n | \; n \geq 0 \}$.

U tohoto jazyka nelze sestrojit gramatiku, která by obecně pro počáteční symboly a dokázala jednoznačně derivovat buď na ukončovací b nebo c. Symbolů a může být na počátku potenciálně libovolný počet, což neumožňuje sestrojit obecně gramatiku pro všechna slova tohoto jazyka.

11.2 Transformace na LL gramatiky

Přestože jsme v minulé podkapitole konstatovali, že existují jazyky, ke kterým nelze sestrojit LL(k) gramatiku pro žádné k, existují rovněž jednoduché transformační techniky, které umožňují v některých případech převést gramatiku na LL(k), resp. LL(1) gramatiku. Dalším převodem, který lze dokonce provést univerzálně, je transformace LL(1) gramatiky na q-gramatiku. Všechny uvedené převodní techniky formulujeme jak teoreticky, tak vyzkoušíme na praktických příkladech.

Při převodu LL(1) gramatiky na q-gramatiku využíváme tzv. techniku rohové substituce. U LL(1) gramatiky je kolidujícím faktorem ve vztahu ke q-gramatice možnost existence neterminálu na počátku pravidla. Tohoto faktoru se lze zbavit, pokud postupně dosadíme (substituujeme) řetězce za tyto neterminály, tak že začneme od řetězců, které již začínají terminálem. Nejprve si popišme vlastní rohovou substituci.

Rohová substituce

Věta 50: Mějme BKG $G=(\Pi,\Sigma,S,P)$ a pravidlo $A \rightarrow B\alpha$ v P, kde $B \in N$ a $B \rightarrow \beta_1 | \ldots | \beta_n$ jsou všechna pravidla pro B. Vytvoříme gramatiku $G_1=(\Pi,\Sigma,S,P_1)$ vyloučením pravidla $A \rightarrow B\alpha$ a přidáním pravidel $A \rightarrow \beta_1\alpha | \ldots | \beta_n\alpha$. Pak platí $L(G) = L(G_1)$ a pokud G je LL(k) gramatika,

pak i G_1 je LL(k) gramatika. Tato transformace se nazývá rohová substituce.

Postupnou aplikací rohové substituce na uspořádané neteriminály v pořadí, kde žádný neterminál s vyšším indexem nepřipisuje na řetězec začínající neterminálem s nižším indexem, dojdeme až ke q-gramatice, pokud původní gramatika byla LL(1).

Věta 51: Pokud BKG G=(Π,Σ,S,P) je LL(1) gramatika, pak existuje q-gramatika G'=(Π,Σ,S,P') a platí, že L(G) = L(G').

Algoritmus 11: Převod LL(1) gramatiky na q-gramatiku.

Převod na q-gramatiku

Vstup: LL(1) gramatika G=(Π,Σ,S,P).
Výstup: q-gramatika G'=(Π,Σ,S,P'), kde L(G) = L(G').

Metoda:

1. Zavedeme uspořádání neterminálů, aby platilo podmínka: $A_i \rightarrow A_j\alpha$, pak i < j. A tedy N={$A_1$, …, A_n}.(Toto uspořádání lze provést pro každou LL(1) gramatiku.)
2. Položíme i = n – 1 a P' = P.
3. Pokud i = 0, pak máme gramatiku G', jinak pokračujeme bodem 4.
4. Každé pravidlo $A_i \rightarrow A_j\alpha$ v P', kde j>i, nahradíme pravidly $A_i \rightarrow \beta_1\alpha \mid \ldots \mid \beta_n\alpha$, kde $A_j \rightarrow \beta_1 \mid \ldots \mid \beta_n$ (rohová substituce).
5. Pokud všechna přidaná pravidla začínají terminálem, pokračujeme krokem 6., jinak se vrátíme na krok 4.
6. Nechť i = i –1 a pokračuje krokem 3.

Aplikujme nyní tento algoritmus na gramatiku z předchozího textu.

Řešený příklad 63:

G = ({S, A, B},{x, +, (,)}, S, P)
P: $S \rightarrow_1 BA$, $A \rightarrow_2 + BA$, $A \rightarrow_3 e$, $B \rightarrow_4 x$, $B \rightarrow_5 (S)$

Krok 1: Neterminály uspořádáme například takto
$A_1 = S$, $A_2 = A$, $A_3 = B$.
Krok 2: i = 2, P = P'.
Krok 3: pokračujeme 4.
Krok 4: neexistuje pro A žádné pravidlo vyhovující dané podmínce.
Krok 5: viz 4., pokračujeme 6.
Krok 6: i = 1, pokračujeme 3.
Krok 3: pokračujeme 4.
Krok 4: provedeme rohovou substituci $S \rightarrow_1 BA$ nahradíme v P' $S \rightarrow_{1a} xA$ a $S \rightarrow_{1b} (S)A$.
Krok 5: vše začíná terminálem a pokračujeme 6.
Krok 6: i = 0, pokračujeme 3.
Krok 3: i = 0 a tedy máme q-gramatiku s pravidly:

P': $S \rightarrow_{1a} xA$ a $S \rightarrow_{1b} (\ S\ )A$, $A \rightarrow_2 +\ BA$, $A \rightarrow_3 e$, $B \rightarrow_4 x$, $B \rightarrow_5 (\ S\ )$

Tvar q-gramatiky může být v některých případech výhodnější pro SA než tvar LL(1) gramatiky a někdy tomu může opačně. Nevýhodou LL(1) gramatiky je delší odvození, protože dochází k postupným přepisům přes neterminály (viz příklad). Naopak výhodou je větší přehlednost gramatiky díky nižšímu počtu pravidel. V duálním pohledu se stejně můžeme dívat na q-gramatiku, která navíc může zredukovat některé neterminály, ovšem za cenu možnosti vzniku velkého množství pravidel a tím nepřehlednosti gramatiky. U praktických aplikací musíte tedy sami zvážit, co je pro vás prioritou v konkrétním případě.

Převody na LL(1) gramatiky z obecných bezkontextových gramatik nelze samozřejmě realizovat vždy (jak už vyplynulo z dřívějšího textu). Existují ale techniky, které v některých případech toto umožňují. Samozřejmě, že jich existuje více než zde uvedeme - jde o následující vybrané metody:

1. Odstranění levé rekurze.
2. Levá faktorizace.
3. Rohová substituce.
4. Pohlcení terminálního symbolu.
5. Pohlcení řetězce.

Ad 1.

Odstranění levé rekurze je operace při které se snažíme zabránit situaci, která se v žádném případě nemůže u LL(1) gramatiky vyskytnout a to je, že neterminál přepisuje na řetězec začínající jím samým. Pomocí následující věty lze tuto situaci odstranit zavedením nového neterminálu.

Odstranění levé rekurze

Věta 52: Nechť $G=(\Pi,\Sigma,S,P)$ je bezkontextová gramatika.
Nechť $\{X \rightarrow X\alpha_1, X \rightarrow X\alpha_2, \ldots X \rightarrow X\alpha_r\}$ ($\alpha_i \in (\Pi \cup \Sigma)^*$, $1 \le i \le r$) je množina pravidel s levou rekurzí, ve kterých se na pravé straně úplně vlevo nachází neterminál X. Nechť $\{X \rightarrow \beta_1, X \rightarrow \beta_2, \ldots X \rightarrow \beta_s\}$ ($\beta_i \in (\Pi \cup \Sigma)^*$ jsou zbývající pravidla pro X mající na levé straně neterminál X. Nechť $G_1=(\Pi \cup \{Z\},\Sigma,S,P_1)$ je bezkontextová gramatika, která vznikla přidáním neterminálu Z k Π a dál nahrazením všech pravidel s levou rekurzí pravidly:
$X \rightarrow \beta_i$, pro $1 \le i \le s$
$X \rightarrow \beta_i Z$, pro $1 \le i \le s$
$Z \rightarrow \alpha_i$, pro $1 \le i \le r$
$Z \rightarrow \alpha_i Z$, pro $1 \le i \le r$
Pak $L(G_1)=L(G)$. Tuto operaci nazveme odstranění levé rekurze.

Poznámka: Uvědomme si, že všechna pravidla s levou rekurzí gramatiky G generují regulární množinu
$\{\beta_1,\beta_2,\ldots\beta_s\}\{\alpha_1,\alpha_2,\ldots\alpha_r\}^*$
což je právě množina generovaná v G_1 pravidly, která mají na levé straně neterminál X nebo Z. Vlastně nahradíme pomocí nového neterminálu Z levou

rekurzi, která umožňuje generovat iteraci řetězců α_i převedením na rekurzi, která ovšem již není levou rekurzí, protože Z není prvním neterminálem v pravidlech se Z na levé straně.

Ad 2.

Levá faktorizace je operace, při která se snažíme zabránit situaci, kdy nám dvě nebo více pravidel začínají stejným řetězcem. To samozřejmě znamená, že FIRST těchto pravidel není disjunktní množina a tedy, že to nemůže být LL(k) gramatika. Odstranění je poměrně jednoduché a spočívá v jakémsi „vytknutí" tohoto řetězce a zavedení nového neterminálu, který bude přepisovat na zbytky řetězců. To samozřejmě může buď vyřešit situaci anebo přinést s sebou novou kolizi (FIRST-FIRST nebo FIRST-FOLLOW).

Věta 53: Nechť G=(Π,Σ,S,P) je bezkontextová gramatika.
Nechť $\{X\rightarrow \alpha\alpha_1, X\rightarrow \alpha\alpha_2, \ldots X\rightarrow \alpha\alpha_r\}$ ($\alpha_i \in (\Pi\cup\Sigma)^*$, $1 \le i \le r$) je množina pravidel s stejným řetězcem na začátku. Nechť $G_1=(\Pi\cup\{Z\},\Sigma,S,P_1)$ je bezkontextová gramatika, která vznikla přidáním neterminálu Z k Π a dál nahrazením všech pravidel se stejným řetězcem na začátku pravidla:
$X\rightarrow\alpha Z$,
$Z\rightarrow\alpha_i$, pro $1 \le i \le r$
Pak $L(G_1)=L(G)$. Tato operace se nazývá levá faktorizace.

Levá faktorizace

Ad 3.

Rohovou substituci jsme již probrali v rámci převodu na q-gramatiku.

Ad 4.

Pohlcení terminálního symbolu nám pomáhá vyřešit situaci, kdy za neterminálem následuje řetězec, kterým může některé pravidlo od tohoto neterminálu začínat. Řeší tedy FIRST-FOLLOW kolizi a to tak, že kolidující terminál spojí s tímto neterminálem, čímž vznikne nový neterminál reprezentující toto spojení.

Pohlcení terminálu

Věta 54: Nechť G=(Π,Σ,S,P) je bezkontextová gramatika.
Nechť $X\rightarrow \alpha Ba\beta$ je pravidlo, kde za neterminálem B následuje terminál a a pro B platí, že $B\rightarrow \alpha_1, X\rightarrow \alpha_2, \ldots X\rightarrow \alpha_r$. Nechť $G_1=(\Pi\cup\{Z\},\Sigma,S,P_1)$ je bezkontextová gramatika, která vznikla přidáním neterminálu [Ba] k Π a dál nahrazením pravidla $X\rightarrow\alpha Ba\beta$ souborem pravidel:
$X\rightarrow\alpha[Ba]\beta$,
$[Ba]\rightarrow\alpha_i a$, pro $1 \le i \le r$
Pak $L(G_1)=L(G)$. Tato operace se nazývá pohlcení terminálního symbolu.

Ad 5.

Podobnou operací jako je pohlcení terminálu je pohlcení celého řetězce. Tuto operaci vzhledem k podobnosti s 4. nebudeme formalizovat.

Nyní se pokusme vybrané operace demonstrovat na příkladech.
Řešený příklad 64:

Mějme gramatiku pro generování seznamu abstraktních příkazů a bloků ve složených závorkách oddělených středníkem.

G = ({S, A},{p, ; , {, } }, S, P)
P: S $\rightarrow_1$ A, S $\rightarrow_2$ S;A,
A $\rightarrow_3$ p, A $\rightarrow_4$ {S}

Tato gramatika nemůže být LL(1), neboť obsahuje levou rekurzi. Provedeme její převod pomocí odstranění levé rekurze. Zavedeme nový neterminál Z a zrušíme pravidlo s levou rekurzí.

G = ({S, A, Z},{p, ; , {, } }, S, P)

S $\rightarrow_1$ A, S $\rightarrow_2$ AZ,
A $\rightarrow_3$ p, A $\rightarrow_4$ {S},
Z $\rightarrow_5$;AZ, Z $\rightarrow_6$;A.

Tato gramatika již neobsahuje levou rekurzi, ovšem obsahuje dvě FIRST-FIRST kolize u neterminálu S a Z. Pokusíme se je tedy odstranit levou faktorizací. Zavedeme nové neterminály X a Y, které vzniknou její aplikací.

G = ({S, A, Z, X, Y},{p, ; , {, } }, S, P)

S $\rightarrow_1$ AX,
A $\rightarrow_2$ p, A $\rightarrow_3$ {S},
Z $\rightarrow_4$;AY,
X $\rightarrow_5$ e, X $\rightarrow_6$ Z,
Y $\rightarrow_7$ Z, Z $\rightarrow_8$ e.

U pravidel 1. – 4. není již žádná kolize. Avšak u neterminálu X a Y může být kolize FIRST-FOLLOW. Nejprve spočítáme FOLLOW(X) = { } , e} a FOLLOW(Y) = { }, e }.

Platí, že FOLLOW(X) $\cap$ FIRST(Z) = { }, e } $\cap$ { ; } = $\varnothing$ a rovněž FOLLOW(Y) $\cap$ FIRST(Z) = { }, e } $\cap$ { ; } = $\varnothing$.

Gramatika tedy je LL(1) gramatika.

Existují ale samozřejmě i případy, kdy nám levá faktorizace nezaručí, že dostaneme LL(1) gramatiku.

Řešený příklad 65:

G = ({S, A, B},{a, x, y, z }, S, P)
P: S $\rightarrow_1$ aAxx, S $\rightarrow_2$ aByy, S $\rightarrow_3$ zy, S $\rightarrow_4$ zx,
A $\rightarrow_5$ aAx, A $\rightarrow_6$ z, B $\rightarrow_7$ aBy, B $\rightarrow_8$ z

Po levé faktorizaci dostaneme gramatiku:

G = ({S, A, B, X, Y},{a, x, y, z }, S, P)
P: S $\rightarrow_1$ aX, S $\rightarrow_2$ zY,
X $\rightarrow_3$ Axx, X $\rightarrow_4$ Byy,
Y $\rightarrow_5$ x, Y $\rightarrow_6$ y,
A $\rightarrow_7$ aAx, A $\rightarrow_8$ z, B $\rightarrow_9$ aBy, B $\rightarrow_{10}$ z

Tato gramatika není LL(1) gramatika, protože
FIRST(Axx)$\cap$FIRST(Byy) = {a, z}.

V určitých případech je potřeba před vlastní faktorizací provést ještě rohovou substituci.

Řešený příklad 66:

G = ({S, A, B},{a, b, c, d}, S, P)
P: S $\rightarrow$ aA, S $\rightarrow$ BA,
A $\rightarrow$ cA, A $\rightarrow$ d, B $\rightarrow$ aB, B $\rightarrow$ bA

Tato gramatika není LL(1), protože FIRST(aA)$\cap$FIRST(BA) = {a} a taktéž nemůžeme provést faktorizaci. Abychom faktorizaci symbolu a mohli provést, je potřeba nejprve provést rohovou substituci B.

S $\rightarrow$ aA, S $\rightarrow$ aBA, S $\rightarrow$ bAA,
A $\rightarrow$ cA, A $\rightarrow$ d, B $\rightarrow$ aB, B $\rightarrow$ bA

Nyní se již může faktorizovat zavedením nového neterminálu X.

G = ({S, A, B, X},{a, b, c, d}, S, P)

S $\rightarrow$ aX, S $\rightarrow$ bAA,
X $\rightarrow$ A, X $\rightarrow$ BA,
A $\rightarrow$ cA, A $\rightarrow$ d, B $\rightarrow$ aB, B $\rightarrow$ bA

Tato gramatika už je LL(1).

Nyní se zaměříme na odstraňování kolize FIRST-FOLLOW pomocí pohlcování symbolů.

Řešený příklad 67:

G = ({S, A, B},{a, b, c}, S, P)

$S \rightarrow AaB$, $A \rightarrow e$, $A \rightarrow aaB$,
$B \rightarrow c$, $B \rightarrow bB$

FIRST(aaB)∩FOLLOW(A) = {a} a tedy nejde o LL(1) gramatiku.

Tuto kolizi můžeme odstranit pomocí pohlcení terminálu a, který kolizi způsobuje jeho pohlcením neterminálem A.

G = ({S, A, B, [Aa]},{a, b, c}, S, P)

$S \rightarrow [Aa]B$, $A \rightarrow e$, $A \rightarrow aaB$,
$B \rightarrow c$, $B \rightarrow bB$, $[Aa] \rightarrow a$, $[Aa] \rightarrow aaBa$

Tím ovšem vznikla FIRST-FIRST kolize posledních dvou pravidel, kterou musíme dále řešit pomocí faktorizace zavedením X. Také lze zredukovat neterminál A, protože se již v gramatice nedá od S přepsat.

G = ({S, B, [Aa], X},{a, b, c}, S, P)

$S \rightarrow [Aa]B$, $B \rightarrow c$, $B \rightarrow bB$, $[Aa] \rightarrow aX$, $X \rightarrow e$, $X \rightarrow aBa$

V této gramatice FOLLOW(X) = {b, c} a tedy není zde FIRST-FOLLOW kolize.

Vhodnou kombinací transformačních technik tedy můžete (ale nemusíte) dospět k LL(k), resp. LL(1) gramatice.

Nejdůležitější probrané pojmy:
- hierarchie LL(k) jazyků
- transformační techniky (odstranění levé rekurze, rohová substituce, levá faktorizace, pohlcení terminálu, pohlcení řetězce)

Úkol k textu:

1. Sestrojte BKG pro syntaktickou strukturu formule predikátové logiky, kde se mohou vyskytovat n-nární predikáty pojmenované symboly p, q, r a dále mohou obsahovat termy – buď vnořené funktory f, g, h s n argumenty nebo konstanty a, b, c nebo symboly pro proměnné x, y, z. Povolené logické spojky jsou konjunkce (∧), disjunkce(∨) a negace(¬) podle standardní definice predikátové logiky. Lze samozřejmě vnořovat i závorkované formule.

   Příklad: $(p(x,f(y,g(c))) \vee \neg r(b)) \wedge q(z,y)$
2. Vámi sestrojenou gramatiku převeďte na LL(1) gramatiku.
3. Sestrojte rozkladovou tabulku pro tuto LL(1) gramatiku a zanalyzujte formuli z příkladu 1.

12 Obecné algoritmy syntaktické analýzy

Cíl:

Po prostudování této kapitoly se stručně seznámíte s metodami SA a jejich implementace (výhody a nevýhody):

- Obecné metody pro BKG – Earleyho algoritmus, CYK algoritmus
- Zásobníkový automat

Z teoretického hlediska jsme se nyní věnovali především rozkladovým tabulkám pro LL, resp. LR gramatiky. Samozřejmě existuje mnoho způsobu pro implementaci SA a ty mají své výhody a nevýhody z hlediska **implementace na prostředcích pro automatizaci** (zejména na počítačích). Jejich důkladnější rozbor je již spíše **náplní kurzu překladače**, neboť k překladačům neoddělitelně syntaktická analýza patří jako jejich podstatná a nezbytná součást. Přesto se v následující kapitole, ale alespoň velmi stručně zmíníme o aspektech některých metod SA. Zejména půjde o možnosti **implementace datových struktur, časovou a prostorovou náročnost**.

12.1 Obecné algoritmy analýzy

Časová složitost jakékoliv metody – algoritmu – hraje v informatice klíčovou úlohu [Pa02]. Praktická realizace algoritmu musí být dostatečně „rychlá“ a nesmí spotřebovat „enormně mnoho paměti“, abychom s ní v praxi uspěli. Jistě znáte optimalizační problémy typu „Problém obchodního cestujícího“, kdy neznáme dostatečně efektivní klasický algoritmus pro jeho řešení. Samozřejmě existují různé moderní metody pro hledání optimálního řešení založené například na evolučních technikách, ale klasický deterministický algoritmus má vždy exponenciální složitost, což jej pro praxi činí nepoužitelným od určité velikosti problému (počtu měst, které má obchodní cestující navštívit).

Earleyho algoritmus

Pro syntaktickou analýzu obecných bezkontextových gramatik existují algoritmy s mnohem lepší funkcí časové složitosti – s kubickou složitostí, resp. kvadratickou pro jednoznačné gramatiky. Prvním z nich je tzv. **Earleyho analyzátor (algoritmus).** Je založen na tečkové notaci, podobně jako algoritmy pro výpočet FIRST a FOLLOW. Jde o algoritmus z rodiny tzv. grafových algoritmů. Efektivní je zejména u gramatik s levou rekurzí. Popišme si nyní jeho způsob výpočtu.

Algoritmus je založen na tečkové notaci, tedy pravidlo $A \rightarrow B.CD$ s tečkou před C reprezentuje situaci, že B již bylo analyzováno a zbývá zanalyzovat CD. Pro každou vstupní pozici analyzovaného řetězce vytvoříme množinu stavů, které reprezentují kombinaci dvou prvků:

1. Pravidlo s tečkovou notací
2. Pozice, na které pravidlo začalo – výchozí stav.

Stav na vstupní pozici k se nazývá S(k). Počáteční stav analýzy je S(0). Analyzátor vykonává iterativně 3 typy operací:

1. Predikci: Pro každý stav v S(k) tvaru (X → α .Y β, j), kde j je výchozí stav, přidej (Y → . γ, k) do S(k) pro každé pravidlo s Y na levé straně.
2. Čtení: Pokud máme a jako další symbol v analyzovaném řetězci, pak pro každý stav v S(k) ve tvaru (X → α .a β, j), přidáme (X → α a . β, j) do S(k+1).
3. Ukončování: Pro každý stav v S(k) tvaru (X → α ., j) najdeme stavy v S(j) tvaru (Y → α . X β, i) a přidáme stavy (Y → α X . β, i) do S(k).

Algoritmus stále přidává nové stavy, dokud lze nové přidat. To lze implementovat například pomocí fronty ještě nevyřešených stavů. Tento postup vlastně simuluje procházení gramatiky na základě příslušných symbolů ve vstupu, podobně jako je tomu při konstrukci množiny FIRST.

Nyní se podívejme na řešený příklad, převzatý z [Wi05].

Řešený příklad 68:

Mějme gramatiku pro generování aritmetických výrazů se sčítáním a násobením a čísly jako operandy.

```
P -> S      # startovací pravidlo
S -> S + M | M
M -> M * T | T
T -> number
```

Vstupní řetězec: 2 + 3 * 4

Generované stavy:

```
== S(0): • 2 + 3 * 4 ==
(1)  P -> • S          (0)   # startovací pravidlo
(2)  S -> • S + M      (0)   # predikce z (1)
(3)  S -> • M          (0)   # predikce z (1)
(4)  M -> • M * T      (0)   # predikce z (3)
(5)  M -> • T          (0)   # predikce z (3)
(6)  T -> • number     (0)   # predikce z (5)

== S(1): 2 • + 3 * 4 ==
(1)  T -> number •     (0)   # čtení z S(0)(6)
(2)  M -> T •          (0)   # ukončování z S(0)(5)
(3)  M -> M • * T      (0)   # ukončování z S(0)(4)
(4)  S -> M •          (0)   # ukončování z S(0)(3)
(5)  S -> S • + M      (0)   # ukončování z S(0)(2)
(6)  P -> S •          (0)   # ukončování z S(0)(1)

== S(2): 2 + • 3 * 4 ==
(1)  S -> S + • M      (0)   # čtení z S(1)(5)
(2)  M -> • M * T      (2)   # predikce z (1)
```

(3) M -> • T (2) # predikce z (1)
(4) T -> • number (2) # predikce z (3)

== S(3): 2 + 3 • * 4 ==
(1) T -> number • (2) # čtení z S(2)(4)
(2) M -> T • (2) # ukončování z S(2)(3)
(3) M -> M • * T (2) # ukončování z S(2)(2)
(4) S -> S + M • (0) # ukončování z S(2)(1)
(5) S -> S • + M (0) # ukončování z S(0)(2)
(6) P -> S • (0) # ukončování z S(0)(1)

== S(4): 2 + 3 * • 4 ==
(1) M -> M * • T (2) # čtení z S(3)(3)
(2) T -> • number (4) # predikce z (1)

== S(5): 2 + 3 * 4 • ==
(1) T -> number • (4) # čtení z S(4)(2)
(2) M -> M * T • (2) # ukončování z S(4)(1)
(3) M -> M • * T (2) # ukončování z S(2)(2)
(4) S -> S + M • (0) # ukončování z S(2)(1)
(5) S -> S • + M (0) # ukončování z S(0)(2)
(6) P -> S • (0) # ukončování z S(0)(1)

Vidíte, že máme i poměrně jednoduchý algoritmus pro obecné BKG a navíc jeho časová složitost (zvláště pro jednoznačné gramatiky) je vcelku přijatelná (kvadratická složitost se považuje vzhledem k exponenciální složitosti optimalizačních problémů za zvládnutelnou). Skrytý problém této konstrukce je však v její paměťové náročnosti. Už v tomto jednoduchém příkladě je jasné, že vzniká velké množství stavů, které je potřeba ukládat v nějaké datové struktuře. Proto se zdá být rozumnější (pokud to lze) používat omezené třídy jazyků z minulých kapitol, kde máme algoritmy SA s lineární časovou složitostí a navíc bez nutnosti uchovávat potenciálně rozsáhlé datové struktury.

CYK algoritmus

Dalším z rodiny SA pro obecné BKG je známý **Cocke-Younger-Kasami (CYK) algoritmus** nebo také (CKY). Tento algoritmus ve své nemodifikované verzi dokáže rozpoznat pro každou BKG v Chomského normální formě (každou BKG lze poměrně jednoduše převést na CHNF – viz první díl textu), zda slovo patří do jazyka generovaného touto gramatikou. Jde o implementaci metod tzv. dynamického programování (metody využívají rozklad na podproblémy a jejich optimalizaci, např. naivní výpočet Fibonnaciho čísel prostou rekurzí vs. výpočet na základě ukládání dílčích hodnot pro menší argumenty, což optimalizuje výpočet, protože není nutné znovu počítat již dříve vyčíslené hodnoty).

CKY algoritmus má rovněž v nejhorším případě kubickou časovou složitost a je důležitý zejména z teoretického hlediska, protože dává důkaz o rozhodnutelnosti problému příslušnosti slova do BKJ.

12.2 Zásobníkový automat

Zásobníkový automat je přirozeným kandidátem na roli syntaktického analyzátoru. I když dokáže rovněž analyzovat obecný BKJ, jeho nedeterministická verze vyžaduje určitou metodu simulace, abychom dostali implementovatelný deterministický postup. Právě protože je jeho myšlenka příliš jednoduchá vede obecně při deterministické simulaci na exponenciální složitost , což je pro praxi nepoužitelné. Nicméně jeho implementace je poměrně jednoduchá – jde vlastně o zásobník s pravidly, které lze zapsat do vícerozměrného pole.

Nejdůležitější probrané pojmy:

- analýza obecných BKG (Earleyho algoritmus, CYK algoritmus)

13 Implementace algoritmů syntaktické analýzy

Cíl:

Po prostudování této kapitoly se stručně seznámíte s metodami SA a jejich implementace (výhody a nevýhody):

- Rozkladové tabulky
- Rekurzivní sestup

13.1 Rozkladové tabulky

Rozkladové tabulky pro LL a LR gramatiky, které jsme detailně rozebírali v tomto textu mají svou výhodu v jednoduché implementaci a lineární časové složitosti analýzy, jsou-li jednou vytvořeny. Tabulku lze jednoduše implementovat jako vícerozměrné pole nebo dynamickou strukturu a pak v ní jen hledat příslušnou kombinaci symbol ve vstupní slově a symbol na vrcholu zásobníku a provést danou akci. To provádíme iterativně až do dosažení přijetí nebo chybové signalizace.
Na druhou stranu pro danou gramatiku je po sestrojení tabulky a implementaci poměrně složité provést rozšíření. Například kdybychom chtěli místo celých čísel v gramatice použít čísla reálná, znamená to přebudovat celou rozkladovou tabulku na základě nové gramatiky. Navíc je rozkladová tabulku pro člověka téměř nečitelná ve smyslu logiky daného jazyka. Kupříkladu byste asi těžko jen na základě rozkladové tabulky dokázali určit jaký jazyk generuje – u výchozí gramatiky to může být přece jen více zřejmé.
Pro ruční implementaci (myslí se tím, vlastní tvorba tabulky a její zápis a použití ve formě vlastního počítačového programu) jsou poměrně vhodné LL(1), resp. LL(k) gramatiky. Naopak tvorba LR analyzátorů je velmi pracná a nepřehledná takže pro ruční implementaci jsou nevhodné. Ale díky větší obecnosti se v praxi více používají automatické generátory (hotové profesionální aplikace) založené na LR gramatikách. Tato oblast je náplní teorie a praxe překladačů, ale pro zájemce je možné dát odkaz na existující a dobře známé programy jako je LL-gen (pro LL gramatiky) nebo více používaný YACC či Bison.

13.2 Metoda rekurzivního sestupu

Rekurzivní sestup

Velice oblíbenou, jednoduchou a přehlednou metodou vedoucí přímo ke zdrojovému kódu je **metoda rekurzivního sestupu** (**Recursive Descent Parsing**) [Le02]. Metoda je založena na principu analýzy „shora dolů“ a je tedy blízká LL gramatikách.
Metoda rekurzivního sestupu spočívá v konstrukci procedur strukturovaného programovacího jazyka přesně dle Backusovy-Naurovy formy (BNF), kde je každému neterminálu přiřazena jedna procedura a je volána procedura GetChar (načítající vždy následující symbol slova) před každým terminálem. Výskyt neterminálu v pravidle je v proceduře nahrazen rekurzivně voláním příslušné

procedury. Iterace a podmínka v Backus-Naurově formě je nahrazena jednoduše jejich programátorskými protějšky. Vyžaduje se, aby jazyk byl definován LL(k) gramatikou. Z hlediska časové složitosti jde opět o obecně neefektivní metodu s exponenciální časovou složitostí, nicméně pro jednoduché gramatiky z praxe je použitelná a zejména je její výhodnou vysoká čitelnost kódu ve vztahu k výchozí gramatice. Navíc existuje modifikace tzv. **packrat parser**, která pro omezenou třídu takzvaných **parsing expression grammars** pracuje v lineární čase. Také je tento postup flexibilní, protože umožňuje kdykoliv změnit a přidat syntaktický element bez nutnosti měnit celý kód, ale pouze dotčenou část gramatiky (například změna struktury číslo z celého čísla na reálné znamená pouze změnu procedury reprezentující tento element). Podívejme se nyní na příklad gramatiky pro generování aritmetických výrazů se sčítáním, násobením, číslicemi a vnořenými závorkovanými strukturami.

Řešený příklad 69:

Gramatiku v Backusově-Naurově formě pro náš zjednodušený příklad (pouze s číslicemi místo identifikátorů) lze zapsat takto:

<Výraz> ::= <Term> { + <Term>}
<Term> ::= <Faktor>{ * <Faktor>}
<Faktor> ::= 0|1|2|…|9|(<Výraz>)

Nyní se schématicky pokusíme ukázat (nejde o zcela hotový kód), jak bychom sestrojili SA metodou rekurzivního sestupu pro tuto gramatiku v jazyce Pascal. Tento kód, pak umožňuje nejen SA, ale i detekci možných chyb. Nejprve sestrojíme proceduru, která zapouzdřuje celou činnost SA. Její hlavička může vypadat například takto:

```
procedure SyntaktickaAnalyza(infix:string;var err,pos:word);
      {procedura analyzuje aritmetický výraz infix, err obsahuje číslo chyby, pos
obsahuje pozici ,kde analýza skončila}
```

Používají se proměnné infixpos (pozice aktuálně čteného znaku ze vstupu), ch (aktuální znak).

Analyzátor dále obsahuje nezbytný lexikální analyzátor pro načítaní jednotlivých symbolů (v našem zjednodušeném případě jde o jednoznakové symboly). Lexikální analýza je realizována procedurou GetChar, která ukládá znak do proměnné ch a případně provede detekci chybové situace err=2, pokud načteme zcela nepřípustný znak.

```
procedure Getchar;      {čte znak z infixu do proměnné ch}
begin
 if  err=0 then
   begin
    Inc(infixpos);
    if infixpos<=Length(infix) then ch:=infix[infixpos] else ch:=#0;
    ch:=Upcase(ch);
            if not((ch in cislice)or(ch in ['(',')','*','+'])) then err:=2;      {ošetření
        nežádoucích znaků}
   end;
end;
```

Jádrem analyzátoru jsou jednotlivé rekurzivní procedury Výraz (sčítaní), Term (násobení), Faktor (číslice, vnořený závorkovaný výraz). Výraz přesně podle BNF buď volá podřízený Faktor nebo čte terminální symboly.

```
procedure Vyraz;              {výraz s nižší prioritou}
begin
  if err=0 then
    begin
      Term;
      while (ch='+') do
        begin
              Getchar;                {sčítání}
              Term;
        end;
    end;
end;

procedure Term;               {výraz s vyšší prioritou}
begin
  if err=0 then
    begin
      Faktor;
      while (ch='*') do
        begin
              Getchar;                {násobení}
              Faktor;
        end;
    end;
end;
```

A dále musíme sestrojit proceduru pro Faktor, která bude mít vzhledem k jinému charakteru přepisovaného řetězce i jiný kód.

```
procedure Faktor;   {synt. analýza operandu}
begin
  if err=0 then
    begin
      case ch of
      '0'..'9':
         begin                        {anal. číslic}
           Getchar;
         end;
      '(':begin
            Getchar;
                                {analýza výrazu se závorkou}
            Vyraz;
            if (ch<>')')and(err=0) then err:=4   {chyba- není ukončen závorkou}
            else if err=0 then
              begin
                Getchar;
              end;
         end;
      else if err=0 then err:=5;    {nebyl detekován ani vyraz,ani číslice}
      end;
    end;
```

Faktor tedy rozlišuje dvě situace – buď jde o číslici nebo jde o výraz začínající závorkou a ukončený opačnou závorkou. Logicky tedy můžeme odhalit další

dvě chyby (err=4, když chybí závorka, err=5, když není detekován ani výraz ani číslice).
Pozn. Samozřejmě, že chybové detekce by mohly odhalit ještě další problematické konstrukce – např. skončení nejvyššího volání procedury Výraz před přečtením posledního znaku apod.

Ilustrujme nyní průběh výpočtu procedury Výraz na výrazu 5 + 3 * 2.

Infixová notace	**Aktuální znak**	**Aktuální procedura**	**Návrat do procedury**
5 + 3 * 2	5	Výraz	
5 + 3 * 2	5	Term	
+ 3 * 2	+	Faktor	Term
+ 3 * 2	+	Term	Výraz
3 * 2	3	Výraz (+)	
3 * 2	3	Term	
* 2	*	Faktor	Term
2	2	Term (*)	
		Faktor	Term (*)
		Term (*)	Výraz (+)
		Vyraz (+)	

13.3 Jiné metody

Existují samozřejmě i jiné metody implementace syntaktických analyzátorů. Za zmínku stojí například metoda spojových seznamů, která vychází z toho, že přepisovací pravidlo pro neterminál si lze představit jako seznam neterminálů a terminálů reprezentujících řetězec, na který pravidlo přepisuje. Jednotlivé přepisovací pravidla pro daný neterminál tedy lze implementovat jako spojový seznam, který obsahuje buď uzly terminální, kdy srovnáváme symboly s analyzovaným řetězcem nebo neterminály, které obsahují pouze odkaz (ukazatel) na jiný spojový seznam, který reprezentuje daný neterminál. Principiálně jde vlastně o analogii rekurzivního sestupu, ale u této implementace není nutná rekurzivita navzájem vnořených procedur a tedy i volání. Iterativnost analýzy namísto rekurzivity může být výhodná zvláště pro velmi složité gramatiky a také je její velká potenciální síla v možnosti měnit gramatiku za běhu programu. Jistě si dokážete představit „elastický“ informační systém, který by umožňoval dokonce vytvářet nové syntaktické struktury svých vstupů, případně programovací jazyk, který byste takto mohli interaktivně rozšiřovat o nové konstrukce bez nutnosti zasahovat přímo do zdrojového kódu jeho překladače!

Nejdůležitější probrané pojmy:
- analýza obecných BKG (Earleyho algoritmus, CYK algoritmus)
- metoda rekurzivního sestupu

Úkol k textu:

Pokuste se promyslet, jak byste u příkladu na metodu rekurzivního sestupu odhalili chybu typu „chybí operátor“. Dále se pokuste vytvořit spojové seznamy pro syntaktickou analýzu pro stejný jazyk jako u metody rekurzivního sestupu.

Korespondenční úkol:

Sestrojte BNF pro jazyk logických výrazů výrokové logiky, kde můžete používat konjukci, disjunkci, implikaci, negaci a dále symboly a..z pro výrokové proměnné, symboly 0 a 1 pro true a false a rovněž můžete do závorek vnořit výraz stejného typu.
Pro sestrojenou gramatiku vytvořte metodou rekurzivního sestupu program v libovolném strukturovaném programovacím jazyce (např. Pascal), který bude provádět syntaktickou analýzu libovolné formule a jejich chybovou detekci.

Literatura

[Ce92] ČEŠKA, Milan, RÁBOVÁ, Zdena. Gramatiky a jazyky. Brno, VUT 1992.
[Ch84] CHYTIL, Milan. Automaty a gramatiky. Praha, SNTL 1984.
[Ho79] HOPCROFT, J.E., ULLMAN, J. D. Introduction to Automata theory, Languages and Computation. Addison-Wesley, Reading (Mass.), 1979
[Hr01] HŘIVŇÁK, J. Formální jazyky a automaty. Graduační práce na : Ostravská Univerzita, PřF. 2001 (interně dostupný v síti OU)
[Ja97] JANČAR, Petr. Teorie jazyků a automatů. VŠB TU Ostrava, Dokument dostupný na URL: http://www.cs.vsb.cz/jancar/ (leden 2003)
[Ja97a] JANČAR, Petr. Vyčíslitelnost a složitost. VŠB TU Ostrava, Dokument dostupný na URL: http://www.cs.vsb.cz/jancar/ (leden 2003)
[Pa02] PAVLISKA, Viktor: Vyčíslitelnost a složitost I. distanční studijní text OU, 2002
Na Internetu lze najít množství odkazů a materiálů – především v angličtině, nicméně existují i české a slovenské webovské stránky s materiály.