Abeceda = lubovolna konecna mnozina  
Prvky abecedy nazyvame znaky / symboly / pismena  
Slovo(retazec) nad abecedou ∑ je lubovolna konecna postupnost znakov nad touto abecedou  
Prazdna postupnost znakov = prazdne slovo Ԑ  
Pocet clenov postupnosti v = |v|  
Pocet znakov a v slove v = #a(v)  
Jazyk nad abecedou ∑ je lubovolna mnozina slov nad ∑  
Mnozina vsetkych slov nad abecedou ∑ = ∑\*  
Mnozina vsetkych neprazdnych slov = ∑+  
=> Jazyky nad ∑ su prave podmnoziny ∑\*

Binarna operacia zretazenia, oznacovana . - u . v = uv, operacia je asociativna – u.(v.w) = (u.v).w  
Ԑ sa chova ako jednotkovy prvok => Ԑ . u = u . Ԑ = u

Slovo u je podslovom slova v, ak existuju slova x,y take, ze v=x.u.y  
 -ak x=Ԑ => slovo u je predponou(prefixom) slova v  
 -ak y=Ԑ => slovo u je priponou(sufixom) slova v

Unarna operacia i-tej mocniny slova, pre kazde i ∊ N0, kde u je lubovolne slovo:  
 -u0 = Ԑ  
 -ui+1 = u.ui

Nech L je jazyk nad abecedou ∑, K je jazyk nad abecedou Δ, vysledkom je vzdy jazyk nad abecedou  
∑ ∪ Δ pri operaciach:  
 -Standardne mnozinove operacie: -zjednotenie(∪), -prienik(∩), -rozdiel(\)  
 -Zretazenim jazykov L a K je jazyk L.K = {u.v | u ∊ L, v ∊ K}

{} . L = L . {} = {}, {Ԑ} . L = L . {Ԑ} = L

i-ta mocnina jazyka L pre i ∊ N0:  
 -L0 = {Ԑ}  
 -Li+1 = L . Li

{}0 = {Ԑ}  
{}i = {}, pre lubovolne i ∊ N  
{Ԑ}i = {Ԑ}, pre lubovolne i ∊ N0

Iteracia jazyka L:  
    
{}\* = {Ԑ}

Pozitivna iteracia jazyka L:  
  
{}+ = {}

Doplnok jazyka L je jazyk: **co – L = ∑\* \ L**

Zrkadlovy obraz slova w = a1….an je slovo **wR = an…a1**, (ԐR = Ԑ)

Zrkadlovy obraz jazyka: **LR = {wR | w ∊ L}**

Nech L je trieda jazykov a o n-arna operacia nad jazykmi. Trieda L je uzavrena na o, ak pre lubovolne   
jazyky L1,…,Ln patriace do L plati, ze tiez jazyk o(L1,…,Ln) patri do L.

**Gramatiky**

Gramatika = popis jazyka pomocou pravidiel, podla ktorych sa vytvaraju vsetky slova daneho jazyka

Gramatika je stvorica: (N, ∑, P, S):  
N = neprazdna konecna mozina neterminalnych symbolov (neterminalov)  
∑ = konecna mnozina terminalov taka, ze N ∩ ∑ = {}, mnozina vsetkych symbolov gramatiky: V = N ∪ ∑  
P ⊆ V\* NV\* x V\* = konecna mnozina pravidiel  
S ∊ N = specialny pociatocny neterminal

Gramatika urcuje relaciu =>G priameho odvodenia na mnozine V\*  
γ =>G δ prave vtedy, ked existuje pravidlo α -> β ∊ P, a slova η,ϱ ∊ V\* take, ze γ = ηαϱ a δ = ηβϱ

**Vetna forma** gramatiky = kazdy retazec, z mnoziny V\*, ktory sa da odvodit z pociatocneho neterminalu  
**Veta** gramatiky = kazda vetna forma, ktora obsahuje len terminaly  
Jazyk generovany gramatikou = mnozina vsetkych viet gramatiky  
 -L(G) = {w ∊ ∑\* | S =>\*G w}  
Gramatiky G1 a G2 su jazykovo ekvivalentne, prave vtedy, ked generuju rovnaky jazyk: L(G1) = L(G2)

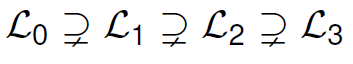
Chomskeho hierarchia gramatik  
-klasifikacia gramatik podla tvaru prepisovacich pravidiel

typ 0: pravidla v obecnom tvare (frazove gramatiky)  
typ 1: pre kazde pravidlo α -> β plati: |α|≤|β|, s vynimkou pravidla S -> Ԑ, pokial sa S nevyskytuje   
 na pravej strane ziadneho pravidla (kontextove gramatiky)  
typ 2: kazde pravidlo je tvaru A -> α, kde |α| ≥ 1, s vynimkou pravidla S -> Ԑ, pokial sa S nevyskytuje  
 na pravej strane ziadneho pravidla (bezkontextove gramatiky (bez Ԑ-pravidiel))  
typ 3: kazde pravidlo je tvaru A -> aB alebo A -> a s vynimkou pravidla S -> Ԑ, pokial sa S nevyskytuje  
 na pravej strane ziadneho pravidla (regularne gramatiky)

Chomskeho hierarchia jazykov  
-hierarchia gramatik urcuje hierarchiu jazykov

L0: trieda vsetkych rekurzivne spocitatelnych jazykov  
L1: trieda vsetkych kontextovych jazykov  
L2: trieda vsetkych bezkontextovych jazykov  
L3: trieda vsetkych regularnych jazykov

Jazyk L je typu 0 (rekurzivne spocitatelny), ak existuje gramatika G typu 0 taka, ze L(G) = L



**Konecne automaty**

Deterministicky konecny automat (DFA) M, je patica (Q,∑,δ,q0,F):  
Q = neprazdna konecna mnozina stavov  
∑ = konecna vstupna abeceda  
δ: Q x ∑ -> Q = parcialna prechodova funkcia  
q0 ∊ Q = pociatocny (inicialny) stav  
F ⊆ Q = mnozina koncovych (akceptujucich) stavov

DFA je jednoduche zariadenie, ktore sa sklada z citacej hlavy a vstupnej pasky. Citacia hlava sa moze   
nachadzat v jednom z konecneho poctu stavov, iduc, po vstupnej paske, cita vstupne slovo a podla   
precitanych symbolov meni svoj stav. Jej stav, po docitani celeho slova urcuje, ci bolo slovo dobre.

**Konfiguracia** DFA, je prvok (q,w) ∊ Q x ∑\*, kde q je stav automatu a w je nespracovana cast vstupneho  
slova, konfiguracia popisuje situaciu, v ktorej sa aktualne automat pocas vypoctu nachadza.

**Krok vypoctu** DFA je relacia  na konfiguraciach definovana: (q,av)  (p,v) ⬄ p = δ(q,a)  
Krok vypoctu hovori, v akej konfiguracii sa bude automat nachadzat v nasledujucom kroku vypoctu,   
ak vieme v akom stave sa nachadza teraz => vypocet automatu je postupnost konfiguracii, pricom  
kazde dve nasledujuce konfiguracie su v relacii “krok vypoctu”.

Rozsirena prechodova funkcia δ’: Q x ∑\* -> Q je parcialna funkcia definovana induktivne vzhladom k  
dlzke slova z ∑\*:   
 - δ’(q,Ԑ) = q pre kazdy stav q ∊ Q  
 - δ’(q,wa) = δ(δ’(q,w),a), ak je δ’(q,w) aj δ(δ’(q,w),a) definovane

Slovo w je akceptovane automatom M, prave ked: δ’(q0,w) ∊ F  
Jazyk akceptovany automatom M je: L(M) = {w ∊ ∑\* | δ’(q0,w) ∊ F}   
Jazyk, ktory je rozpoznatelny deterministickym konecnym automatom je regularny.  
Automaty M a M’ su ekvivalentne, prave ked: L(M) = L(M’)

Ku kazdemu DFA M existuje ekvivalentny DFA M’ s totalnou prechodovou funkciou

Synchronna paralelna kompozicia automatov  
-pre dane automaty M1 a M2 umoznuje zostrojit automat rozpoznavajuci prienik (zjednotenie, rozdiel)  
 jazykov L(M1) a L(M2)  
-Nech M1 = (Q1, ∑, δ1, q1, F1) a M2 = (Q2, ∑, δ2, q2, F2) a prechodove funkcie δ1,δ2 su totalne.  
 => Definujeme DFA M3 = (Q3, ∑, δ3, q3, F3), kde:  
 - Q3 = Q1 x Q2 = {(p,q) | p ∊ Q1, q ∊ Q2}  
 - F3 = F1 x F2 = {(p,q) | p ∊ F1, q ∊ F2}  
 - q3 = (q1,q2)  
 - δ3((p,q),a) = (δ1(p,a),δ2(q,a))

L(M3) = L(M1) ∩ L(M2)

Automat pre komplement  
K automatu M = (Q,∑,δ,q0,F) s totalnou prechodovou funkciou, zostrojime automat M’ rozpoznavajuci  
jazyk co – L(M): M’ = (Q,∑,δ,q0,Q\F)

Uvazme vypocet M na slove anbn, kde n > k a k je pocet stavov automatu M:  
Pretoze n > k, musi existovat stav p taky, ze pri citani inicialnej postupnosti symbolov a prejde automat  
stavom p aspon dva krat. (Je to odvodene z dirichletovho principu)

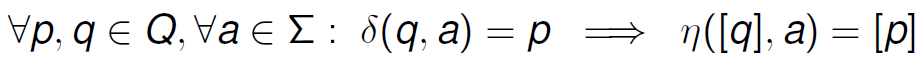
Pumping lemma  
Nech L je regularny jazyk, potom existuje n ∊ N take, ze lubovolne slovo w ∊ L dlzky aspon n, sa da  
napisat v tvare w = xyz, kde |xy| ≤ n, y ≠ Ԑ a xyiz ∊ L, pre kazde i ∊ N0

DFA M s totalnou prechodovou funkciou je minimalny konecny automat pre jazyk L(M), ak neexistuje  
ekvivalentny DFA s totalnou prechodovou funkciou a mensim poctom stavov.

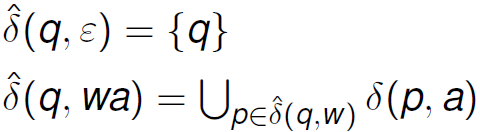
* Minimalny konecny automat akceptujuci regularny jazyk L je urceny jednoznacne az na   
  izomorfizmus (premenovanie stavov)

Stav q ∊ Q je **dosiahnutelny**, ak existuje w ∊ ∑\* take, ze δ’(q0,w)=q.  
Stav je **nedosiahnutelny**, ak nieje dosiahnutelny.

Stavy p,q su **jazykovo ekvivalentne**, p ≡ q, ak:  


**Reduktom automatu** M = (Q,∑,δ,q0,F) je konecny automat M/≡ = (Q/≡,∑,η,[q0],F/≡), kde:  
 -stavy su triedy rozkladu Q/≡  
 -prechodova funkcia η je funkcia splnajuca:   
   
 -pociatocny stav je trieda rozkladu Q/≡ obsahujuca stav q0 -koncove stavy sup rave tie triedy rozkladu Q/≡, ktore obsahuju aspon jeden koncovy stav

Nech je DFA M bez nedosiahnutelnych stavov s totalnou prechodovou funkciou, potom: L(M) = L(M/≡)

Nedeterministicky konecny automat (NFA) je M = (Q,∑,δ,q0,F), kde vyznam zloziek je rovnaky ako pri  
DFA, s vynimkou prechodovej funkcie δ. Ta je definovana ako (totalne) zobrazenie δ: Q x ∑ -> 2Q  
Rozsirena prechodova funkcia δ’: Q x ∑\* -> 2Q:  
 

Vyraz δ(q,a) = {p1,p2,…,pn} chapeme tak, ze automat sa v stave q pri citani symbolu **a** moze rozhodnut,  
do akeho stavu sa dostane po posune citacej hlavy.  
Pri NFA na danom slove moze prebiehat viacero roznych vypoctov, niektore mozu byt na danom slove   
akceptacne, niektore nie, slovo akceptujeme, ak aspon jeden vypocet na nom je akceptacny.  
NFA moze robit aj kroky bez toho, aby musel citat symbol zo vstupu (Ԑ-kroky)

**Krok vypoctu** NFA je relacia na konfiguraciach definovana: (q,an)(p,n) ⬄ p ∊ δ(q,a)

Jazyk prijimany NFA:  
   
**Krok vypoctu** NFA je relacia na konfiguraciach definovana: (q,an)(p,n) ⬄ p ∊ δ(q,a)

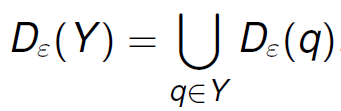
**Pre kazdy NFA existuje ekvivalentny DFA.**

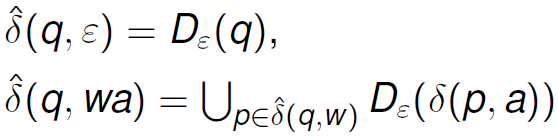
Pre NFA M = (Q,∑,δ,q0,F), konstruujeme DFA M1 = (Q1,∑,δ1,{q0},F1), kde:  
Q1 = 2Q, tj stavy automatu M1 su vsetky podmnoziny Q  
  
Mnozina koncovych stavov F1 je tvorena prave tymi podmnozinami Q, ktore obsahuju nejaky prvok   
mnoziny F.

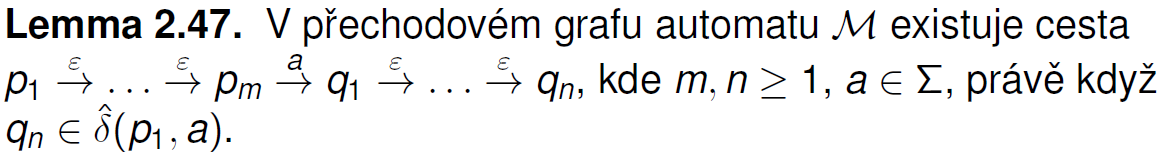
Pre kazde n ∊ N existuje NFA o n stavoch taky, ze ekvivalentny DFA ma aj po minimalizacii 2n stavov.

NFA s Ԑ-krokmi je M=(Q,∑,δ,q0,F), kde vyznam zloziek je rovnaky ako v definicii NFA s vynimkou   
prechodovej funkcie, ktora je definovana δ: **Q x (∑ ∪ {Ԑ}) -> 2Q**

Definovana funkcia (epsilonove okolie) DԐ: Q -> 2QDԐ(p) je najmensia mnozina X ⊆ Q taka, ze plati:  
 -p ∊ X  
 -pokial q ∊ X a r ∊ δ(q,Ԑ), potom tiez r ∊ X

Definicia prechodovej funkcie (epsilonovho okolia) DԐ na mnoziny stavov:  
 

Rozsirena prechodova funkcia: Q x ∑\* -> 2Q  
 



Jazyk prijimany automatom M s Ԑ-krokmi: **L(M) = {w ∊ ∑\* | δ’(q0,w) ∩ F ≠ 0}**

Ku kazdemu NFA M = (Q,∑,δ,q0,F) s Ԑ-krokmi existuje ekvivalentny NFA bez Ԑ-krokov.  
Konstrukcia M’ = (Q,∑,γ,q0,G) bez Ԑ-krokov:  
 γ(q,a) = δ’(q,a) pre kazde q ∊ Q, a ∊ ∑  
 G = F, pokial DԐ(q0)∩ F = {}, inak F ∪ {q0

Ku kazdemu NFA M =(Q,∑,δ,q0,F) existuje ekvivalentny NFA, ktory neobsahuje prechody na Ԑ a pre ktory  
plati [∀](http://en.wikipedia.org/wiki/Turned_a)q∊Q, [∀](http://en.wikipedia.org/wiki/Turned_a)x∊∑; |δ(q,x)|=1, hovori vlastne ze ku kazdemu NFA existuje ekvivalentny DFA.

Na DFA sa da pozerat ako na normalny tvar NFA.

Nech G je regularna gramatika a nech G0 = (N, ∑, P, S) je gramatika, pre ktoru plati P0=P \ {X -> X | X ∊ N},  
potom L(G0)=L(G)

Dokaz dvoch inkluzii rovnosti L(G0)=L(G):

⊆ : z predpokladu w ∊ L(G0) vyplyva, ze existuje odvodenie slova w v G0: S => u1 => u2 => … => un = w,  
toto odvodenie je zaroven aj odvodenim slova w v G, lebo P0 ⊆ P

⊇ : z predpokladu w ∊ L(G) vyplyva, ze existuje odvodenie slova w v G, kedze existuje odvodenie v G,   
potom existuje najkratsie odvodenie w v G, v tomto odvodeni sa nemohlo pouzit pravidlo tvaru X -> X,  
inak by sme vedeli najst kratsie odvodenie, preto toto odvodenie je aj odvodenim v G0

Ekvivalencia konecnych automatov a regularnych gramatik  
Konecne automaty akceptuju prave regularne jazyky.  
K lubovolnemu DFA M existuje regularna gramatika G taka, ze L(G)=L(M)  
K lubovolnej regularnej gramatike G existuje NFA M taky, ze L(M)=L(G)

Uzaverove vlastnosti triedy regularnych jazykov  
Trieda jazykov X je uzavreta na binarnu operaciu Y, ak pre vsetky jazyky L1,L2 ∊ X plati, ze L1 Y L2 ∊ X  
Trieda regularnych jazykov = R

Trieda R je uzavreta na:  
 -zjednotenie  
 -prienik  
 -rozdiel  
 -komplement  
 -zretazenie  
 -iteraciu  
 -reverze

Kleeneho veta: Kazdy konecny jazyk je regularny.  
Trieda R je najmensia trieda jazykov, ktora obsahuje vsetky konecne jazyky a je uzavreta na zjednotenie,  
zretazenie a iteraciu.  
Lubovolny jazyk je popisatelny regularnym vyrazom prave, ked je rozpoznatelny konecnym automatom.

Regularne vyrazy

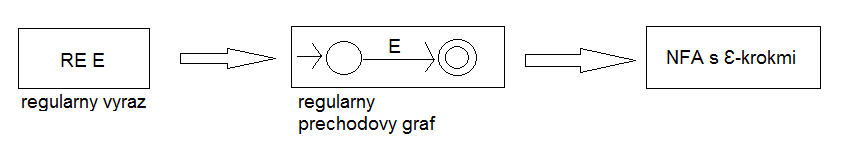
Mnozina regularnych vyrazov nad abecedou ∑, je definovana induktivne:  
 1. Ԑ, {} a x pre kazde x ∊ ∑ su regularne vyrazy nad ∑  
 2. Ak su E, F regularne vyrazy nad ∑, potom aj (E.F), (E+F) a (E\*) su regularne vyrazy nad ∑  
 3. Kazdy regularny vyraz vznikne po konecnom pocte aplikacii krokov 1-2

Priorita operatorov regularnych vyrazov: 1. Iteracia (\*), 2. Zretazenie (.), 3. Zjednotenie (+)

Ak E je regularny vyraz, potom existuje konecny automat rozpoznavajuci L(E).  
Nech L je akceptovany nejakym DFA, potom L je popisatelny regularnym vyrazom.

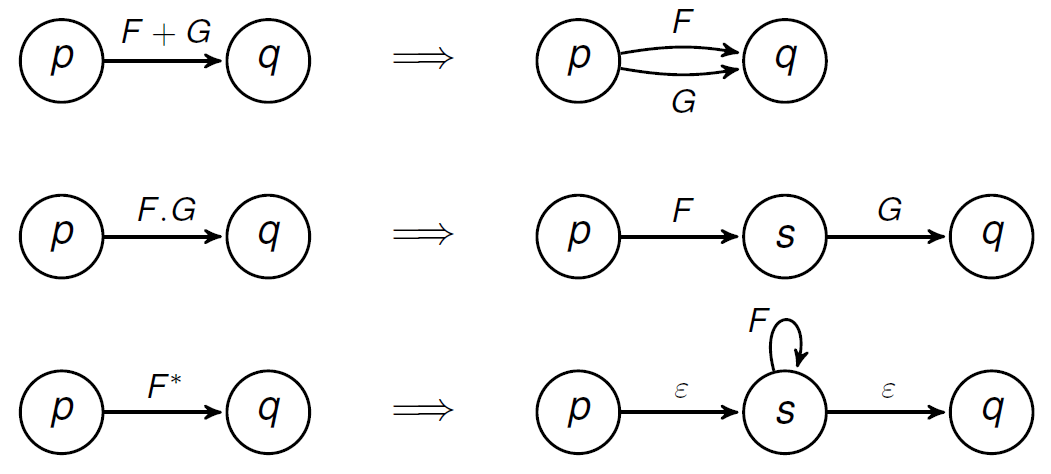
Regularny prechodovy graf je pätica (Q,∑,δ,I,F):  
 -Q = neprazdna konecna mnozina stavov  
 -∑ = vstupna abeceda  
 -δ: Q x Q -> RE(∑) je parcialna prechodova funkcia  
 -I ⊆ Q = mnozina pociatocnych stavov  
 -F ⊆ Q = mnozina koncovych stavov

Pre lubovolny regularny prechodovy graf existuje ekvivalentny NFA s Ԑ-krokmi.

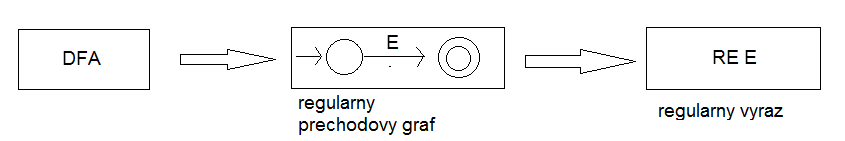


Algoritmus prevodu regularneho prechodoveho grafu na NFA s Ԑ-krokmi:

1. Ku grafu M pridame novy stav q0 a hranu q0 -> q pod Ԑ pre kazde q ∊ I, stav q0 sa stane jedinym  
pociatocnym stavom automatu M’, prvky F ostanu jeho koncovymi stavmi.

2. Opakovane realizuj kroky (a) a (b) pokial prechodovy graf obsahuje aspon jednu hranu ohodnotenu  
symbolom, ktory nepatri do ∑ ∪ {Ԑ}, teda je tvaru F+G, F.G, F\* alebo {}  
 (a) Odstran vsetky hrany, ktore su ohodnotene symbolom {}  
 (b) Vyber lubovolnu hranu p -> q ohodnotenu vyrazom E, kde E ∉ ∑ ∪ {Ԑ}, odstran ju a sprav:  
 

Prevod DFA na regularny prechodovy graf



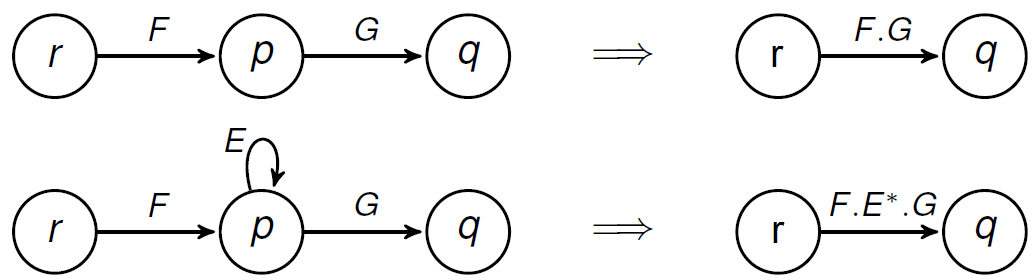
Pre kazdy regularny prechodovy graf M = (Q,∑,δ,I,F) existuje ekvivalentny prechodovy graf   
M’ = ({x,y},∑,δ’,{x},{y}), kde δ’ moze byt definovana len pre dvojicu (x,y).  
=> nemoze byt definovany prechod medzi (x,x), (y,y) ani medzi (y,x)

Algoritmus prevodu DFA na regularny prechodovy graf:

1. Ku grafu M pridame novy pociatocny stav x a novy koncovy stav y. Pridame hrany x -> q pod Ԑ, pre   
kazde q ∊ I a hrany r -> y pod Ԑ pre kazde r ∊ F.

2. Pokial do stavu p nevedie hrana z ineho uzlu, je nedosiahnutelny z pociatocneho stavu x, pokial z p   
nevedie hrana do ineho uzlu, neda sa z neho dosiahnut koncovy stav y, v oboch pripadoch takyto stav  
odstranime bez nahrady.

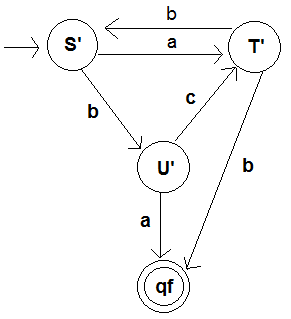
Pre kazdu dvojicu vstupnej hrany veducej do p z ineho uzlu a vystupnej hrany veducej z p do ineho uzlu  
pridame priamy prechod a stav p odstranime.



Prevod regularnej gramatiky na konecny automat

Ku kazdej regularnej gramatike G = (N,∑,P,S) existuje NFA M = (Q,∑,δ,q0,F) taky, ze L(G)=L(M).

Konstrukcia konecneho automatu M = (Q,∑,δ,q0,F) pre gramatiku:  
 -Q = { A’ | A ∊ N } ∪ {qf}, kde qf ∉ N  
 -q0 = S’  
 -δ je najmensia funkcia Q x ∑ -> 2Q splnajuca:  
 -pokial A -> aB je pravidlo v P, tak B’ ∊ δ(A’,a)  
 -pokial A -> a je pravidlo v P, kde a ≠ Ԑ, tak qf ∊ δ(A,a)  
 -F = {S’,qf} pokial S -> Ԑ je pravidlo v P, inak F = {qf}

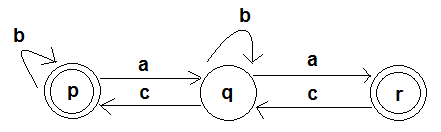
G = ({S,T,U},{a,b,c},P,S)  
P = { S -> aT  
 S -> bU  
 T -> bS  
 U -> cT 🡺  
 U -> a  
 T -> b }

Prevod konecneho automatu na regularnu gramatiku

Pro kazdy konecny automat M=(Q,∑,δ,q0,F) existuje regularna gramatika G = (N,∑,P,S) taka, ze L(M)=L(G).

Konstrukcia gramatiky(predpokladajme ze M je nedeterministicky):

-N = {q’ | q ∊ Q} ∪ {S}, kde S ∉ Q   
 -P je najmensia mnozina pravidiel splnajuca:  
 -pokial p ∊ δ(q,a), potom q’ -> ap’ je pravidlo v P  
 -pokial p ∊ δ(q,a) a p ∊ F, q’ -> a je pravidlo v P  
 -pokial p ∊ δ(q0,a), je S -> ap’ je pravidlo v P   
 -pokial p ∊ δ(q0,a) a p ∊ F, S -> a je pravidlo v P  
 -pokial q0 ∊ F, S -> Ԑ je pravidlo v P



G =({S,P’,Q’,R’},{a,b,c},P,S)  
🡺  
 P = { p’ -> aq’ | bp’ | b,  
 q’ -> cp’ | ar’ | bq’ | a | c,  
 r’ -> cq’   
 S -> Ԑ | aq’ | bp’ | b }

DFA, NFA, NFA s Ԑ-krokmi, regularne gramatiky a regularne vyrazy definuju triedu regularnych jazykov, a  
dokazeme medzi nimi prevadzat.

Rozhodnutelne problemy pre triedu regularnych jazykov

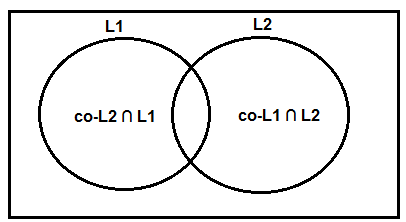
Majme dane konecne automaty M a M’ nad ∑:  
 -ekvivalencia: plati L(M) = L(M’) ?   
 -inkluzia (jazykov): plati L(M) ⊆ L(M’) ?   
 -prislusnost (slova k jazyku): ak mame w ∊ ∑\*, plati w ∊ L(M) ?  
 -prazdnost (jazyka): L(M) = {} ?  
 -univerzalita (jazyka): L(M) = ∑\* ?

-konecnost (jazyka): je L(M) konecny jazyk ?

Problem prazdnosti:

L(M) je prazdny, prave ked medzi dosiahnutelnymi stavmi automatu M nieje ziadny koncovy stav.

Problem univerzality:

L(M) je univerzalny, prave ked jazyk reprezentovany automatom pre doplnok tohto jazyka je prazdny,   
=> L(M) = ∑\* ⬄ co-L(M)={}.  
  
Problem ekvivalencie:   
(L1 ⬄ L2) ⬄ (L1 ∩ co – L2) ∪ (co-L1 ∩ L2) = {}, alebo pomocou minimalizacie a kanonizacie automatov.  


Problem inkluzie:   
L ⊆ L’ ⬄ L’ ∪ L = L’

Problem prislusnosti:   
automat M prevedieme na NFA a mechanicky zbehneme vypocet automatu na danom slove

Problem konecnosti:  
Nech M je DFA, L je nekonecny prave ked M akceptuje aspon jedno slovo w ∊ ∑\* s vlastnostou  
n ≤ |w| < 2n, kde n = card(Q) (pocet stavov automatu)

Bezkontextove jazyky

Bezkontextove jazyky su jazyky generovane bezkontextovou gramatikou.

Bezkontextova gramatika G je stvorica (N,∑,P,S):  
 -N = neprazdna konecna mnozina neterminalov  
 -∑ = konecna mnozina terminalnych symbolov taka, ze N ∩ ∑ = {}  
 -S ∊ N = pociatocny neterminal  
 -P ⊆ N x V\* = konecna mnozina pravidiel (V = N ∪ ∑)

Derivacne stromy pre bezkontextove gramatiky

1. Koren ma navestie S, vnutorne uzly maju navestie z N, listy maju navestie z N ∪ ∑ ∪ {Ԑ}

2. Ak ma vnutorny uzol navestie A a vsetci jeho potomkovia n1,…,nk maju v usporiadani zlava doprava  
navestie X1,…,Xk ∊ V ∪ {Ԑ}, potom A -> X1…Xk ∊ P

3. Kazdy list s navestim Ԑ je jedinym synom svojho otca (a zaroven je splneny samozrejme bod 2.)

Derivacia je sekvencia S => α1 => α2 => … => αn

Lava (resp. prava) derivacia je taka derivacia, kde kazde αi+1 vznikne z αi prepisanim najlavejsieho (resp.   
najpravejsieho) neterminalu.

Kazdemu derivacnemu stromu zodpoveda prave jedna lava derivacia (resp. prava derivacia).  
Kazdej lavej (resp. pravej) derivacii zodpoveda prave jeden derivacny strom.

CFG (bezkontextova gramatika) G sa nazyva **viacznacna(nejednoznacna)** prave ked existuje w ∊ L(G)   
majuce aspon dva rozne derivacne stromy, v opacnom pripade je G jednoznacna.

Bezkontextovy jazyk L sa nazyva **vnutorne(inherentne) viacznacny**, prave ked kazda CFG, ktora ho  
generuje, je viacznacna.

Problemy jednoznacnosti CFG a jednoznacnosti resp. vnutornej viacznacnosti bezkontextoveho jazyka su   
nerozhodnutelne.

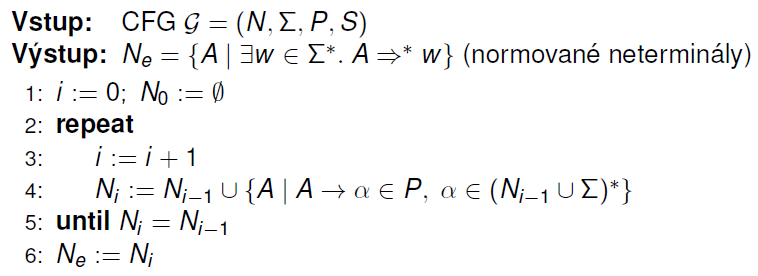
Symbol X ∊ N ∪ ∑ je **nepouzitelny** v CFG G = (N,∑,P,S) prave, ked v G neexistuje derivacia tvaru:  
S =>\*wXy =>\* wxy, pre ziadne w,x,y ∊ ∑\*

Gramatika je **redukovana**, ak neobsahuje ziadne nepouzitelne symboly.

X je **nepouzitelny typu 1 (tj. nenormovany)** ⬄ neexistuje w ∊ ∑\* splnajuce X =>\* w

X je **nepouzitelny typu 2 (tj. nedosiahnutelny)** ⬄ neexistuju α,β ∊ (N∪∑)\* splnajuce S =>\* αXβ

Nenormovane mozu byt len neterminaly, lebo vzdy plati a =>\* a, kde a je terminal  
Nedosiahnutelne mozu byt aj terminaly aj neterminaly.

Najdenie nepouzitelnych symbolov typu 1   


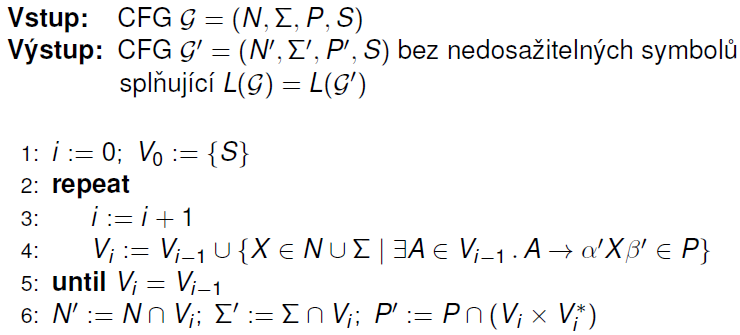
Pr: Mame CFG G = ({S,B,C,D},{a,b,c},P,S)  
 P = { S -> BCa | Da  
 B -> Ba | Ca  
 C -> bCa | c  
 D -> aDc }

Riesenie: N0 = {}  
 N1 = {C}  
 N2 = {C,B}  
 N3 = {C,B,S}

N4 = {C,B,S} => Ni = Ni-1 => Ne = Ni => Ne = N4 => Ne = {C,B,S} => D je nenormovane

Dosledky tohto algoritmu:  
1.Existuje algoritmus, ktory pre lubovolnu CFG G rozhoduje, ci L(G)={}, staci overit, ci S ∉ Ne.  
2. Pre CFG, ktora generuje neprazdny jazyk, tj L(G) ≠ {} existuje ekvivalentna CFG G’ bez nepouzitelnych  
symbolov typu 1.

Najdenie nepouzitelnych symbolov typu 2



Pr: Mame CFG G = ({S,A,B},{a,b,c,d,e},P,S)  
 P = { S -> aSb | c | aB

A -> dA | d  
 B -> eB }

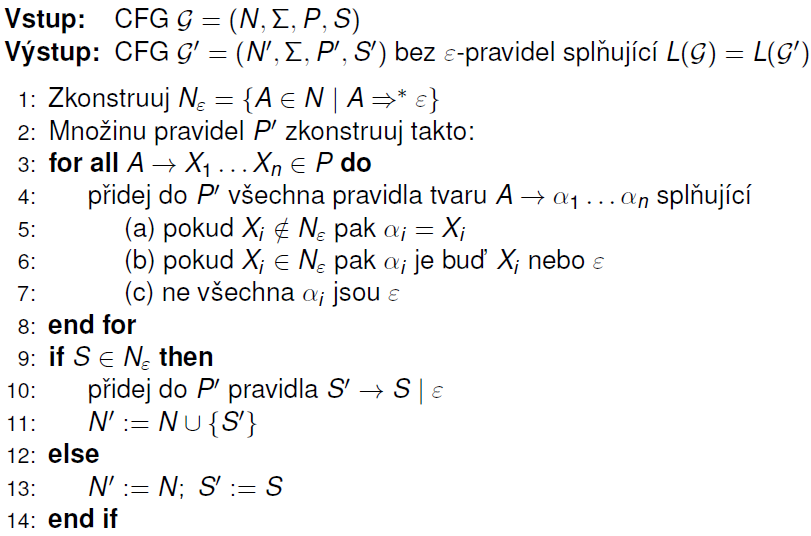
Riesenie: V0 = {S}  
 V1 = {S,a,b,c,B}  
 V2 = {S,a,b,c,B,e}  
 V3 = {S,a,b,c,B,e} => Vi = Vi-1 => N’ = {S,B}, ∑’ = {a,b,c,e}, P’ = { S -> abS | c | aB, B -> eB }

* G’ = ({S,B},{a,b,c,e},{S -> aSb | c | aB, B -> eB},S)

Kazdy neprazdny bezkontextovy jazyk L je generovany nejakou redukovanou CFG.

CFG G = (N,∑,P,S) je bez Ԑ-pravidiel prave, ked:  
1. P neobsahuje ziadne Ԑ-pravidlo (tj. pravidlo tvaru A -> Ԑ) alebo  
2. v P existuje prave jedno Ԑ-pravidlo S -> Ԑ a S sa nevyskytuje na pravej strane ziadneho pravidla z P

Odstranovanie Ԑ-pravidiel v CFG



Pr: Mame CFG G = ({S,A,B},{a,b,c},P,S)  
 P = { S -> aAbBc | AB  
 A -> BB | a | Ԑ  
 B -> AA | b }

Riesenie:

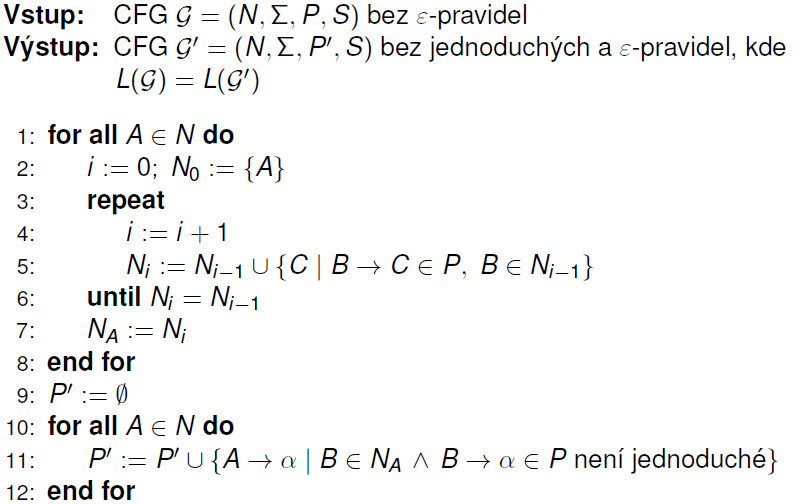
N0 = {A}  
N1 = {A,B}  
N2 = {A,B,S} => NԐ = {A,B,S}

CFG G’ = ({S,A,B,S’},{a,b,c},P’, S’)

P’ = {S -> aAbBc | aAbc | abBc | abc | AB | A | B  
 A -> BB | B | a   
 B -> AA | A | b   
 S’ -> S | Ԑ }

Jednoduche pravidlo je pravidlo tvaru A -> B, kde A,B ∊ N

Odstranovanie jednoduchych pravidiel



Pr: Mame CFG G = ({S,A,B,C},{a,b,c,},P,S)  
 P = { S -> ABC   
 A -> aA | B | a  
 B -> bB | A  
 C -> cC | A }

Riesenie:

NS = {S}  
NA = {A,B}

NB = {B,A}  
NC = {C,A,B}

G’ = ({S,A,B,C},{a,b,c},P’,S)

P’ = { S -> ABC  
 A -> aA | a | bB  
 B -> bB | aA | a  
 C -> cC | aA | a | bB }

CFG sa nazyva necyklicka, prave ked neexistuje A ∊ N take, ze A =>+ A.

CFG sa nazyva vlastna, prave ked je bez nepouzitelnych symbolov, bez Ԑ-pravidiel a je necyklicka.

Ak gramatika CFG G bez Ԑ-pravidiel ma cyklus => v G su jednoduche pravidla

Ku kazdemu neprazdnemu bezkontextovemu jazyku existuje vlastna bezkontextova gramatika.

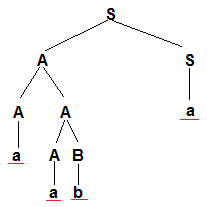
Chomskeho normalna forma (CNF)

Bezkontextova gramatika G = (N,∑,P,S) je v Chomskeho normalnej forme (CNF), prave ked G je bez  
Ԑ-pravidiel a kazde pravidlo z P ma jeden z tychto tvarov:  
 1. A -> BC, kde B,C ∊ N  
 2. A -> a, kde a ∊ ∑  
 3. S -> Ԑ

Pr: Majme bezkontextovu gramatiku G v CNF tvare:  
 G = ({S,A,B},{a,b},P,S)  
 P = { S -> AS | a  
 A -> AB | AA | a  
 B -> b }

Vygenerovanie slova aaba touto gramatikou(lavou derivaciou):  
S -> AS -> AAS -> aAS -> aABS -> aaBS -> aabS -> aaba

Derivacny strom pre odvodenie tohto slova:



Dlzka odvodenia prazdneho slova w = Ԑ je **1**, iba pomocou pravidla S -> Ԑ

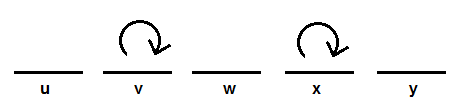
Dlzka odvodenia slova w ≠ Ԑ = |w| - 1 + |w| = **2|w| - 1** |w| - 1: zodpoveda mnozeniu neterminalov ( -1 lebo na zaciatku tam uz jeden mame (S))  
 |w|: prepisanie neterminalov na terminaly

Algoritmus transformacie do CNF  
1. Pre jazyk L = {} zostrojime gramatiku v CNF generujucu prazdny jazyk, napr: G = ({S},∑,{S -> SS}, S)

2. Pre jazyk L ≠ {}  
 Gramatiku pre L prevedieme na vlastnu a bez jednoduchych pravidiel.  
 Pravidla tvaru X -> aB, X-> ab prevedieme na pravidla X -> a’B, X -> a’b’, kde a’,b’ su neterminaly  
 Pravidla tvaru X -> aBcD prevedieme na pravidla:   
 X -> a’<Bc’D>, <Bc’D> -> B<c’D>, <c’D> -> c’D, c’ -> c, a’ -> a

Lemma o vkladani pre bezkontextove jazyky  
Nech L je CFL => existuju p,q ∊ N take, ze kazde slovo z ∊ L dlhsie ako p sa da pisat v tvare z = uvwxy, kde:  
 aspon jedno zo slov v,x je neprazdne (tj vx ≠ Ԑ)  
 |vwx| ≤ q   
 uviwxiy ∊ L pre kazde i ∊ N0

Tvrdenie plati aj ked namiesto konstant p,q budeme vsade pisat len konstantu n.



Nech L je generovany gramatikou v CNF:  
dlzka cesty z korena do listu je:

=pocet modrych hran (hrany medzi neterminalmi)  
 =pocet neterminalov na ceste – 1

Hlbka stromu = maximalna dlzka cesty

Derivacny strom hlbky k ma max 2k listov => slovo dlzky najviac 2k

Derivacny strom pre slovo dlhsie ako 2k-1 ma cestu dlzky aspon k. Tato cesta ma aspon k+1 neterminalov.

Lemma o vkladani je implikacia P => Q ,kde P je vyrok, ze L je CFL a Q su uvedene vlastnosti:  
 -pre lubovolnu konstantu n ∊ N   
 -kazde slovo z ∊ L dlhsie ako n  
 -sa da pisat v tvare z = uvwxy, pricom splna vx ≠ Ԑ a |vwx| ≤ n, uviwxiy ∊ L pre kazde i ∊ N0

Obmena lemmatu o vkladani ¬Q => ¬P sa pouziva k dokazu, ze nejaky jazyk L nieje CFL, ukazeme platnost ¬Q:

¬Q:

-pre lubovolnu konstantu n ∊ N  
-existuje slovo z ∊ L dlhsie ako n take, ze  
-pre vsetky slova u,v,w,x,y splnajuce z = uvwxy, vx ≠ Ԑ a |vwx| ≤ n  
-existuje i ∊ N0 take, ze uviwxiy ∉ L:

Rekurzivne neterminaly a gramatiky

Neterminal A v CFG G = (N,∑,P,S) sa nazyva lavorekurzivny ak v G existuje derivacia A =>+ Aβ

CFG bez lavorekurzivnych neterminalov sa nazyva nelavorekurzivna.  
Ak je v CFG pravidlo tvaru A -> Aα, hovorime o priamej lavej rekurzii na A.

Priama lava rekurzia Pr: P = { A -> Ac } => Priama lava rekurzia: A -> Ac  
Nepriama lava rekurzia Pr: P = { A -> Bc, B -> Ad } => Nepriama lava rekurzia: A -> Bc -> Adc

Algoritmus odstranenia priamej lavej rekurzie

Nech CFG G = (N,∑,P,S) je necyklicka a bez Ԑ-pravidiel, v ktorej vsetky A-pravidla (pravidla majuce na  
lavej strane A) su tvaru:   
A -> Aα1 | … | Aαm | β1 | … | βn, kde kazdy retazec βi zacina symbolom roznym od A

Dostaneme gramatiku G’ = (N ∪ {A’}, ∑,P’,S), kde P’ obdrzime tak, ze vsetky vyssie uvedene pravidla  
nahradime pravidlami:  
A -> β1 | … | βn | β1A’ | … | βnA’  
A’ -> α1 | … | αm | α1A’ | … | αmA’

* Potom L(G) =L(G’) a G’ je necyklicka a bez Ԑ-pravidiel

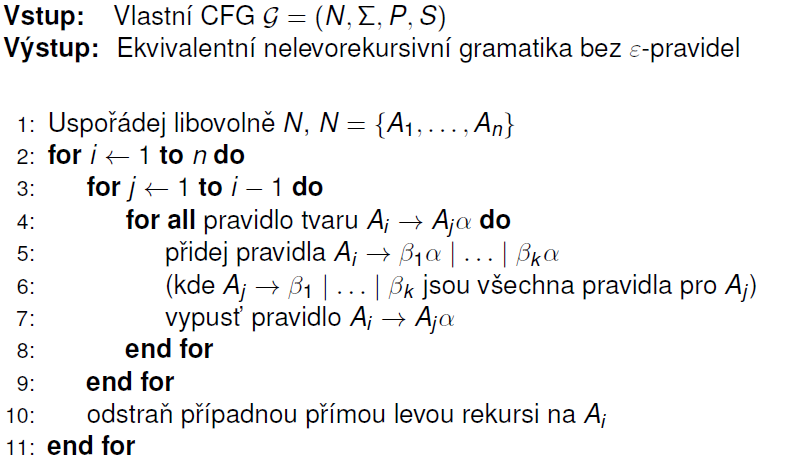
Lemma o substitucii

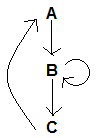
Nech G = (N,∑,P,S) je CFG. Nech A -> α1Bα2 ∊ P  
Nech B -> β1 | … | βr su vsetky pravidla v P tvaru B -> α

Definujeme G’ = (N,∑,P’,S), kde   
 P’ = ( P \ {A -> α1Bα2}) ∪ {A -> α1β1α2 | … | α1βrα2}

* L(G) = L(G’)

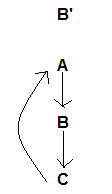
Algoritmus odstranenia lavej rekurzie



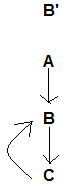


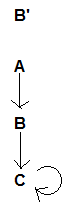
Pr:

P = { A -> Bd | c  
 B -> Bdd | Ccc | aAd 🡺  
 C -> Aa }

Krok 1: Odstranenie priamej lavej rekurzie na B

P = { A -> Bd | c  
 B -> Ccc | aAd | CccB’ | aAdB’ 🡺  
 B’ -> dd | ddB’  
 C -> Aa }

Krok 2: Odstranenie rekurzie C -> A  
P = { A -> Bd | c  
 B -> Ccc | aAd | CccB’ | aAdB’ 🡺  
 B’ -> dd | ddB’  
 C -> Bda | ca }



Krok 3: Odstranenie rekurzie C -> B  
P = { A -> Bd | c  
 B -> Ccc | aAd | CccB’ | aAdB’ 🡺  
 B’ -> dd | ddB’  
 C -> Cccda | aAdda | CccB’da | aAdB’da | ca }



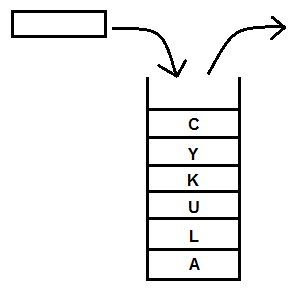
Krok 4: Odstranenie priamej lavej rekurzie na C:  
P = { A -> Bd | c  
 B -> Ccc | aAd | CccB’ | aAdB’  
 B’ -> dd | ddB’ 🡺  
 C -> aAdda | aAdB’da | ca | aAddaC’ | aAdB’daC’ | caC’   
 C’ -> ccda | ccB’da | ccdaC’ | ccB’daC’ }

Greibachova normalna forma

Bezkontextova gramatika G = (N,∑,P,S) je v Greibachovej normalnej forme (GNF) prave, ked:  
 -G je bez Ԑ-pravidiel  
 -kazde pravidlo z P je tvaru A -> aα, kde a ∊ ∑ a α ∊ N\* (s pripadnou vynimkou pravidla S -> Ԑ)

Kazdy bezkontextovy jazyk sa da generovat bezkontextovou gramatikou v GNF.

Zasobnikove automaty



Pre precitanie symbolu zo zasobnika, musime symbol zo  
zasobnika vybrat (potom ho tam mozme znova vlozit).

Stack = zasobnik, v ktorom je dovolene citat symboly kdekolvek  
uprostred zasobnika, bez nutnosti vytiahnutia

Nedeterministicky zasobnikovy automat (PDA) je sedmica M = (Q,∑,Γ,δ,q0,Z0,F), kde:  
 -Q = konecna mnozina stavov  
 -∑ = vstupna abeceda  
 -Γ = zasobnikova abeceda  
 -δ: Q x (∑ ∪ {Ԑ}) x Γ -> PFIN(Q x Γ\*) = parcialna prechodova funkcia  
 -q0 ∊ Q = pociatocny stav  
 -Z0 ∊ Γ = pociatocny symbol v zasobniku  
 -F ⊆ Q = mnozina koncovych stavov

Konfiguracia je lubovolny prvok (p,w,α) ∊ Q x ∑\* x Γ\*, kde:  
 -Q je stav  
 -∑\* je neprecitana cast slova  
 -Γ\* je obsah zasobniku

Na mnozine vsetkych konfiguracii zasobnikoveho automatu M definujeme binarnu relaciu krok vypoctu:



Akceptujuci vypocet zasobnikoveho automatu

Jazyk akceptovany PDA M **koncovym stavom** definujeme ako:



* Slovo je akceptovane, ak sme ho precitali cele a skoncili sme v akceptujucom stave qf, pricom  
  neberieme ohlad na obsah zasobnika po dokonceni vypoctu

Jazyk akceptovany PDA M **prazdnym zasobnikom** definujeme ako:



* Slovo je akceptovane, ak sme ho precitali cele a na konci vypoctu mame prazdny zasobnik,  
  pricom neberieme ohlad na to, ci sme po vypocte skoncili v akceptujucom stave qf
* Rovnaky zasobnikovy automat moze akceptovat dva rozne jazyky, jeden moze akceptovat  
  koncovym stavom a druhy prazdnym zasobnikom.

Trieda jazykov, ktoru akceptuje zasobnikovy automat **koncovym stavom** je ta ista, ako trieda jazykov,   
ktoru akceptuje zasobnikovy automat **prazdnym zasobnikom**.

Trieda jazykov rozpoznavanych zasobnikovymi automatmi je prave trieda bezkontextovych jazykov.

Pr: PDA M = ({q0},{a,b},{Z,A},δ,q0,Z,{})  
 δ(q0,a,Z) = {(q0,A)}  
 δ(q0,a,A) = {(q0,AA)}  
 δ(q0,b,A) = {(q0,Ԑ)}

L(M) = {} – jazyk akceptovany koncovym stavom tohto automatu je prazdny lebo M nema koncovy stav

Le(M) = {w ∊ {a,b}+ | #a(w) = #b(w) a pre kazdy vlastny prefix u slova w plati #a(u) > #b(u)

Vlastny prefix = cast slova, ktora nieje Ԑ ani cele slovo

Pre kazdy jazyk L plati:  
L = L(N) pre nejaky PDA N ⬄ L = Le(M) pre nejaky PDA M

Koncovy stav => prazdny zasobnik

K danemu automatu N, ktory akceptuje koncovym stavom skonstruujeme automat M akceptujuci  
prazdnym zasobnikom, pricom tento automat M bude simulovat cinnost automatu N.

Ak vojde automat N do koncoveho stavu, M sa nedeterministicky rozhodne:  
 -pokracovat v simulacii automatu N alebo  
 -prejde do novo vytvoreneho stavu qԐ, v ktorom vyprazdni zasobnik

Komplikacia:



* Pred zahajenim simulacie bude u M na dne zasobnika novy symbol, ktory nedovolime odstranit   
  nikde inde ako v stave qԐ.

Konstrukcia:

Nech N = (Q,∑,Γ,δ,q0,Z0,F)  
=> M = (Q ∪ {q0’,qԐ}, ∑,Γ ∪ {Z’}, δ’,q0’,Z’,{}), kde Z’ ∉ Γ, q0’,qԐ ∉ Q a δ’ je definovana:

δ’(q0’,Ԑ,Z’) = {(q0,Z0Z’)} // novy stav q0’ sluzi len pre vlozenie Z’ ako noveho dna

ak δ(q,a,Z) obsahuje (r,γ), potom δ’(q,a,Z) obsahuje (r,γ)

δ’(q,Ԑ,Z) obsahuje (qԐ,Z) pre vsetky q ∊ F a Z ∊ Γ ∪ {Z’}

δ’(qԐ,Ԑ,Z) = {(qԐ,Ԑ)} pre vsetky Z ∊ Γ ∪ {Z’}

Prazdny zasbonik => koncovy stav

K danemu automatu M, ktory akceptuje prazdnym zasobnikom skonstruujeme automat N akceptujuci  
koncovym stavom, pricom tento automat N bude simulovat cinnost automatu M.  
 -N si pred simulaciou prida na dno zasobniku novy symbol  
 -Ak je N schopny precitat tento symbol (tj. zasobnik automatu M je prazdny), tak N prejde do  
 novo vytvoreneho stavu qf, ktory je koncovym stavom.

Konstrukcia:

Nech M = (Q,∑,Γ,δ,q0,Z0,{})  
=> N = (Q ∪ {q0’,qf},∑,Γ ∪ {Z’}, δ’,q0’,Z’,{qf}), kde Z’ ∉ Γ, q0’,qf ∉ Q a δ’ je definovana:

δ’(q0’,Ԑ,Z’) = {(q0,Z0Z’)}

ak δ(q,a,Z) obsahuje (r,γ), potom δ’(q,a,Z) obsahuje (r,γ)

δ’(q,Ԑ,Z’) = {(qf,Ԑ)} pre vsetky q ∊ Q

Rozsireny zasobnikovy automat

-sedmica R = (Q,∑,Γ,δ,q0,Z0,F), kde  
 -vsetky symboly az na δ maju rovnaky vyznam ako PDA  
 -δ je zobrazenie z **konecnej podmnoziny mnoziny** Q x (∑ ∪ {Ԑ}) x Γ\* do **konecnych podmnozin  
 mnoziny** Q x Γ\*

Konfiguracie, akceptovany jazyk koncovym stavom alebo prazdnym zasobnikom zostavaju ako pri PDA.  
Krok vypoctu je definovany:



* Rozsireny zasobnikovy automat dovoluje manipulovat s viacerymi symbolmi zasobniku naraz  
  v jednom kroku

Rozsireny PDA => PDA

Ku kazdemu rozsirenemu PDA existuje ekvivalentny (obycajny) PDA.

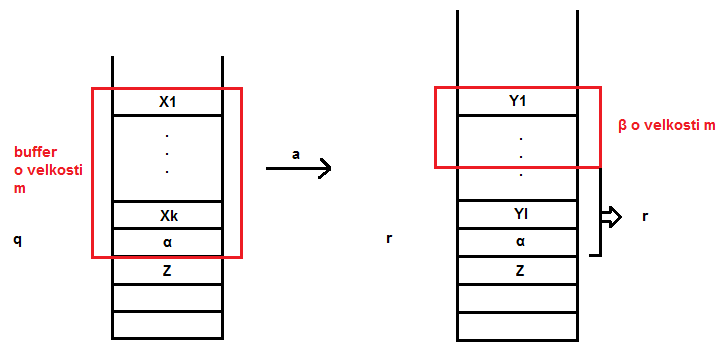
Nech R = (Q,∑,Γ,δ,q0,Z0,F) je rozsireny PDA a m = max { |α| δ(q,a,α) je definovane pre nejake  
q ∊ Q, a ∊ ∑ ∪ {Ԑ} }

Definujeme P = (Q1,∑,Γ1,δ1,q1,Z1,F1), kde:

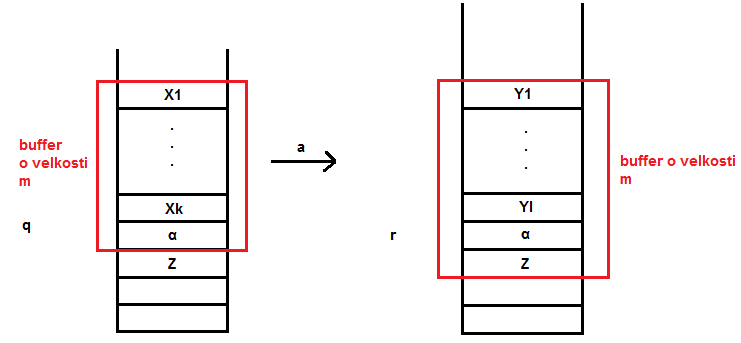
-Q1 = {[q,α] | q ∊ Q, α ∊ Γ1\*, 0 ≤ |α| ≤ m}  
 -Γ1 = Γ ∪ {Z1}, kde Z1 je novy symbol  
 -q1 = [q0,Z0Z1m-1]  
 -F1 = {[q,α] | q ∊ F, α ∊ Γ1\*, 0 ≤ |α| ≤ m}

δ1 je definovana takto:  
-ak δ(q,a,X1…Xk) obsahuje (r,Y1…Yl), potom

1. l ≥ k: δ1([q,X1…Xkα],a,Z) obsahuje ([r,β],γZ), kde βγ = Y1 … Ylα a |β|=m



2. l < k: δ1([q,X1…Xkα],a,Z) obsahuje ([r,Y1 … YlαZ],Ԑ) pre vsetky Z ∊ Γ1 a α ∊ Γ1\* take,  
 ze |α| = m – k



-δ1([q,α],Ԑ,Z) = {([q,αZ],Ԑ)} pre vsetky q ∊ Q, Z ∊ Γ1 a α ∊ Γ1\* take, ze |a| < m

Ku kazdemu PDA M sa da zostrojit CFG G taka, ze Le(M) = L(G)

Ku kazdej CFG G sa da zostrojit PDA M taky, ze L(G) = Le(M)

CFG => PDA

w ∊ L(G) ⬄ v G existuje derivacny strom s vysledkom w

1. Syntakticka analyza zhora nadol  
2. Syntakticka analyza zdola nahor

1. Nedeterministicka syntakticka analyza zhora nadol

Budovanie derivacneho stromu simuluje lavu derivaciu, tj. vzdy rozvijame najlavejsi neterminal.

V lavej derivacii je v jednom kroku odvodenia nahradeny najlavejsi neterminal A pravou stranou X1…Xn   
nejakeho A-pravidla.

V automate M skonstruovanemu k danej gramatike jednemu kroku lavej derivacie zodpoveda nahrada   
A na vrcholu zasobniku retazcom X1…Xn

M = ({q},∑,N ∪ ∑,δ,q,S,{}), kde δ je definovana:  
-δ(q,Ԑ,A) obsahuje (q,α) prave ked A -> α ∊ P // Ak na vrchole zasobniku je neterminal A a v mnozine

P pravidiel gramatiky existuje pravidlo A -> α

na vrchol zasobniku zapis α miesto A.

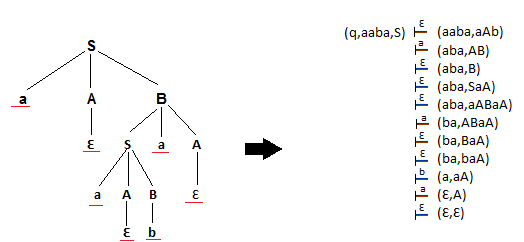
-δ(q,a,a) = {(q,Ԑ)} pre vsetky a ∊ ∑ // Ak prvy symbol neprecitanej casti slova je terminal a a

terminal a je aj na vrchole zasobniku, zmaz ho odtial.

Pr: G = ({S,A,B},{a,b},P,S) 🡺 M = ({q},∑,N ∪ ∑, δ,q,S,{})   
 P = { S -> aAB 🡺 δ(q,Ԑ,S) = {(q,aAB)}  
 A -> Aa | Ԑ 🡺 δ(q,Ԑ,A) = {(q,Aa),(q,Ԑ)}  
 B -> SaA | b } 🡺 δ(q,Ԑ,B) = {(q,SaA),(q,b)}

δ(q,a,a) = δ(q,b,b) = {(q,Ԑ)}

Vypocet syntaktickou analyzou zhora nadol pre slovo aaba:

S => aAB

=> aB  
 => aSaA  
 => aaABaA 🡺  
 => aaBaA  
 => aabaA  
 => aaba

2. Nedeterministicka syntakticka analyza zdola nahor

Nech G je lubovolna CFG, potom sa da skonstruovat rozsireny PDA R taky, ze L(G) = L(R).

**Vrchol zasobniku piseme vpravo.**  
Konstruujeme rozsireny PDA R, ktory simuluje pravu derivaciu v G v obratenom poradi.

PDA R ma kroky dvoch typov:

1. Moze kedykolvek nacitat do zasobniku symbol zo vstupu
2. (redukcia) ak je na vrcholu zasobniku retazec tvoriaci pravu stranu nejakeho pravidla v G,  
   moze ho nahradit odpovedajucim lavostrannym neterminalom (a zo vstupu nic necita)

Nech G = (N,∑,P,S)  
=> R = ({q,r},∑,N ∪ ∑ ∪{⊥}, δ, q,⊥,{r}), kde δ je definovana:

1. δ(q,a,Ԑ) = {(q,a)} pre vsetky a ∊ ∑
2. ak je A -> α pravidlo v P, potom δ(q,Ԑ,α) obsahuje (q,A)
3. δ(q,Ԑ,⊥S) = {(r,Ԑ)}

Vlastnosti bezkontextovych jazykov

Trieda bezkontextovych jazykov je uzavreta na:  
 -zjednotenie

-zretazenie

-iteracia

-pozitivna iteracia

-prienik s regularnym jazykom

Trieda bezkontextovych jazykov nieje uzavreta na:

-prienik

-doplnok

Rozhodnutelne problemy pre bezkontextovy jazyky:

-Problem prislusnosti: w ∊ L(G) ?  
 -Problem prazdnosti: L(G) = {} ?  
 -Problem konecnosti: L(G) je konecny ?

Konecnost

Ku kazdej CFG G sa daju urcit cisla m,n take, ze L(G) je nekonecny prave ked existuje slovo z ∊ L(G)  
take, ze m < |z| ≤ n.

Vlastnost sebevlozenia

Nech G = (N,∑,P,S) je CFG => G ma vlastnost sebevlozenia, ak existuju A ∊ N a u,v ∊ ∑+ take, ze   
A =>+ uAv.

CFL L ma vlastnost sebevlozenia, ak kazda bezkontextova gramatika, ktora ho generuje, ma vlastnost  
sebevlozenia.

CFL L ma vlastnost sebevlozenia, prave ked L nieje regularny.

Nerozhodnutelne problemy pre bezkontextove jazyky:

**Problem regularity**  
Neexistuje algoritmus, ktory pre lubovolnu CFG G rozhoduje, ci je L(G) regularny ci nie.  
(Teda nieje rozhodnutelne, ci L(G) ma vlastnost sebevlozenia ci nie).

**Problem universality**  
Neexistuje algoritmus, ktory pre lubovolnu CFG G rozhoduje, ci L(G) = ∑\* ci nie.

**Problemy ekvivalencie a inkluzie**  
niesu rozhodnutelne, plynie to z nerozhodnutelnosti problemu universality

Deterministicke zasobnikove automaty

PDA M = (Q,∑,Γ,δ,q0,Z0,F) je **deterministicky (DPDA)**, ak su splnene tieto podmienky:

1. pre vsetky q ∊ Q a Z ∊ Γ plati:  
   kedykolvek δ(q,Ԑ,Z) ≠ {}, potom δ(q,a,Z) = {} pre vsetky a ∊ ∑
2. pre ziadne q ∊ Q, Z ∊ Γ a a ∊ ∑ ∪ {Ԑ} neobsahuje δ(q,a,Z) viac ako jeden prvok

Jazyk L je deterministicky bezkontextovy jazyk (DCFL), prave ked existuje DPDA M taky, ze L=L(M).

Vlastnosti deterministickych bezkontextovych jazykov (DCFL):  
Trieda DCFL je uzavreta na:  
 -doplnok

-prienik s regularnym jazykom

Trieda DCFL nieje uzavreta na:

-prienik

-zjednotenie

Trieda DCFL tvori vlastnu podtriedu triedy bezkontextovych jazykov => existuju bezkontextove jazyky,  
ktore niesu DCFL.

Pr: Jazyk co – {ww | w ∊ {a,b}\*} je CFL, ale nieje DCFL

Turingov stroj

Turingov stroj (Turing Machine, TM) je devatica M = (Q,∑,Γ,⊳,⊔,δ,q0,qacc,qrej), kde:

-**Q** = konecna mnozina stavov

-**∑** = vstupna abeceda

-**Γ** = pracovna abeceda, ∑ ⊆ Γ

-**⊳** ∊ Γ \ ∑ = lava koncova znacka

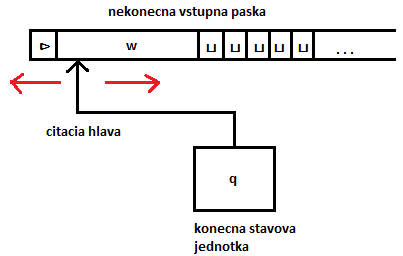
-**⊔** ∊ Γ \ ∑ = symbol oznacujuci prazdne policko

-**δ**: (Q \ {qacc,qrej}) x Γ -> Q x Γ x {L,R} = totalna prechodova funkcia

-**q0** ∊ Q = pociatocny stav

-**q**a**cc** ∊ Q = akceptujuci stav

-**qrej**∊ Q = zamietajuci stav



Pre kazde q ∊ Q musi v TM existovat p ∊ Q taky, ze δ(q, ⊳) = (p, ⊳,R) => tj. znak ⊳ sa neda prepisat, a  
ani sa neda posunut citacia hlava za okraj pasky

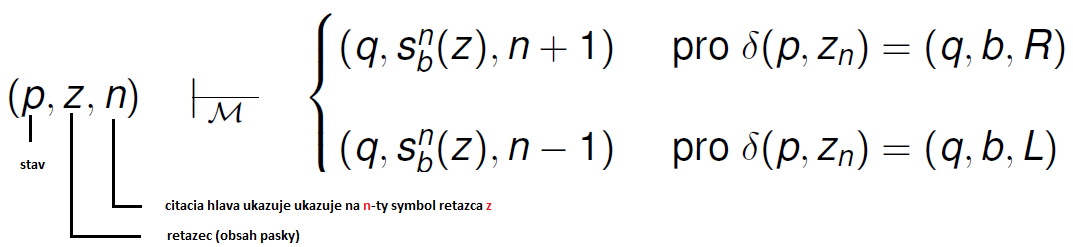
⊔ω = nekonecna postupnost prazdnych policok

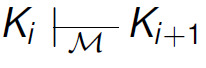
Konfiguracia turingovho stroja je trojica (q,z,n) ∊ Q x {y⊔ ω | y ∊ Γ\*} x N0, kde:  
 -q je stav  
 - y⊔ ω je obsah pasky (konecne slovo + nekonecny retazec prazdnych policok)  
 -n znaci poziciu hlavy na paske, znaci sa prirodzenym cislom (cislo 0 = pociatocny znak)

**Pociatocna konfiguracia** pre vstup w ∊ ∑\* je trojica (q0, ⊳w⊔ ω,0)  
**Akceptujuca konfiguracia** je kazda trojica tvaru (qacc,z,n)  
**Zamietajuca konfiguracia** je kazda trojica tvaru (qrej,z,n)

Pre lubovolny nekonecny retazec z nad Γ, zn oznacuje n-ty symbol retazca z (z0 je najlavejsi symbol  
retazca z).

 oznacuje retazec vzniknuty z retazca z nahradenim symbol zn symbolom b.

Na mnozine vsetkych konfiguracii stroja M definujeme binarnu relaciu **krok vypoctu** takto:  


**Vypocet** TM M na vstupe w je maximalna (konecna alebo nekonecna) postupnost konfiguracii K0,K1,K2,…, kde K0 je pociatocna konfiguracia pre w a pre vsetky i ≥ 0

Stroj M **akceptuje** slovo w prave ked vypocet M na w je konecny a jeho posledna konfiguracia je  
akceptujuca.  
Stroj M **zamieta** slovo w prave ked vypocet M na w je konecny a jeho posledna konfiguracia je  
zamietajuca.  
Stroj M pre vstup w **cykli** prave ked vypocet M na w je nekonecny.

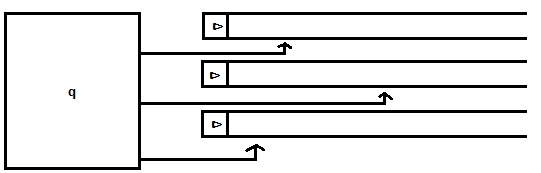
**Jazyk akceptovany TM M** je definovany: {w ∊ ∑\* | M akceptuje w}

Viacpaskovy Turingov stroj

k-paskovy Turingov stroj je definovany rovnako ako TM s vynimkou prechodovej funkcie δ, ktora je   
definovana ako totalna funkcia:  
 δ: (Q \ {qacc,qrej}) x Γk -> Q x Γk x {L,R}k

Konfiguracie maju tvar (q,z1,…,zk,n1,…,nk) ∊ Q x (Γ\*.{⊔ ω}k x N0k

Pociatocna konfiguracia pre w ∊ ∑\* je (q0,⊳w⊔ ω, ⊳⊔ ω,…, ⊳⊔ ω,0,…,0)  
Definicia akceptujucej/zamietajucej konfuguracie a kroku vypoctu sa zmeni podobne ale minimalne.



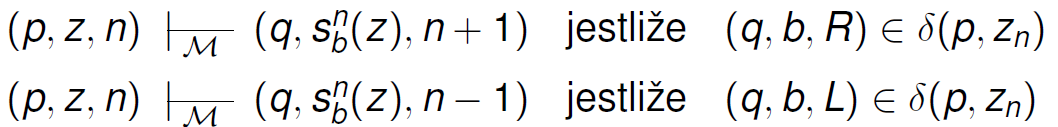
Ekvivalencia viacpaskoveho a jednopaskoveho TM  
Pre kazdy viacpaskovy turingov stroj existuje ekvivalentny jednopaskovy TM

1. Neprazdny obsah k pasok a polohy hlav vlozime za seba na jednu pasku
2. Simulacia jedneho kroku = zistit inormacie pod hlavami, zapisat nove a posunut hlavy
3. Ak je potreba dalsie policko nejakej pasky, posunieme aktualny obsah

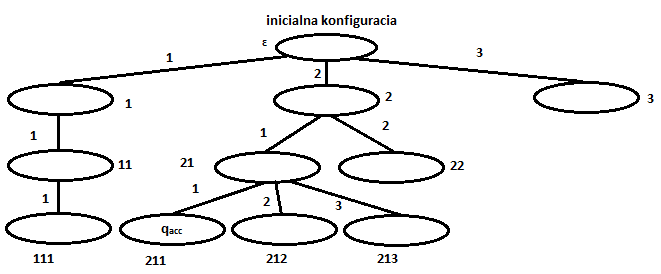
Nedeterministicky Turingov stroj

Je definovany rovnako ako TM s vynmikou prechodovej funkcie δ, ktora je definovana ako totalna:  
 δ: (Q \ {qacc,qrej}) x Γ -> 2Q x Γ x {L,R}

Vsetky pojmy sa definuju rovnako ako u deterministickeho TM, drobne zmeny su len u definicie  
kroku vypoctu a akceptacie slova:



M akceptuje slovo w, prave ked existuje vypocet z pociatocnej konfiguracie pre w do nejakej  
akceptujucej konfiguracie.



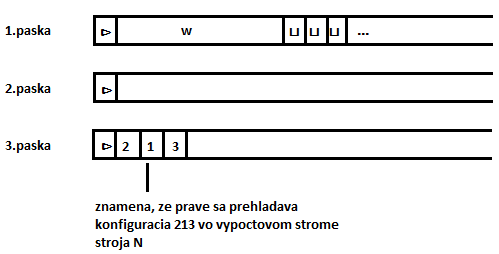
Vypocet Turingovho stroja sa da reprezentovat takymto stromom.

Ekvivalencia nedeterministickeho a deterministickeho TM  
Pre kazdy nedeterministicky turingov stroj N, existuje ekvivalentny deterministicky turingov stroj TM.

Prevod nedeterministickeho turingovho stroja N na deterministicky TM:  
Zostrojime 3-paskovy deterministicky TM preskumavajuci vypoctovy strom stroja N. Tento 3-paskovy  
stroj sa da previest na jednopaskovy deterministicky TM.

Nech k = max {| δ(q,z)||q ∊ Q \ {qacc,qrej}, z ∊ Γ}

1. Paska obsahuje vzdy len vstup, nemeni sa  
2. Paska sluzi k simulacii nedeterministickeho stroja  
3. Paska obsahuje retazec {1,…,k}\* urcujuci, ktory uzol stromu je prave prehladavany.



Hlada sa akceptujuca konfiguracia vo vypoctovom strome prehladavanim do sirky.  
Kontrola jedneho uzlu vypoctoveho stromu:

1. Skopiruj prvu pasku na druhu.
2. Na 2. Paske simuluj N, nedeterministicke volby ries podla cisel na 3. paske. Ak narazis  
   na akceptujuci stav, akceptuj. V ostatnych pripadoch (prislusna volba neexistuje alebo N  
   dojde do zamietajuceho stavu alebo dosli cisla na 3.paske) pokracuj dalsim krokom.
3. Nahrad obsah retazca na 3.paske jeho naslednikom v lexikografickom usporiadani a zacni   
   znovu.

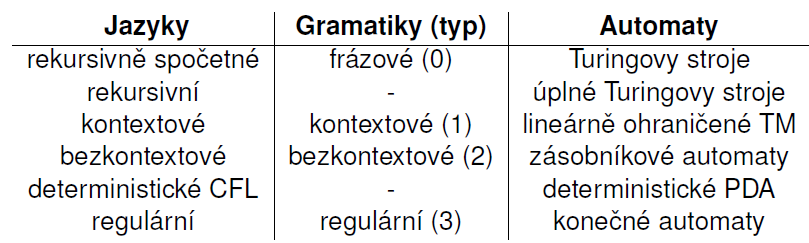
Churchova teza: Kazdy proces, ktory sa da intuitivne nazvat algoritmom, sa da realizovat na TM.

Turingove stroje a triedy jazykov

Jazyk L je **rekurzivne spocitatelny** (tj. generovany gramatikou typu 0) ⬄ L je akceptovany nejakym  
Turingovym strojom.

Turingov stroj so vstupnou abecedou ∑ sa nazyva **uplny**, ak jeho kazdy vypocet je konecny. Jazyk sa   
nazyva **rekurzivny**, ak je akceptovany nejakym uplnym Turingovym strojom.

(Obecny) TM M **akceptuje**/**rozpoznava**/**prijima** jazyk L(M).  
Uplny TM M **rozhoduje** jazyk L(M).



**Trieda na nizsom riadku je vzdy vlastnou podtriedou triedy na vyssom riadku.**

Uzaverove vlastnosti reukurzivnych a rekurzivne spocitatelnych jazykov

Rekurzivne a rekurzivne spocitatelne jazyky su uzavrete na:  
 -zjednotenie  
 -prienik  
 -zretazenie  
 -iteraciu

Rekurzivne jazyky su uzavrete este na:  
 -doplnok  
Trieda rekurzivne spocitatelnych jazykov **nieje** uzavreta na:  
 -doplnok

**Jazyk L je rekurzivny, prave ked jazyky L a co-L su rekurzivne spocitatelne  
== Problem P je rozhodnutelny ⬄ problem P aj jeho doplnok su ciastocne rozhodnutelne**

Kodovanie TM

Kazdy TM sa da zakodovat do binarneho retazca, predpokladame ze M:  
 -ma stavy Q = {q1,q2,q3,…,qn}  
 -q1 je inicialny stav, q2 je akceptujuci stav, q3 je zamietajuci stav  
 -ma paskovu abecedu Γ = {X1,X2,…,Xz}  
 -X1 = ⊳, X2 = ⊔

Prechod δ(qi,Xj) = (qk,Xl,L) kodujeme retazcom = 0i10j10k10l10  
Pr: δ(q3,X4) = (q2,X3,L) = 00010000100100010

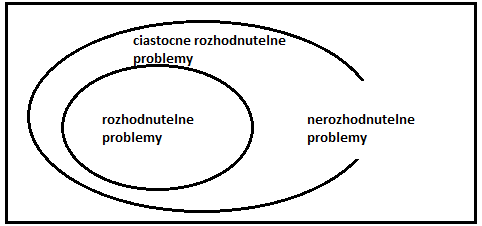
Prechod δ(qi,Xj) = (qk,Xl,R) kodujeme retazcom 0i10j10k10l100  
Pr: δ(q3,X4) = (q2,X3,R) = 000100001001000100

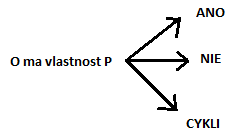
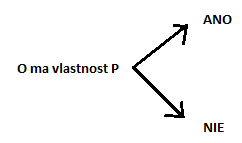
Z kodov jednotlivych prechodov (v lubovolnom poradi) zostavime kod M:  
<M> = 111 kod1 11 kod2 11 … 11 kodr 111  
Retazce nekodujuce ziadny TM povazujeme za kod TM akceptujuceho {}  
Slova kodujeme podobne: <M,w> = <M>#<w>

Univerzalny turingov stroj  
Existuje univerzalny turingov stroj U, ktory dokaze simulovat lubovolny zadany TM na zadanom vstupe:  
U akceptuje <M>#<w> ⬄ M akceptuje w

Rozhodnutelnost problemov

Problem P odpovedajuci jazyku L = {<O> | O ma vlastnost P} je   
 -**rozhodnutelny**, prave ked L je rekurzivny  
 -**nerozhodnutelny**, prave ked L nieje rekurzivny  
 -**ciastocne rozhodnutelny (semirozhodnutelny)**, prave ked L je rekurzivne spocitatelny

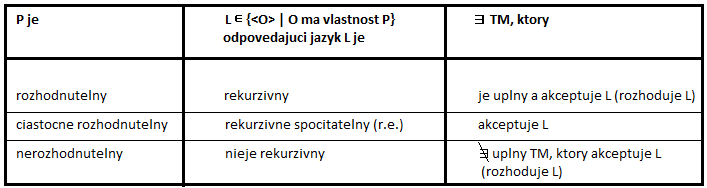


**Rozhodnutelny problem: Ciastocne rozhodnutelny problem:**  


Ciastocne rozhodnutelny problem napr: problem, pre ktory existuje turingov stroj, ktory ho akceptuje,  
ale neexistuje uplny turingov stroj, ktory ho rozhoduje.

Problem akceptovania (problem prislustnosti pre Turingove stroje):  
je problem rozhodnut, ci dany TM M akceptuje dane slovo w nad jeho vstupnou abecedou.  
Problem stotoznime s jazykom ACC = {<M,w> | M je TM a M akceptuje w}

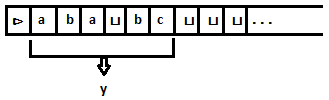
Problem akceptovania je ciastocne rozhodnutelny.



Problem zastavenia (halting problem):  
je problem rozhodnut, ci dany TM M ma na danom slove w nad jeho vstupnou abecedou konecny   
vypocet (teda ci M na vstupe w zastavi).  
Problem stotoznime s jazykom HALT = {<M,w> | M je TM a vypocet M na w je konecny}

Problem zastavenia je ciastocne rozhodnutelny.

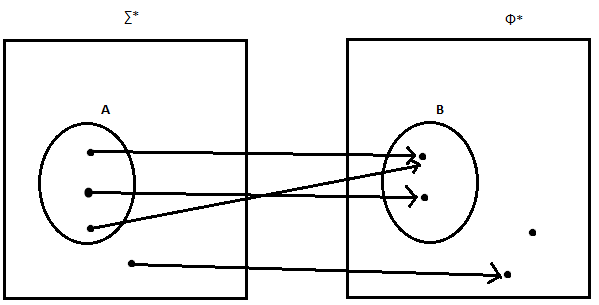
Jednopaskovy deterministicky Turingov stroj sa da chapat ako funkcia, ktorej hodnotou pre dany vstup  
je obsah pasky po skonceni vypoctu. Tj. pokial stroj M na vstupe w zastavi s obsahom pasky ⊳y⊔ω  
kde y nekonci na ⊔, potom y oznacime ako M(w).

 Ak M na w cykli => M(w) = nedefinovana

Vycislitelne funkcie

Funkcia f: ∑\* -> Φ\* je **vycislitelna**, pokial existuje TM M, ktory na vstupe w zastavi, prave ked f(w) je  
definovana a naviac f(w) = M(w).  
Funkcia je **totalne vycislitelna**, ak je vycislitelna a totalna.

Redukcia  
Nech A ⊆ ∑\* a B ⊆ Φ\* su jazyky. Povieme, ze A sa m-redukuje na B, piseme A ≤mB, prave ked existuje  
totalne vycislitelna funkcia f: ∑\* -> Φ\* taka, ze: w ∊ A ⬄ f(w) ∊ B  
Funkciu f nazveme redukciou A na B.



A a B su **m-ekvivalentne**, pisane A ≡mB, pokial A ≤ mB a B ≤ mA.

Pr: HALT ≡ mACC

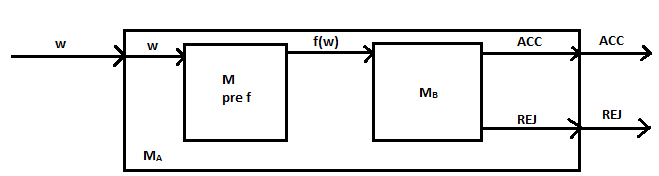
HALT ≤ mACC  
f(x) = <M’w> pokial x = <M,w> a M’ vznikne z M tak, ze prechody do qrej presmerujeme do qacc   
 = x pokial x nieje tvaru <M,w>  
🡺 f je totalna vycislitelna funkcia  
🡺 <M,w> ∊ HALT ⬄ <M’,w> ∊ ACC == x ∊ HALT ⬄ f(x) ∊ ACC

ACC ≤ mHALT  
f(x) = x pokial x nieje tvaru <M,w>  
 = <M’,w> pokial x = <M,w> a M’ vznikne z M tak, ze prechody do qrej presmerujeme do noveho stavu,  
 ktory cykli  
🡺 f je totalna vycislitelna funkcia  
🡺 <M,w> ∊ ACC ⬄ <M’,w> ∊ HALT == x ∊ ACC ⬄ f(x) ∊ HALT

Redukcia a rozhodnutelnost

Nech A ≤ mB  
 -B je rekurzivny => A je rekurzivny  
 -B je rekurzivne spocitatelny => A je rekurzivne spocitatelny

Nech f je redukcia A na B a MB je TM akceptujuci B.  
Stroj MA akceptujuci A na vstupe w:  
1. Spocita f(w)  
2. Spusti MB na vstupe f(w) a vrati rovnaky vysledok ako MB



Nech A ≤ mB  
 -A nieje rekurzivny => B nieje rekurzivny  
 -A nieje rekurzivne spocitatelny => B nieje rekurzivne spocitatelny

Nech A ≡ mB  
 -A je rekurzivny ⬄ B je rekurzivny  
 -A je rekurzivne spocitatelny ⬄ B je rekurzivne spocitatelny

Dokaz (ciastocnej) rozhodnutelnosti A:  
musime najst B, ktore je (ciastocne) rozhodnutelne a najst redukciu A ≤ mB

Dokaz nerozhodnutelnosti B:  
musime najst A, ktore je nerozhodnutelne a najst redukciu A ≤ mB

Problem neprazdnosti (nonempty problem):  
je problem rozhodnutia, ci dany TM akceptuje neprazdny jazyk  
Problem stotoznime s jazykom NONEMPTY = {<M> | M je TM a L(M) ≠ {}}

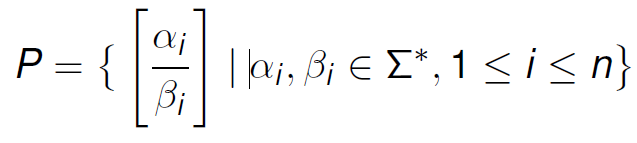
Problem neprazdnosti nieje rozhodnutelny.

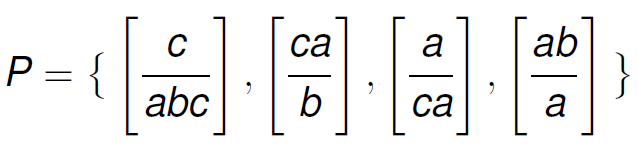
Pr: ACC ≤ mNONEMPTY

f(x) = 0 pokial x nieje tvaru <M,w>  
 <M’> pokial x = <M,w> a M’ vznikne z M tak, ze na vstupe roznom od w cykli a na w sa chova ako M  
🡺 f je totalna vycislitelna funkcia  
🡺<M,w> ∊ ACC ⬄ <M’> ∊ NONEMPTY == x ∊ ACC ⬄ f(x) ∊ NONEMPTY

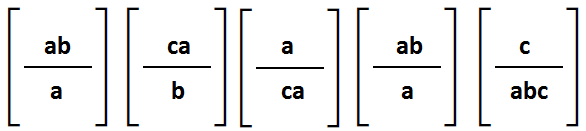
Postov system

Postov system P nad abecedou ∑ je konecna mnozina dvojic

  
Riesenim systemu P je konecna neprazdna postupnost prirodzenych cisel i1,i2,…,ik taka, ze 1 ≤ij ≤ n   
a αi1αi2…αik = βi1βi2…βik

Pr:

Risenie:



Pri ocislovani kociek v zadani zlava doprava 1,2,3,4 je riesenim sekvencia cisel: 42341, dalsie riesenie je  
napr: 4234142341

Postov korespondencny problem (PCP):  
je problem rozhodnut, ci ma dany Postov system P nejake riesenie  
Problem stotoznime s jazykom PCP = {<P> | P je postov system, ktory ma nejake riesenie}

PCP nieje rozhodnutelny.

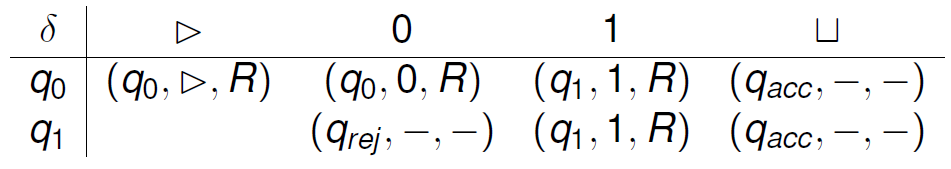
Inicialny Postov korespondencny problem (inPCP):  
je problem rozhodnut, ci ma Postov system P nejake riesenie zacinajuce cislom 1  
Problem stotoznime s jazykom inPCP = {<P> | P je Postov system, ktory ma riesenie zacinajuce cislom 1}

Casova zlozitost algoritmu

-pocet “krokov” vypoctu  
-zavisi na vstupe a vypoctovom modeli  
-ako zakladny model pouzijeme **Turingov stroj**  
-skumame **najhorsi pripad**, teda maximalny pocet krokov v zavislosti na **dlzke vstupu**  
-da sa skumat aj priemerny pripad

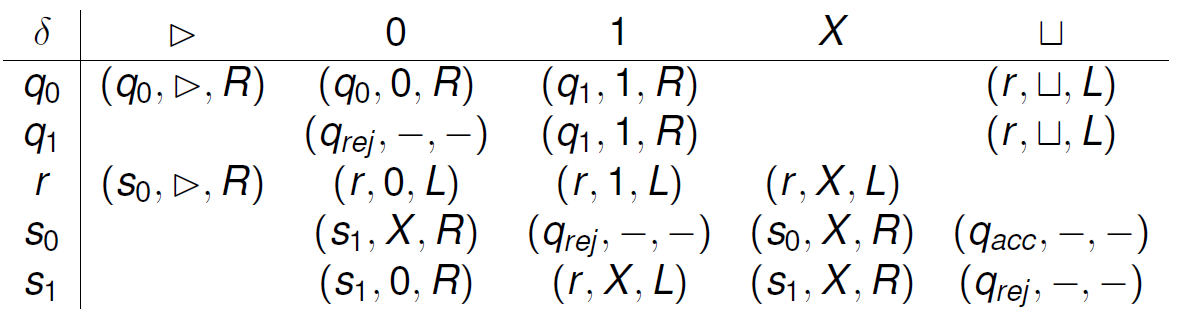
Casova zlozitost deterministickeho Turingovho stroja:  
Nech M je uplny deterministicky (jednopaskovy alebo viacpaskovy) Turingov stroj so vstupnou abecedou  
∑. Pre kazde w ∊ ∑\* definujeme tM(w) ako pocet krokov vypoctu stroja M na vstupe w. Casova zlozitost  
stroja M je potom funkcia TM: N0 -> N definovana vztahom: TM(n) = max {tM(w) | w ∊ ∑n}.  
🡺 Hovorime, ze M pracuje v case TM(n).

Pr: M = ({q0,q1,qacc,qrej}, {0,1}, {0,1, ⊳,⊔}, ⊳, ⊔,δ,q0,qacc,qrej)



* L = {0}\*.{1}\* a TM(n) = n+2

Pr2: M = ({q0,q1,r,s0,s1,qacc,qrej}, {0,1}, {0,1, ⊳,⊔}, ⊳, ⊔,δ,q0,qacc,qrej)



* L = {0n1n | n ≥ 0} a TM(n) = (3n2/4)+(7n/2) + 4 pre parne n

Pre kazdy deterministicky uplny TM M a pre kazde m > 1 sa da skonstruovat deterministicky uplny  
TM M’ tak, ze L(M) = L(M’) a

TM’(n) ≤ (TM(n) / m) + n + 1

n+1 uz nezrychlime, lebo to je minimalny pocet krokov pre precitanie vstupu o dlzke n

O - notacia

Nech f,g: N0 -> R+ su funkcie. Povieme, ze g je asymptoticka horna zavora pre f, a piseme to  
f ∊ O(g) alebo f = O(g), ak existuju konstanty c,n0 ∊ N take, ze: [**∀**](http://en.wikipedia.org/wiki/Turned_a)**n ≥ n0: f(n) ≤ cg(n)**.

Deterministicke casove zlozitostne triedy problemov  
casova zlozitost problemu = najmensia casova zlozitost, s akou sa da dany problem rozhodnut

Kazda f: N -> N definuje (deterministicku) casovu zlozitostnu triedu problemov

TIME(f(n))={L | L je rozhodovany nejakym deterministickym TM M s casovou zlozitostou TM(n) = O(f(n))}

Pr: L = {0k1k | k ≥ 0}

-existuje jednopaskovy deterministicky, ktory ho rozhoduje v case O(nlog n)

-neexistuje jednopaskovy deterministicky TM, ktory ho rozhoduje s mensou zlozitostou

-existuje dvojpaskovy deterministicky TM, ktory ho rozhoduje v case O(n) => teda L je v triede

TIME(n)

Vplyv vypoctoveho modelu  
-na rozdiel od vycislitelnosti, v zlozitosti na vypoctovom modeli zalezi  
-rozdiel je aj medzi jednopaskovym a dvojpaskovym deterministickym TM

Casova zlozitost nedeterministickeho Turingovho stroja:

Nech M je uplny nedeterministicky Turingov stroj so vstupnou abecedou ∑. Pre kazde w ∊ ∑\* definujeme  
tM(w) ako pocet krokov najdlhsieho vypoctu stroja M na vstupe w. Casova zlozitost stroja M je potom  
funkcia TM: N0 -> N definovana: TM(n) = max {tM(w) | w ∊ ∑n}

Nedeterministicke casove zlozitostne triedy problemov

Kazda funkcia f: N -> N definuje (nedeterministicku) casovu zlozitostnu triedu problemov

NTIME(f(n)) = {L | L je rozhodovany nejakym nedeterministickym TM M s casovou zlozitostou  
TM(n) = O(f(n))}.

Z definicii plynie => TIME(f(n)) ⊆ NTIME(f(n))

Najvyznamnejsie casove zlozitostne triedy

**Deterministicke**



-trieda vsetkych problemov, ktore sa daju riesit v polynomialnom case deterministickym TM



-trieda vsetkych problemov, ktore sa daju riesit v exponencialnom case deterministickym TM

**Nedeterministicke**

****

-trieda vsetkych problemov, ktore sa daju riesit v polynomialnom case nedeterministickym TM



-trieda vsetkych problemov, ktore sa daju riesit v exponencialnom case nedeterministickym TM

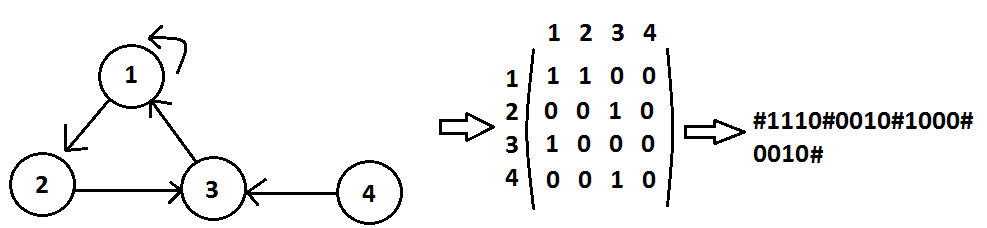
* P ⊆ NP ⊆ EXPTIME ⊆ NEXPTIME
* Bezne deterministicke sekvencne modely vypoctu sa daju medzi sebou prevadzat s   
  polynomialnym narastom casovej zlozitosti => definicia P a EXPTIME niesu citlive na model
* EXPTIME je obvykle zlozitost algoritmov riesiacich problem hrubou silou
* Cook-Karpova teza: P obsahuje prave vsetky prakticky riesitelne problemy

Prislusnost problemu v P

-staci ukazat, ze problem je riesitelny v polynomialnom pocte krokov a ze kazdy krok je  
implementovatelny (vykonatelny) v polynomialnom case.

-kodovanie/dekodovanie objetkov O do slov <O> musi byt preveditelne v polynomialnom case

Priklad vhodneho kodovania: reprezentacia grafu maticou susednosti:



Priklad nevhodneho kodovania: reprezentacia sekvencie cislic unarnym zapisom cisla:  
<13> = 1111111111111

Vhodne zakodovanie <13> = 13 alebo 1101 …

Problem existencie cesty

-je problem rozhodnut, ci v danom orientovanom grafe G existuje cesta z s do t.  
-problem stotoznime ho s jazykom PATH = {<G,s,t> | G je orientovany graf obsahujuci cestu z s do t}

-PATH ∊ P

Postupne spocitame uzly dosiahnutelne z s:  
1. Oznac stav s

2. pokial sa da oznacit novy stav opakuj:  
 -prejdi vsetky hrany v G a oznac kazdy uzol, do ktoreho vedie hrana z oznaceneho uzlu

3. ak je t oznacene, akceptuj, inak zamietni

Celkom O(n) krokov (n je pocet stavov G), kazdy sa da previest v polynomialnom case.

Problem Hamiltonovskej cesty

-problem rozhodnut, ci v danom orientovanom grafe G existuje Hamiltonovska cesta z s do t  
-problem stotoznime s jazykom:   
 HAMPATH = {<G,s,t> | G je orientovany graf obsahujuci Hamiltonovsku cestu z s do t}

-**Hamiltonovska cesta** = cesta prechadzajuca kazdym uzlom grafu prave raz

-HAMPATH ∊ NP

Hamiltonovska cesta v grafe G s n uzlami ma dlzku n-1

Hamiltonovsku cestu budeme nedeterministicky hadat:

1. zacni budovat cestu zo stavu s

2. (n-1) krat opakuj: nedeterministicky vyber hranu veducu z posledneho uzlu cesty a pridaj ju na koniec  
 cesty

3. ak je t posledny uzol cesty a ziadny uzol sa neopakuje, akceptuj, inak zamietni

Kazdy vypocet ma O(n) polynomialnych krokov

Hamiltonovska cesta existuje ⬄ existuje akceptacny vypocet

Problem zlozenych cisel

-problem rozhodnut, ci je dane cislo x zlozene, teda sucinom dvoch cisel vacsich ako 1

-problem stotoznime s jazykom COMPOSITES = {<x> | x = pq pre nejake prirodzene cisla p,q > 1}

-COMPOSITES ∊ NP

Riesenie problemu sa da deterministickym TM v polynomialnom case:  
 -**najst**, ak je problem v P

-**overit**, ak je problem v NP (pokial nam niekto riesenie poskytne)

Polynomialny verifikator

Polynomialny verifikator pre jazyk L je deterministicky TM V splnajuci  
 w ∊ L ⬄ existuje retazec c taky, ze V akceptuje <w,c>  
 a pracuje v polynomialnom case vzhladom k |w|

L ∊ NP ⬄ existuje polynomialny verifikator pre L

Polynomialna redukcia

A sa polynomialne redukuje na B, piseme A ≤ pB, prave ked A ≤ mB a redukcna funkcia f je vycislitelna  
(deterministickym) Turingovym strojom pracujucim v polynomialnom case. Funkciu f nazveme redukciou  
A na B v polynomialnom case.

≤p = ≤m + vypocet f v polynomialnom case

Polynomialna redukcia a zlozitostne triedy

Nech A ≤ pB  
 -B ∊ P => A ∊ P

-B ∊ NP => A ∊ NP

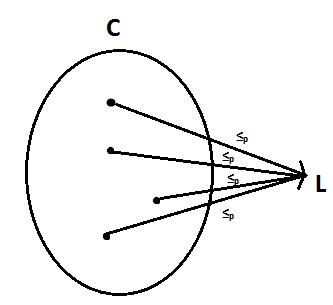
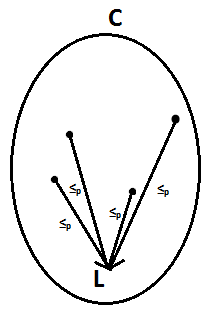
Nech f je redukcia A na B v polynomialnom case a MB je TM akceptujuci B. Stroj MA rozhodujuci A na   
vstupe w:  
1. Spocita f(w)

2. spusti MB na vstupe f(w) a vrati rovnaky vysledok ako MB

Ak je MB deterministicky, potom aj MA je deterministicky. Krok 1 sa da previest v polynomialnom case   
vzhladom k |f(w)|, co je polynomialne aj vzhladom k |w|. MA teda pracuje v polynomialnom case.

Nech C je zlozitostna trieda. Jazyk L nazveme tazky v triede C (C-tazky), prave ked pre kazdy jazyk  
L’ ∊ C plati L’ ≤ pL.

Povieme, ze L je uplny v triede C (C-uply), pokial naviac L ∊ C.



Problem splnitelnosti (SAT)

-je problem rozhodnut, ci dana booleovska formula (formula poskladana z vyrokovych premennych  
 s vyuzitim operacii ∧, ∨ a ¬) splnitelna.

-tento problem stotoznime s jazykom SAT = {<ϕ> | ϕ je splnitelna Booleovska formula}

-SAT je NP-uplny problem

Konjunktivna normalna forma (cnf) formuli:

Literal = premenna alebo jej negacia

Klauzula = disjunkcia literalov

Formula v cnf = konjunkcia literalov

Formula v 3cnf = formula v cnf, kde kazda klauzula ma prave 3 literaly

Problem 3SAT

-problem rozhodnut, ci je dana booleovska formula v 3cnf splnitelna

-tento problem stotoznime s jazykom 3SAT = {<ϕ> | ϕ je splnitelna formula v 3cnf}

-3SAT je NP-uplny

Redukcia a C-tazkost

Nech C je zlozitostna trieda. A je C-tazky a A ≤ pB => B je C-tazky

Ak je A NP-uplny, A ≤ pB a B ∊ NP, potom B je tiez NP-uplny.

Priestorova zlozitost algoritmu

-pamat pouzita pri vypocte

-zavisi na vstupe

-ako zakladny model pouzijeme Turingov stroj

-skumame najhorsi pripad, teda maximalny pocet precitanych policok pasky v zavislosti na dlzke vstupu

-da sa skumat aj priemerny pripad

Priestorova zlozitost deterministickeho Turingovho stroja:

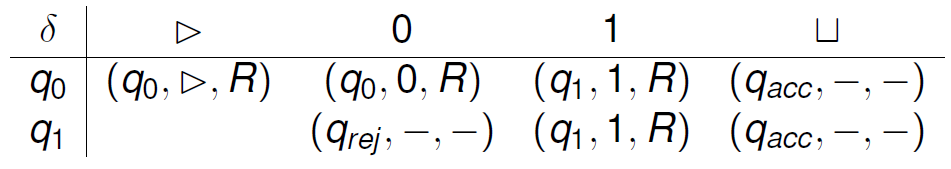
Nech M je uplny deterministicky (jednopaskovy alebo viacpaskovy) Turingov stroj so vstupnou abecedou  
∑. Pre kazde w ∊ ∑\* definujeme sM(w) ako pocet policok pasky, ktore stroj M cita pri vypocte na vstupe  
w. Priestorova zlozitost stroja M je potom funkcia SM: N0 -> N definovana:   
 SM(n) = max {sM(w) | w ∊ ∑n}

Priestorova zlozitost nedeterministickeho Turingovho stroja:

Nech M je uplny nedeterministicky (jednopaskovy alebo viacpaskovy) Turingov stroj so vstupnou  
abecedou ∑. Pre kazde w ∊ ∑\* definujeme sM(w) ako **maximalny** pocet policok pasky, ktore stroj M  
cita pri vypocte na vstupe w. Priestorova zlozitost stroja M je potom funkcia SM: N0 -> N definovana:

SM(n) = max {sM(w) | w ∊ ∑n}

Pr: M = ({q0,q1,qacc,qrej}, {0,1}, {0,1, ⊳,⊔}, ⊳, ⊔,δ,q0,qacc,qrej)

 🡺 SM(n) = n+2

Pre kazdy deterministicky uplny TM M a pre kazde m > 1 sa da skonstruovat deterministicky uplny  
TM M’ tak, ze L(M) = L(M’) a SM’(n)=(SM(n)/m) + n + 2.

**priestorova zlozitost problemu** = najmensia priestorova zlozitost, s akou sa da dany problem rozhodnut

Priestorove zlozitostne triedy problemov

Kazda funkcia f: N -> N definuje priestorove zlozitostne triedy problemov.

SPACE(f(n)) = {L | L je rozhodovany nejakym deterministickym TM M s priestorovou zlozitostou

SM(n) = O(f(n))}

NSPACE(f(n)) = {L | L je rozhodovany nejakym nedeterministickym TM N s priestorovou zlozitostou  
 SN(n) = O(f(n))}

SAT ∊ SPACE(n)

SAT moze byt rozhodovany deterministickym 3-paskovym TM M:  
 -na 2.pasku postupne zapisujeme vsetky ohodnotenia premennych

-na 3.paske pre dane ohodnotenie overime, ci splna vstupnu formula

-akceptujeme, pokial narazime na splnujuce ohodnotenie

-zamietneme, pokial ziadne ohodnotenie nieje splnujuce

Cas a priestor

Pre kazdu funkciu f: N -> N plati:

1. TIME(f(n)) ⊆ SPACE(f(n))

2. NTIME(f(n)) ⊆ NSPACE(f(n))

3. SPACE(f(n)) ⊆ TIME(kf(n)) pre vhodne k ∊ N

4. NSPACE(f(n)) ⊆ NTIME(kf(n)) pre vhodne k ∊ N

Savitchova veta

Pre kazdu funkciu f: N -> N splnujuca f(n) ≥ n plati: NSPACE(f(n)) ⊆ SPACE(f2(n))

Nech N je nedeterministicky TM s priestorovou zlozitostou f(n). Stroj upravime tak, aby pred  
akceptovanim zmazal pasku a posunul hlavu uplne vlavo. Ma teda len jednu akceptujucu konfiguraciu  
cacc. Vypocet stroja ma maximalne kf(n) krokov.

Ekvivalentny deterministicky stroj M implementuje procedure comp(c1,c2,t), ktora akceptuje, pokial  
sa da v stroji N behom najviac t krokov prejst z konfiguracie c1 do c2, inak zamieta. Ak je cw inicialna  
konfiguracia stroja N pre w, staci spustit comp(cw,cacc,kf(n)).

Polynomialne priestorove zlozitostne triedy

PSPACE = ∪k ∊ N SPACE(nk)

NSPACE = ∪k ∊ N NSPACE(nk)

PSPACE = NSPACE (plynie z definicie a zo savitchovej vety)

P ⊆ NP ⊆ NPSPACE ⊆ PSPACE ⊆ EXPTIME ⊆ NEXPTIME