Kompilátory
Lexikálna analýza

Ján Šturc
zima 2009

O čom je lexikálna analýza

- Názov
  - Lexikálny analyzátor, Lexer, Scanner
- Úloha
  - načítanie vstupu a transformácia na vstup pre syntaktickú analýzu (postupnosť tokenov)
- Presnejšie
  - vytvorenie postupnosti tokenov
  - zápis informácie o tokenoch do tabuľky symbolov
  - odstránenie (kompresia) bieleho priestoru (␣, Tab, LF, CR)
  - poskytnúť lokalizačnú informáciu (riadok, pozícia) pre chybové hlásenia
- Popis
  - regulárny jazyk
- Realizácia
  - konečný automat

Prečo oddelujeme lexikálnu analýzu

- Zjednodušenie hlavnej časti kompilátora – syntaktickej analýzy
- Zvýšenie portability (nezávislosť zvyšku kompilátora na kódovaní vstupu)
- Zvýšenie efektívnosti špecializáciou vstupnej regulárnej časti
- Sústredenie „špinavej časti“ manipulácia so vstupnými zariadeniami do jedinej časti kompilátora
  - Dnes možno od toho abstrahovať. Vstup je stream a všetku manipuláciu robí OS alebo „runtime support“ programovacieho jazyka.

Základné pojmy

- Token (gramatická trieda)
  - Reprezentuje množinu reťazcov s rovnakým syntaktickým významom zodpovedajucích jednemu patternu.
  - Je to výstup lexikálnej analýzy a vstup do syntaktickej analýzy. Pre syntaktickú analýzu je to terminál.
  - Tokeny sú: rezervované slová, identifikátory, konštanty, operátory a oddelovače
- Pattern (vzor, šablóna)
  - Regulárny výraz popisujúci nejaký token
- Lexéma (inštancia tokenu)
  - Reťazec znakov vo vstupnom texte, ktorý zodpovedá vzoru (patternu) pre najaký token
- Lexikálna analýza
  - Číta vstup a transformuje ho na postupnosť tokenov.
  - Zapisuje lexémy do tabuľky symbolov.
  - Poskytuje parseru dvojice <token,smerník do tabuľky symbolov>.

Príklad

Vstup: dráha:=pociatok + cas ∗ 60

LEXÉMA	TOKEN		PATTERN
dráha	**id**	(identifikátor)	(letter)(digit|letter)*
:=	**assign**	(symbol priradenia)	:=
počiatok	**id**	(identifikátor)	(letter)(digit|letter)*
+	**op_plus**	(operátor sčítania)	+
čas	**id**	(identifikátor)	(letter)(digit|letter)*
*	**op_mul**	(operátor násobenia)	*
60	**num**	(číselná konštanta)	(digit)+

Výstup: <id,1> assign <id, 2> op_plus <id, 3> op_mul <num, 4>

Identifikátory a konštanty sú zaznamené v tabuľke symbolov.

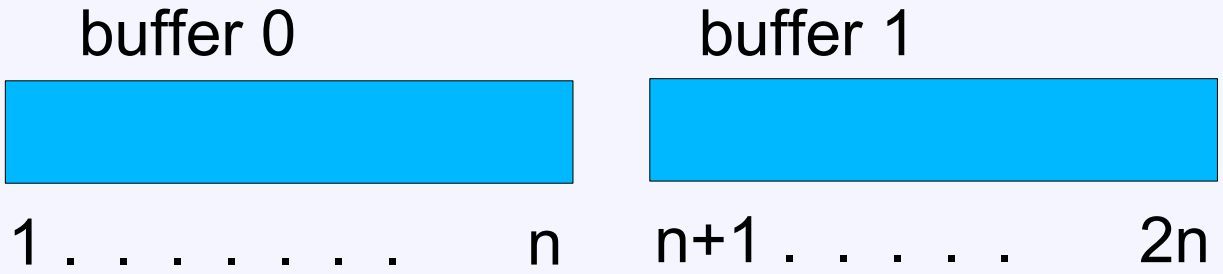
Diagnostika a ošetrenie chýb

- Chyba nastáva ak postupnosť znakov na vstupe nezodpovedá žiadnemu vzoru
- Možnosti zotavenia
  - vynechanie nevhodných znakov
  - vloženie znaku alebo znakov
  - zámena znaku iným znakom
  - transpozícia susedných znakov
- Lexika programovacích jazykov má malú redundanciu.
  - Chyba nastane iba v prípade, že vstup obsahuje znak, ktorý sa nevyskytuje v lexike jazyka.
- Syntaktická analýza zhora dolu môže napovedať, aké tokeny očakáva.

Vstupné rozhranie

- Používa pomalé zariadenia, alebo zariadenia, z ktorých sa číta po blokoch (disk). Tieto zariadenia často môžu pracovať súčasne s procesorom (DMA).
- Často na rozpoznanie lexémy je potrebné prečítať niekoľko znakov dopredu.
- Administrácia pomocou vyrovnávajúcej pamäti – buffra.
- Cyklus spracovania buffra
  - Naplní sa celý buffer naraz.
  - Smerník ukazuje na práve spracovaný symbol.
  - Smerník sa môže posúvať oboma smermi: čítanie a „vrátenie“ znaku na vstup.
  - Po spracovaní celého buffra sa načítavá znovu pokiaľ nie je koniec súboru (EOF, $).
- Urychlenie – paralelizmus dva buffre. Jeden čítame, druhý spracovávame

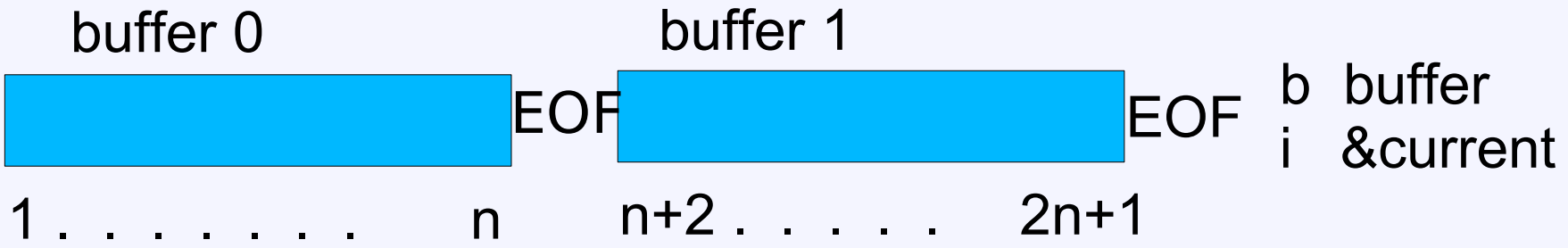
Dvojbuffrová schéma



b buffer
i &current

```
    b:= 0; read(b); i:=1;
L:      b:= b + 1 mod 2;  /* XOR 1 */
        parbegin read(b);
                    while i mod n ≠ 0 do
                          { if *i = EOF then Exit L
                                else { spracuj(*i); i++;}
                          }
        parend;
        if i > n then i:= 1;
        goto L;
```

Dvojbuffrová schéma so zarážkou

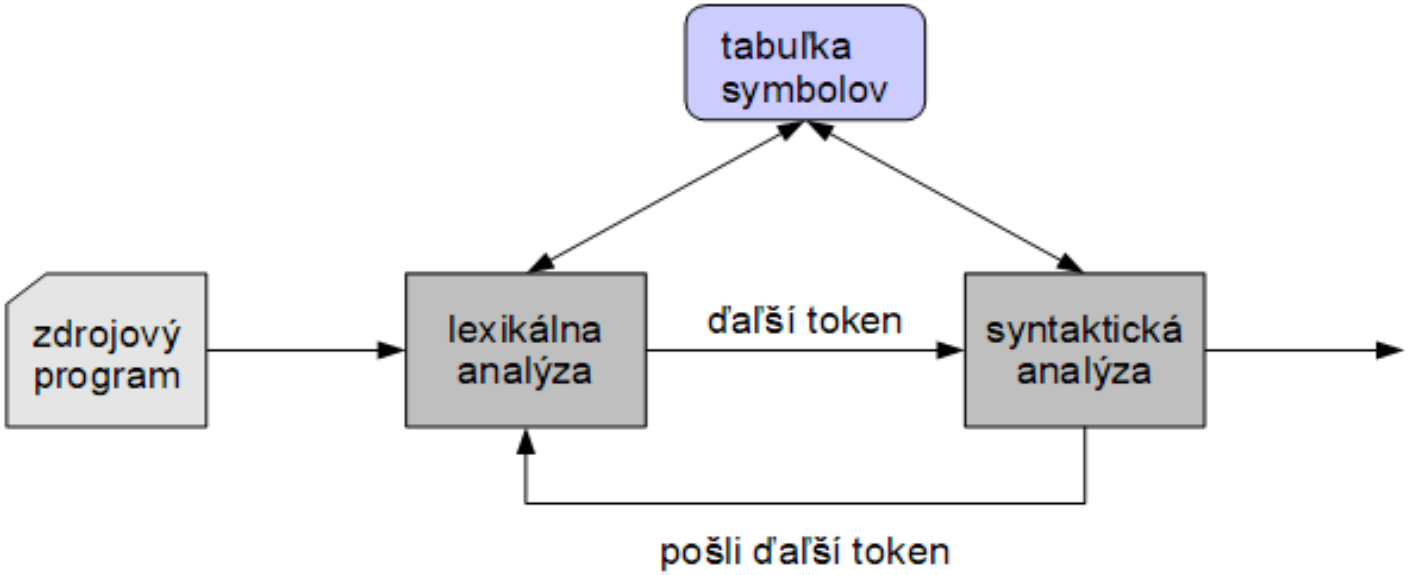



```
    b:= 0; read(b); i:=1;
L:  if i > 2n then i:= 1;
    b:= b + 1 mod 2;  /* XOR 1 */
    parbegin read(b);
                 while *i ≠ EOF do
                       { spracuj(*i); i++; }
    parend;
    if i mod n+1 ≠ 0 then goto L;
```

Problém, že lexéma presahuje hranicu buffra, riešime tak, že procedúra spracuj používa privátnu reťazcovú premennú, kde si uchováva už spracovaný prefix lexémy.

Výstup a ostatná komunikácia

- Lexikálna analýza
  - Obvykle posiela syntaktickej analýze token a jeho a smerník na jeho item v tabuľke symbolov vždy, keď si to parser vyžiadá.
  - Vynimočne môže byť samostatný prechod
  - Komunikuje s tabuľkou symbolov
    - zapisuje pri prvom výskyte lexémy
    - hľadá a vracia smerník opakovaných výskytoch



Popis vzorov – regulárne výrazy

1. Pre každé $r \in T \cup \{\varepsilon\}$, r je regulárny výraz označujúci jazyk $\mathcal{L}(r)$.
2. Ak r a s sú regulárne výrazy označujúce jazyky $\mathcal{L}(r)$ a $\mathcal{L}(s)$. Potom $r \mid s$ je regulárny výraz označujúci označujúci jazyk $\mathcal{L}(r) \cup \mathcal{L}(s)$.
3. Ak r a sú regulárne výrazy označujúce jazyky $\mathcal{L}(r)$ a $\mathcal{L}(s)$. Potom rs je regulárny výraz označujúci označujúci jazyk $\mathcal{L}(r)\mathcal{L}(s)$.
4. Ak r je regulárny výraz označujúci jazyk $\mathcal{L}(r)$. Potom r* je regulárny výraz označujúci označujúci jazyk $\mathcal{L}(r)^*$.

Pozn.: $\varepsilon\varepsilon = \varepsilon$. $(r) = r$. Zátvorky.

Skratky: $r^+ = rr^*$. $r? = r \mid \varepsilon$. $[0 – 9] = 0|1|2| \ldots |9$.

Regulárne definície

- Regulárna definícia je pomenovaný regulárny výraz. Zapisujeme **meno** → r.
- Príklady:

letter → [A – Z] | [a – z]
digit → [0 – 9]
blank → N | Tab | LF | CR
whitespace → **blank**$^+$
number → **digit**$^+$
identifier → **letter(letter|digit)***
begin → b e g i n
relop → < | <= | = | >= | <>

- Iteračné výrazy môžeme písať aj ako ľavolineárne (pravolineárne) pravidlá.

identifier → letter | identifier letter | identifier digit
identifier → letter id
id → letter id | digit id | ε

Rozpoznávanie regulárných definíci

- Thomsonová metóda
  - konštrukcia nedeterministického konečného automatu indukciou (presnejšie dekompozíciou) vzhľadom na štruktúru výrazu
  - transformácia NFA na DFA (štandardný postup).
  - minimalizácia DFA
  - používajú generátory scannerov
- Priama konštrukcia stavov a prechodovej tabuľky deterministického konečného automatu (DFA).
- Metódy hľadania vzorky (pattern matching)
  - Dömölki
  - Aho-Corasick
- Naprogramovanie v programovacom jazyku
  - (napr. C) is_letter, is_digit
- Prepísať na pravidlá a zahrnúť do syntaktickej analýzy

Úskalia lexikálnej analýzy

- Odsadenie ako syntaktická konštrukcia
  - Odsadenie na vstupe niečo znamená (Python, Flex)
- Biely priestor neznamená nič (napr. Fortran)
  - DO 5 I = 1.25 (identifier DO5I assign constant 1.25)
  - DO 5 I = 1,25 (reserved word DO label 5 identifier I assign constant 1 delimiter , costant 25)
- Kľúčové slová môžu byť použité aj ako identifikátory PL/1
  - IF THEN THEN THEN = ELSE; ELSE ELSE=THEN;
- Kontextovo závislé tokeny (napr. PL/1)
  - DECLARE(ARG1, ARG2, ..., ARGn)
  - volanie procedury, deklarácia, názov n-rozmerného poľa
- Konfliktné operátory a oddelovače (napr. C++)
  - template: Foo < Bar >, stream: cin >> var
  - konflikt pri vnorených templates: Foo < Bar < Baz >>

Technická časť - implementácia

Konečné automaty

Konečný automat $A = \langle Q, T, \delta, q_0, F \rangle$, kde

- Q je konečná množina stavov
- T je konečná abeceda (terminálne symboly)
- δ je prechodová funkcia
- q_0 je počiatočný stav
- $F \subset Q$ je množina akceptujúcich stavov

Deterministický konečný automat (DFA): $\delta: Q \times T \rightarrow Q$

Nedeterministický konečný automat (NFA): $\delta: Q \times (T \cup \{\varepsilon\}) \rightarrow Q$

(Častejšie sa používa alternatívna definícia NFA: $\delta: Q \times T \rightarrow 2^Q$.
Dú: Dokážte, že obe definície NFA sú ekvivalentné.)

DFA aj NFA akceptujú regulárne jazyky.

Konštrukcia ekvivalentného DFA k NFA

Hlavný program:

Vstup: NKA $N = (K, \Sigma, \delta, q_0, F)$.
Výstup: DKA D akceptujúci ten istý jazyk, definovaný množinou stavov *Dstates* a prechodovou tabuľkou *Dtran*.

na začiatku ε-*closure*(q_0) je jediný stav v *Dstates* a je neoznačný;
while je v *Dstates* nejaký neoznačený stav q **do begin**
    označ q;
    for každý vstupný symbol a **do begin**
        $U := \varepsilon$-*closure*(*move*(q, a));
        if U nie je v *Dstates* **then**
            pridaj U do *Dstates* ako neoznačený stav;
        Dtran[q, a] := U;
    end;
end;

Konštrukcia ekvivalentného DFA k NFA

Podprogram pre ε-uzáver

- výpočet ε-*closure*(*T*):

```
vlož všetky stavy z T do zásobníka; inicializuj ε-closure(T) na T;
while zásobník nie je prázdny do begin
    vyber q, vrchný symbol zo zásobníka;
    for každý stav s do ktorého sa dá dostať z q na ε do begin
        if s nie je v ε-closure(T) then
            pridaj s do ε-closure(T);
        vlož s na vrch zásobníka;
    end;
end;
```

Pozn. Stav je akceptujúci (patrí do F), ak obsahuje aspoň jeden akceptujúci stav nedeterministického automatu.

Minimalizácia DFA

- Ekvivalentné stavy: Stavy q_i a q_j sú ekvivalentné, ak
  - oba patria do F, alebo ak oba patria do Q – F a
  - pre každé a $\in$ T také, že $\delta(q_i, a) \notin \{q_i, q_j\} \vee \delta(q_j, a) \notin \{q_i, q_j\}$, platí $\delta(q_i, a) = \delta(q_j, a)$.
- Je to lokálna na prechodovej tabuľke ľahko testovateľná vlastnosť
- Minimalizácia spočíva v „zlúčení ekvivalentných stavov.
  - Ak stavy q_i a q_j sú ekvivalentné, vyberieme jeden z nich napr. q_i a všetky výskyty druhého q_j v prechodovej tabuľke nim nahradíme. Na záver odstránime riadok zodpovedajúci q_j z tabuľky prechodov.

Simulácia DFA programom

- Vstup: reťazec zakončený $.
- Výstup: Accept (akceptujem) alebo Reject (zamietam)
- Reprezentácia: δ je move: **array**[0..|Q|- 1, 0..|T|- 1] **of** Q, **set** F
- Algoritmus:

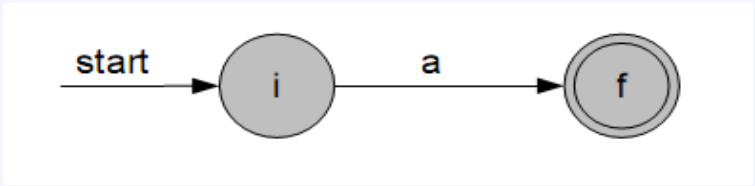
```
q:= q0;
Loop c:= getchar;
        if c = $ then Exit Loop;
        q:= move[q, c];
EndLoop;
if q in F then accept else reject;
```

Modifikácia pre scanner

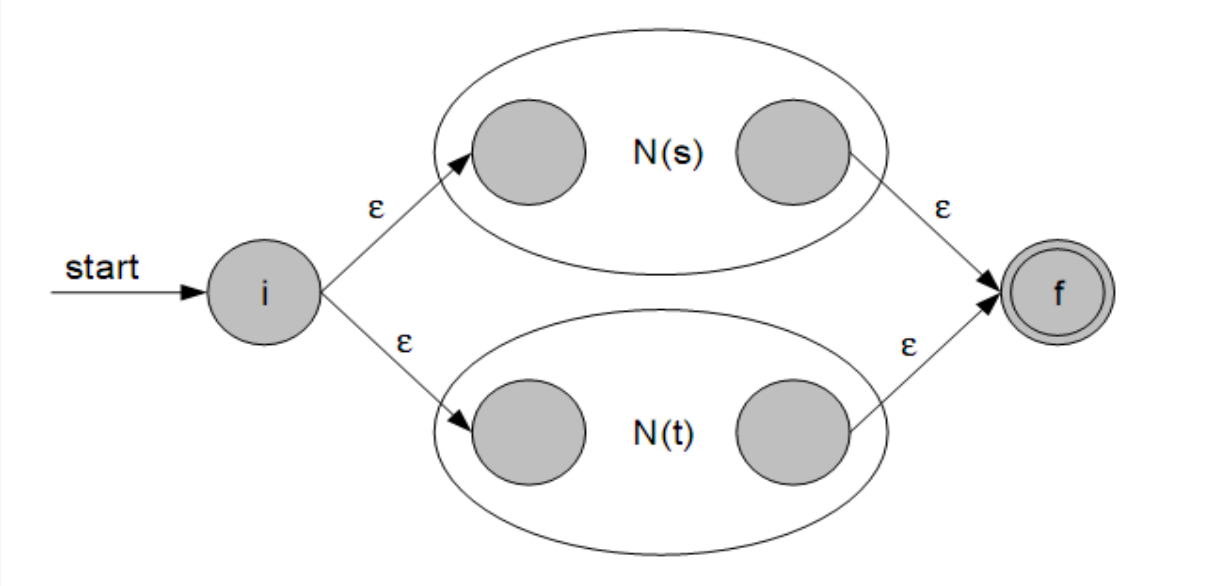
```
q:= q0;
Loop
        if q in F then
           report_accepted_string;
        c:= getchar;
        if c = $ then Exit Loop;
        q:= move[q, c];
EndLoop;
```

Thomsonova metóda 1

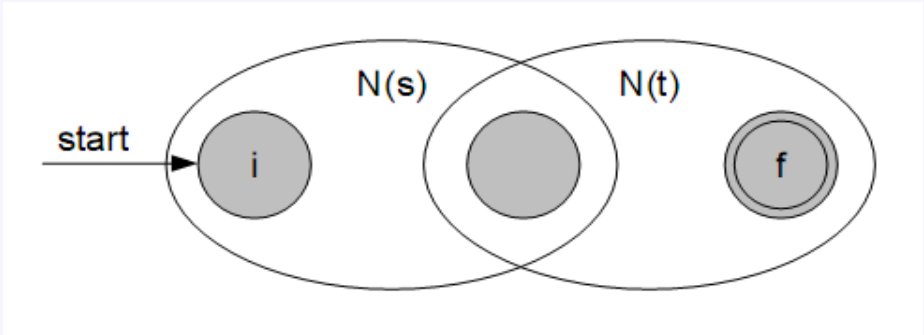
- NFA pre a∈T



- NFA pre s | t

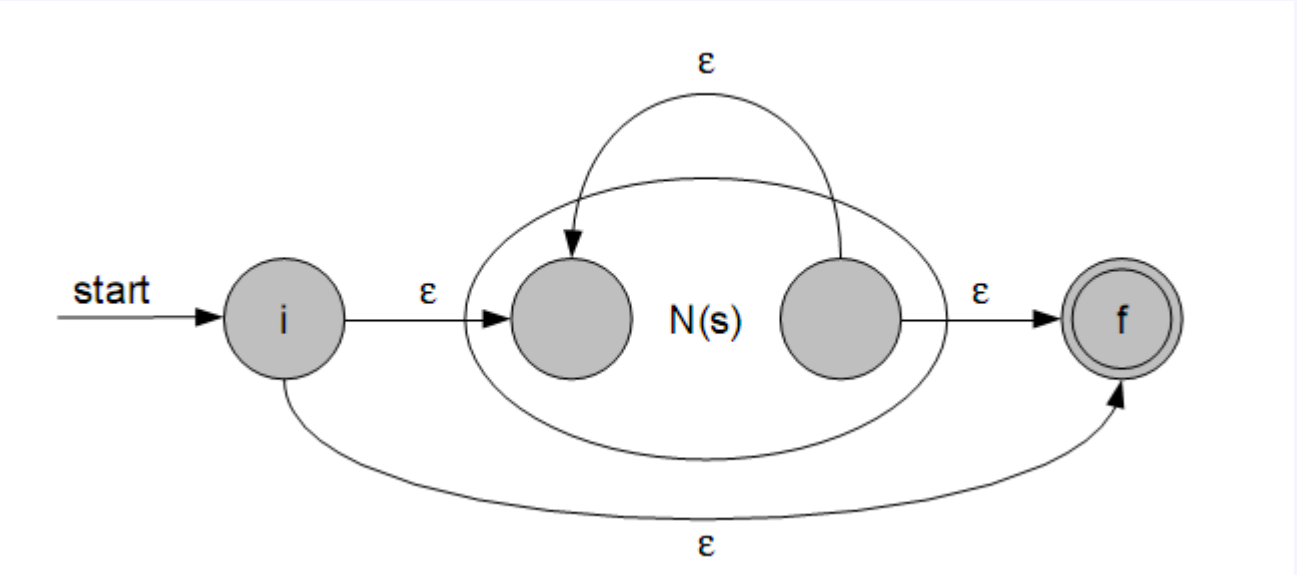


- NFA pre st



Thomsonova metóda 2

- NFA pre s*



Nekonštruujeme NFA pre ε. Môže sa vyskytnúť len v or konštrukcii. Vtedy prepojíme počiatočný a finálny stav ε-šipkou.
Automat získaný Thomsonovou metódou buď priamo simulujeme na počítači, alebo lepšie transformujeme na DFA minimalizujeme a potom simulujeme na počítači.

Hľadanie vzorky – pattern matching

- Algoritmus, ktorý hľadá všetky výskyty, všetkých vzoriek vo vstupnom reťazci.
- Tieto algoritmy spočívajú v transformácii vzoriek na datovú štruktúru (alebo konečný automat).
- Univerzálny program potom hľadá vzorky vo vstupnom reťazci.
- Pracuje to ako riešenie „skrývačky“, *fgrep* v Unixe.
- Niektoré vzorky sa dajú rozlíšiť až po na načítaní nasledujúceho znaku (napr. rezervované slovo **`begin`** a identifikátor **`beginner`**). Takéto „neprávom“ načítané znaky treba vrátiť na vstup.

Dömölkiho algoritmus – predspracovanie

- K množine vzoriek priradíme booleoskú maticu, ktorá bude mať toľko riadkov, koľko znakov má abeceda (trochu menej, všetkým znakom, ktoré sa v žiadnej vzorke nevyskytujú, môžeme priradiť ten istý nulový riadok.
- Vzorky usporiadame v nejakom poradí. Usporiadaným vzorkám zodpovedajú stĺpce matice.
- M[i, j] = 1 práve vtedy, keď i-tý znak sa vyskytuje na j-tej pozícii usporiadanej vzorky.
- Vektor začiatkov jednotlivých vzoriek u[j], a vektor v[j] koncov vzoriek pre $0 \leq j \leq m$ sú definované nasledovne:
  - u[j] = 1 práve pre tie pozície j, kde nejaká vzorka začína.
  - v[j] = 1 práve pre tie pozície j, kde nejaká vzorka končí.

Dömölkiho algoritmus – program

- Stav výpočtu je reprezentovaný vektorom q[1:m] inicializovaným ako q = **0**.
- **Loop**

  c:= getchar; **if** c = EOF **then** exit loop;

  i:= ord(c); /* convert c to row index of the matrix M */

  q:= (q>>1)∨ u) ∧ M[i];

  x:= q ∧ v;

  if x ≠ **0 then** identify_found_patterns;

  endLoop;
- >>1 je jednoduchý posun do prava s odstránením najpravejšieho bitu a doplnením nulou z ľava.
- identify_found_patterns je postupné hľadanie výskytu jedničky v x zľava. Normalizačný posun je vhodný.

Dömölkiho algoritmus – príklad

- Vzorky {ab, abcd, acd, dab}

matica M

	a	b	c	d	a	c	d	a	b
a	1	0	0	0	1	0	0	1	0
b	0	1	0	0	0	0	0	0	1
c	0	0	1	0	0	1	0	0	0
d	0	0	0	1	0	0	1	0	0

u = (1,0,0,0,1,0,1,0,0) začiatky vzoriek
v = (0,1,0,1,0,0,1,0,1) konce vzoriek

Urobil som miernú optimalizáciu, spojil som: vzorku ab a začiatok vzorky abcd a koniec vzorky acd so začiatkom vzorky dab. Takéto prelínacie optimalizácie nie sú nutné, ale môžu značne zmenšiť rozmer matice M a skrátiť dĺžku vektorov q, u, v.

Technické detaily

- Duálna podoba tohto algoritmu (jednotky a nuly a and a or sú zamenené) je známa ako Dömölki, Baeza – Yates, Gonnet SHIFT and OR algoritmus.
- Asi jediný dôvod pre duálnu podobu je inžinierská viera, že test na nulu je rýchlejší ako test na nie nulu.
- Ak sú vzorky príliš dlhé alebo strojové slovo príliš krátké možno vektory posekať na strojové slová. Treba ošetriť posun najľavejších bitov do predošlých slov ak existujú.
- Hlavným zdrojom neefektívnosti je veľmi neúsporné kódovanie stavov 2^m.
- Zatial rozpoznáva le konečnú množinu slov. Úpravu na regulárne a spätne deterministické jazyky poviem pri syntaktickej analýze.

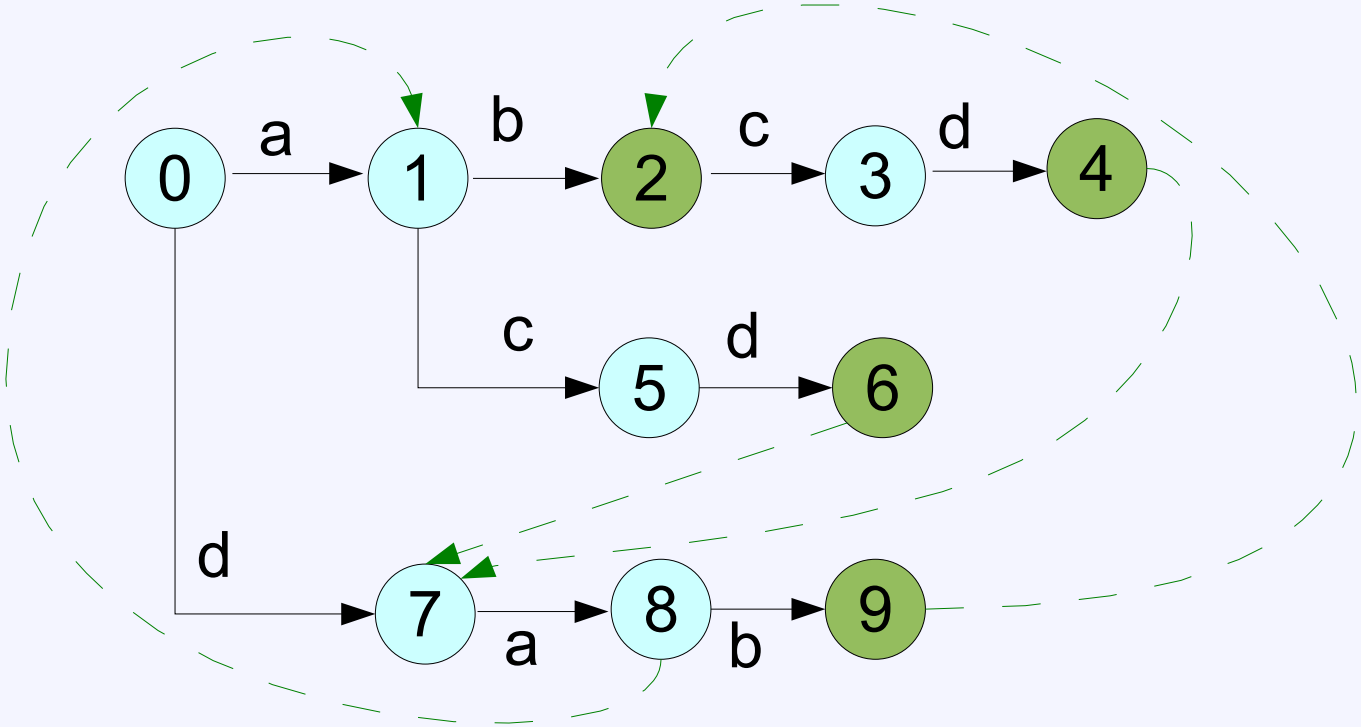
Aho Corasick – predspracovanie

- Vektor q umožňuje zaznamenať súčasne všetky možné kombinácie pozície vo všetkých vzorkách. Aj také, ktoré nikdy nemôžu nastať. Veľká časť hodnôt vektora q je nevyužitá.
- Trie pre množinu vzoriek definuje všetky prípustné (dosiahnuteľné) stavy.
- Trie definuje aj akceptujúce stavy, stavy v ktorých nejaká vzorka bola akceptovaná a prechody pri akceptovaní vzorky. Niekedy môže byť akceptovaných viac vzoriek súčasne.
- Ostatné prechody (fail) definujeme tak, aby najdlhší suffix zlyhanej vzorky tvoril najdlhší prefix cieľovej vzorky.

Aho Corasick – príklad

- Vzorky {ab, abcd, acd, dab}

Trie: acceptujúce stavy



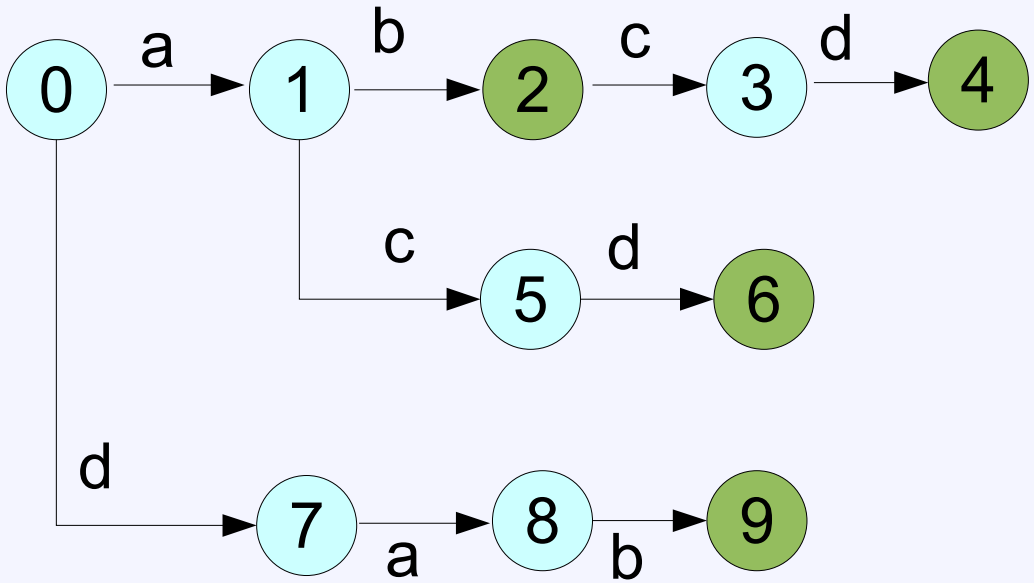
Prechová tabuľka

	a	b	c	d	Fail
٠	١			٧	٠
١		٢	٥		٠
٢			٣		٠
٣				٤	٠
٤					٧
٥				٦	٠
٦					٧
٧	٨				٠
٨		٩			١
٩					٢

Keď chceme prechodovú tabuľku plne naplniť, môžeme ušetriť ε prechody (fail) ušetriť. Či je to výhodné, závisí od implementácie, počtu znakov abecedy a ich kódovania.

Aho Corasick – implementácia

- Princíp: Žiaden vtip, len hrubá sila. Preč s ε-prechodmi!



Prechodová tabuľka

T

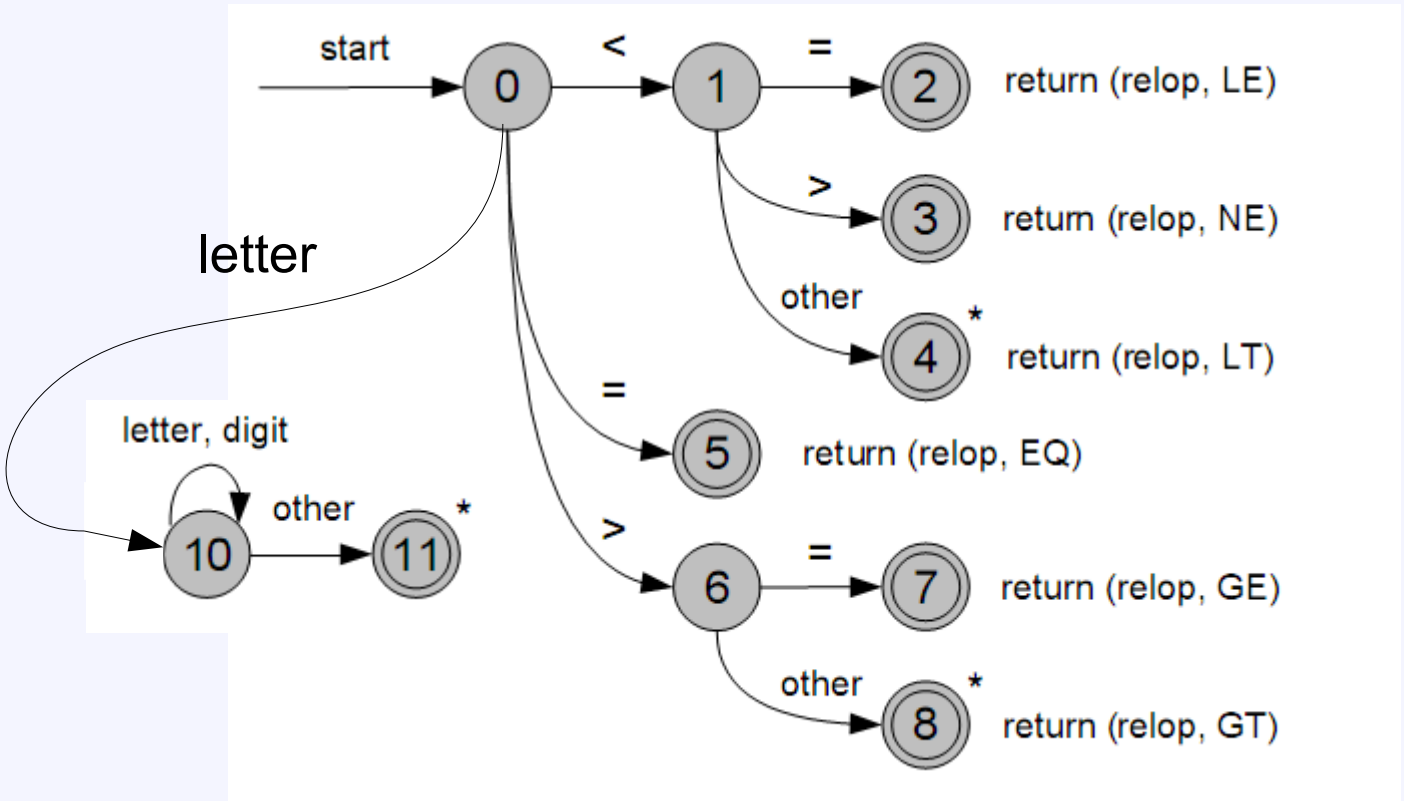
	a	b	c	d
0	1	0	0	7
1	1	2	5	7
2	1	0	3	7
3	1	0	0	4
4	8	0	0	7
5	1	0	0	6
6	8	0	0	7
7	8	0	0	7
8	1	9	0	7
9	1	0	3	7

Program:

```
        q:= 0;
 Loop
    if q is final_state then report_patterns;
    c:=getchar; if c=EOF then exit Loop;
    q:= T[q,c];
 endLoop;
```

Priama konštrukcia DFA

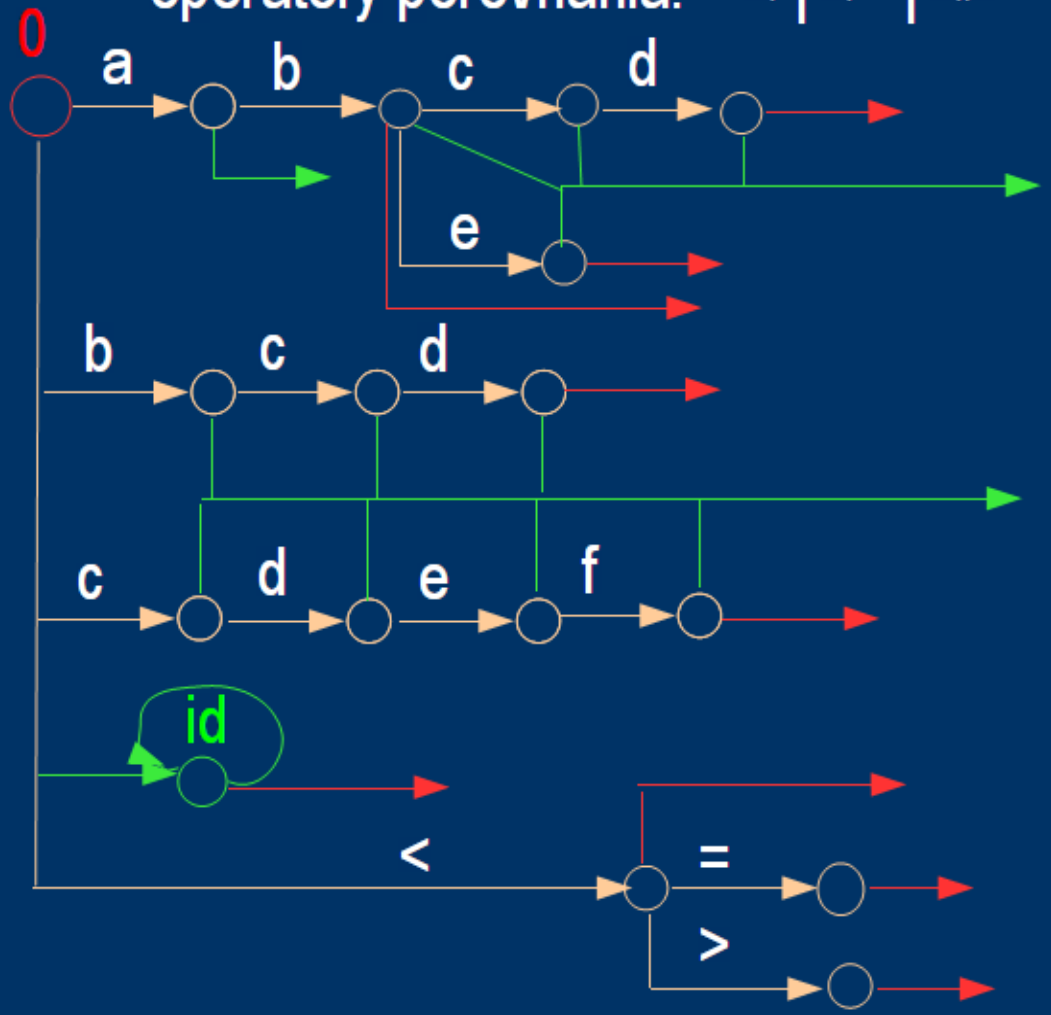
- Zostrojíme trie, keď to potrebujeme, načítame nasledujúci symbol. * označuje stavy v ktorých sa posledny načítany symbol vracia na vstup.
- Je to konečný automat, cykly nerobia problém.



Trochu je konflikt medzi identifikatorom a rezervovanými slovami.

Priorita prechodov

Príklad: rezervované slová: abcd | abe | ab | bcd | cdef
identifikátory: písmeno{písmeno|číslica}*
operátory porovnania: < | <= | <>



Oranžové šípky prechod do nasledujúceho stavu na znak, ktorým sú označené.
Zelené šípky prechod na písmeno alebo číslicu do stavu id (Zo stavu 0 iba na písmeno). Použije sa iba, ak sa oranžová šípka nedá použiť.
Červená šípka: akceptuje rozpoznaný reťazec a na prázdny symbol prechádza do počiatočného stavu. Použije sa iba, ak sa žiadná iná šípka nedá použiť.

Alternatívne implementácie

- Možná je priama implementácia
  - namiesto jedneho čítaneho symbolu dva current a lookahead.
  - čítanie {current:= lookahead; lookahead:= getchar;}
  - rozhodovanie o akcii na základe lookahead: **case** lookahead **of …**
  - ε-prechody (červené šípky) nečitajú.
- Konečne za cenu miernej straty efektívnosti môžeme rozpoznávať len identifikátory (slová) a rezervovanými slovami inicializovať tabuľku symbolov.
  - Ak jazyk má veľa rezervovaných slov môže to zredukovať prácu pri návrhu scannera.
  - Či identifikátor je rezervované slovo, sa zistí po jeho rozozznaní, konzultáciou s tabuľkou symbolov, prípadne to rozhodne až syntaktická analýza.

Spolupráca s tabuľkou symbolov

- Ukladajú sa sem ďaľšie informácie o inštanciách tokenov
  - Meno identifikátora, hodnota konštanty, atď.
- Pri rozpoznaní identifkátora sa najskôr skontroluje, či už je v tabuľke symbolov
  - Ak áno: vráti sa smerník na príslušný záznam
  - Ak nie: pridá sa nový záznam
- Predvyplnenie tabuľky rezervovanými slovami zjednodušuje lex. analýzu
- Funkcie:
  - `insert(s,t)` - vráti index nového záznamu pre reťazec `s`, token `t`
  - `lookup(s)` - vráti index záznamu pre reťazec `s` alebo 0 ak sa `s` nenájde

Implementácia tabuľky symbolov

- Uloženie reťazcov (lexém)
  1. Ohraničená tabuľka
     - Jednoduchá správa
     - Problém, ak máme príliš veľa identifikátorov alebo príliš dlhé identifikátory
  2. Tabuľka s premenlivou dĺžkou
     - Flexibilná, ale horšie sa spravuje

- Dátové štruktúry
  1. *Lineárny zoznam*
     - Jednoduchá implementácia, ale pomalé vyhľadávanie
  2. *Hašovacia tabuľka*
     - Hašovacia funkcia napr. $h(key) = num(key)\ mod\ SIZE$, kde *num* konvertuje vstupný reťazec na celé číslo
     - Zložitejšia implementácia, ale rýchlejšie vyhľadávanie

Realizácia tabuľky symbolov premenlivá dĺžka lexém