KOMPILÁTORY: Lexikálna analýza

Jana Dvořáková
dvorakova@dcs.fmph.uniba.sk

Úlohy lexikálnej analýzy

1. Primárna úloha: čítanie znakov zo vstupu a ich preklad na postupnosť *tokenov*, ktorú ďalej využije syntaktická analýza
2. Ďalšie úlohy:
   - Uloženie informácie o tokenoch do tabuľky symbolov
   - Odstránenie komentárov a bieleho priestoru (medzery, tabulátory, znaky nového riadku)
   - Zosúladiť chybové hlásenia so zdrojovým programom
     - Napr. priradenie lokalizácie chyby k chybovému hláseniu (číslo riadku, príslušnú časť kódu)

- Pracuje na úrovni *regulárnych jazykov*

Výhody oddelenia od syntaktickej analýzy

- Zjednodušenie návrhu
  - Lexikálna aj syntaktická analýza
  - Napr. zahrnutie prázdneho priestoru do gramatiky parsera by ju značne skomplikovalo
- Zvýšenie efektívnosti kompilátora
  - Špecializácia lex. analyzátora
- Zvýšenie portability kompilátora
  - Pri zmenenej reprezentácii znakov je potrebné modifikovať iba lexikálnu analýzu

Tokeny, lexémy, patterny

- *Token:*
  - Reprezentuje množinu reťazcov so spoločným významom
  - Výstup lexikálnej analýzy a vstup syntaktickej analýzy
  - Z pohľadu syntaktickej analýzy je to terminál
  - Zvyčajne sú to: rezervované slová, operátory, identifikátory, konštanty (numerické, reťazcové, znakové), oddeľovače
- *Pattern:*
  - Pravidlo popisujúce množinu reťazcov pre daný token
  - Zvyčajne špecifikovaný regulárnym výrazom
- *Lexéma:*
  - Postupnosť znakov v zdrojovom programe, ktorá zodpovedá patternu pre nejaký token

Lex. analyzátor rozpoznáva lexémy v zdrojovom programe a prekladá ich na príslušné tokeny.

Tokeny, lexémy, patterny

Príklad

Vstup: `dráha := počiatok + čas * 60`
Výstup: **id assign id op_plus id op_mul num**

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)+

Ošetrenie chýb

- Lex. analýza zvyčajne odhalí iba malú časť chýb
- Chyba nastane ak postupnosť čítaných znakov zo vstupu nezodpovedá žiadnemu patternu
- Spôsoby zotavenia
  1. Vymazanie znakov zo vstupu, kým sa nenájde reťazec zodpovedajúci niektorému patternu
  2. Vloženie znakov navyše
  3. Výmena nesprávneho znaku za správny
  4. Výmena susedných znakov

Vstupné rozhranie LA

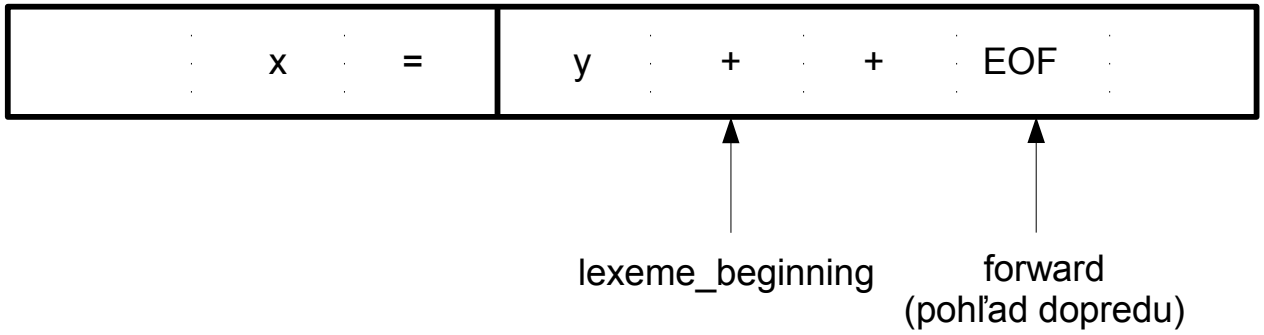
- Používa sa *vstupný buffer*
  - LA je jediná fáza čítajúca vstup znak po znaku; zaberá značnú časť času kompilácie
  - Niekedy je potrebné prečítať viac znakov zo vstupu na rozpoznanie lexémy ako je jej dĺžka (lookahead) a potom prebytočné znaky vrátiť späť na vstup
  - Použitie bufferu urýchľuje čítanie:

1. Naraz je načítaný jeden blok znakov
2. Pozícia práve spracovávaného znaku je v bufferi označená smerníkom
3. Čítanie a spätné vrátenie znakov na vstup je riešené presunutím smerníka

Dvojbufferová schéma

- Buffer rozdelený na dve polovice o veľkosti *N*
- Jedným príkazom sa do každej polovice načíta *N* znakov
- Dva smerníky:
  - *lexeme_beginning* - začiatok lexémy
  - *forward* - posúva sa dopredu, kým nerozpozná nejakú lexému a nastaví sa na jej koniec
- Po spracovaní lexémy sa oba smerníky posunú za jej koniec
- Ak *forward* prejde hranicu jednej z polovíc buffera, druhá sa naplní novými znakmi zo vstupu a presunie sa na jej začiatok
- Lookahead je obmedzený (problém, ak sa je potrebné na rozpoznanie nejakej lexémy pozrieť ďaleko dopredu)

Dvojbufferová schéma

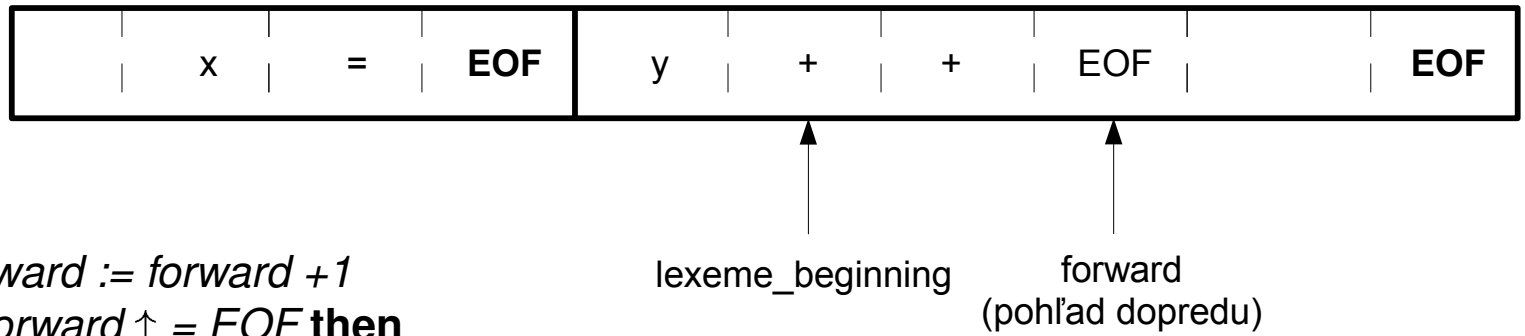


if *forward na konci 1. polovičky* **then**
begin
    načítaj 2. polovičku
    forward := forward + 1
end
else if *forward na konci 2. polovičky* **then**
begin
    načítaj 1. polovičku
    forward := 0
end

Dvojbufferová schéma

Použitie zarážok

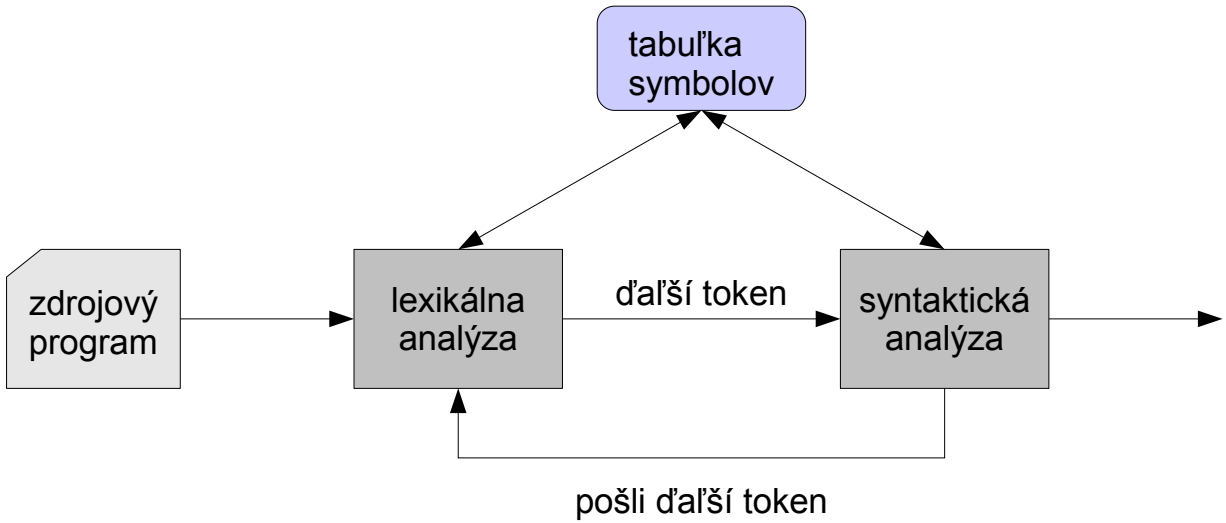
- Zníži sa počet potrebných testov




```
forward := forward +1
if forward ↑ = EOF then
begin
    if forward na konci 1. polovičky then
    begin
        načítaj 2. polovičku
        forward := forward + 1
    end
    else if forward na konci 2. polovičky then
    begin
        načítaj 1. polovičku
        forward := 0
    end
    else /* EOF označujúce koniec súboru */
        ukonči lexikálnu analýzu
end
```

Výstupné rozhranie LA

- Používa sa *výstupný buffer s tokenmi*
  (= rozhranie medzi LA a SA)
  - Lex. analyzátor = producent, parser = konzument
  - Lex. analyzátor produkuje tokeny, dáva ich do buffera a parser ich odtiaľ podľa potreby odoberá ("konzumuje")
  - Buffer má obvykle *veľkosť 1* (obsahuje iba jeden token) a LA je *procedúrou volanou SA*
  - Okrem tokenu posiela lex. analyzátor ďalej aj atribúty tokenu (hodnota, smerník do tabuľky symbolov,..)



Patterny: špecifikácia tokenov

Regulárne výrazy

1. Symbol ε je regulárny výraz označujúci $\{\varepsilon\}$.
2. Ak $a \in \Sigma$, potom a je regulárny výraz označujúci $\{a\}$.
3. Ak r a s sú regulárne výrazy označujúce jazyky $L(r)$ a $L(s)$, potom:
   a) $(r)|(s)$ je regulárny výraz označujúci $L(r) \cup L(s)$
   b) $(r)(s)$ je regulárny výraz označujúci $L(r)L(s)$
   c) $(r)^*$ je regulárny výraz označujúci $L(r)^*$
   d) (r) je regulárny výraz označujúci $L(r)$

Patterny: špecifikácia tokenov

Príklad

- Regulárne definície (pomenované regulárne výrazy):

letter	→	`[A-Za-z]`
digit	→	`[0-9]`
delim	→	**blank** | **tab** | **newline**
ws	→	(**delim**) *
while	→	`while`
relop	→	`<` | `<=` | `=` | `<>` | `>` | `>=`
id	→	**letter** (**letter** | **digit**) *
num	→	(**digit**) +

- Používajú sa niektoré skratky v zápise regulárnych výrazov

`r+`	1 a viac výskytov
`r?`	0 alebo 1 výskyt
`[a-z]`	trieda znakov, `a` | `...` | `z`

Zločiny lexikálnej analýzy

- Odsadenie na vstupnom riadku
  - Odsadenie ako syntaktická konštrukcia (Python, Flex)
- Identifikátory
  - Povolené medzery v mene identifikátora, napr. Fortran:

    `DO 5 I = 1.25` (identifikátor `DO5I`)
    `DO 5 I = 1,25` (kľúčové slovo `DO`)
  - Kľúčové slová nie sú rezervované a môžu byť použité ako identifikátory, napr. PL/I:

    ```
    IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
    ```
- Kontextovo závislé tokeny, napr. PL/I:

  ```
  DECLARE(ARG1,ARG2, ... , ARGn)
  ```

  kľúčové slovo alebo názov poľa?
- Aj moderné jazyky majú problémy, napr. C++:

  template: `Foo < Bar >`, stream: `cin >> var`

  konflikt s vnorenými template: `Foo < Bar < Baz >>`

Atribúty tokenov

- Lex. analyzátor vracia v skutočnosti dvojicu *(token, atribút)*
- Atribút je upresnenie konkrétnej inštancie tokenu, pre synt. analýzu zväčša nemá význam ale využíva sa v ďalších fázach pri preklade
- **ws** nevracia žiadny token, je to oddeľovač tokenov

Príklad: `while (i <= 25) j++;`

LEXÉMA	TOKEN	ATRIBÚT
`while`	**while**	-
`(`	**left_par**	-
`i`	**id**	smerník do tabuľky symbolov
`<=`	**relop**	`LE`
`25`	**num**	smerník do tabuľky symbolov/hodnota
`)`	**right_par**	-
`j`	**id**	smerník do tabuľky symbolov
`++`	**op_inc**	-
`;`	**semicolon**	-

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

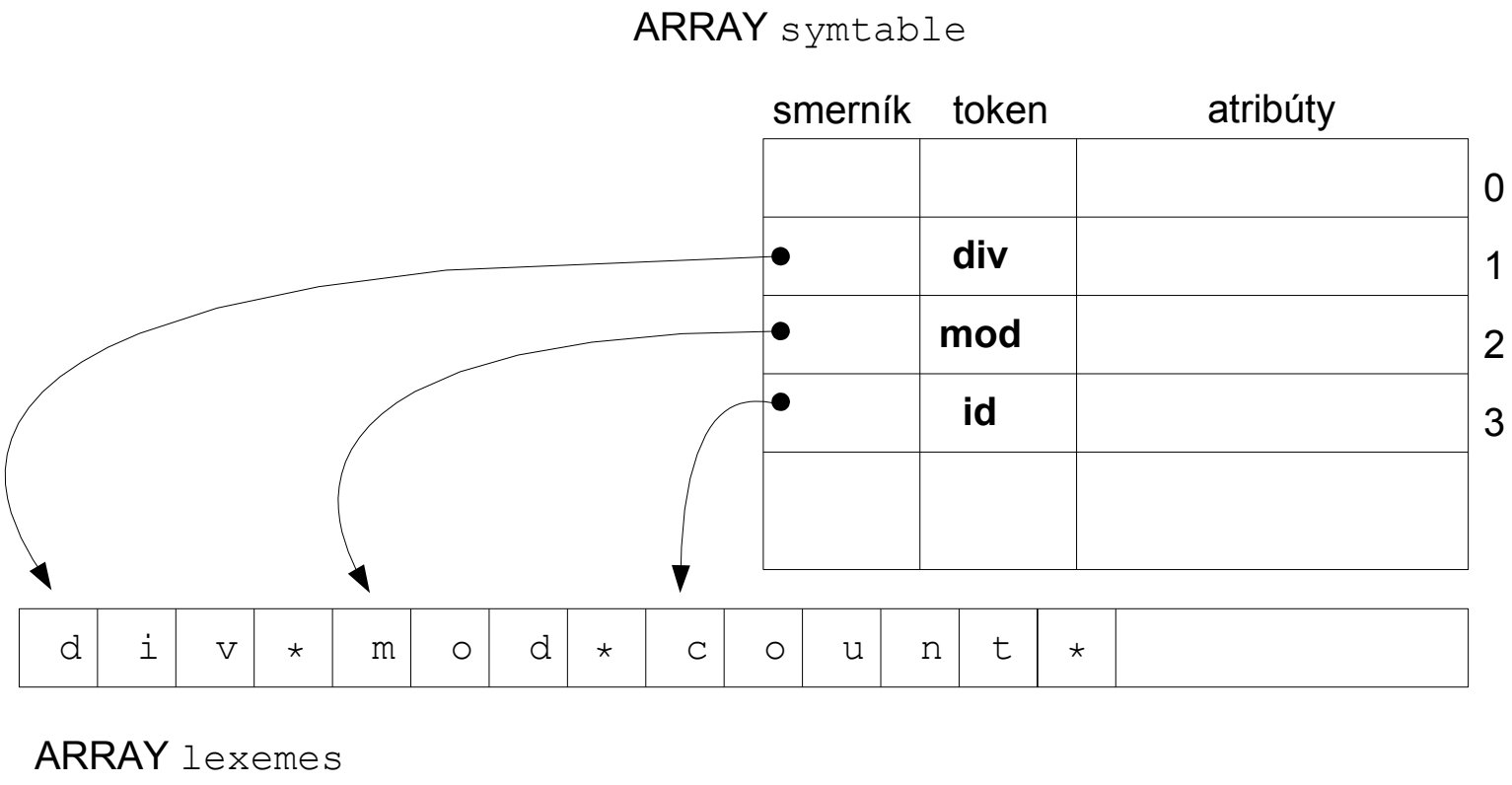
- Funkcie:
  - `insert(s,t)` - vráti index nového záznamu pre reťazec `s`, token `t`
  - `lookup(s)` - vráti index záznamu pre reťazec `s` alebo 0 ak sa `s` nenájde

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

Implementácia tabuľky symbolov

- Realizácia (premenlivá dĺžka identifikátorov):



Tvorba lexikálneho analyzátora

1. Najskôr definuj množinu tokenov
   - Tokeny by mali zahŕňať typické nerekurzívne konštrukcie vstupného jazyka
   - Výber tokenov závisí na *vstupnom jazyku* a *návrhu parsera*
2. Vytvor patterny pre jednotlivé tokeny
3. Implementuj rozpoznávanie patternov
   - Tento krok môže byť automatický ak existuje taký nástroj

Tvorba lexikálneho analyzátora

Metódy

1. Prechodové diagramy
   - Patterny sa špecifikujú pomocou *prechodových diagramov*
   - Efektívna metóda, ale náročnejšia
2. Thomsonova metóda
   - Patterny sa špecifikujú pomocou *regulárnych výrazov*
   - Implementácia:
     - Algoritmus zostrojenia NKA k regulárnym výrazom
     - Simulácia NKA (resp. vytvorenie DKA a simulácia)
   - Sú na nej založene generátory lex. analyzátorov
   - Jednoduchá metóda, ale menej efektívna
3. Naprogramovanie v programovacom jazyku
   - Niektoré jazyky priamo podporujú funkcie na kontrolu typu znaku (v C sú to napr. *is_digit*(), *is_letter*())
4. (Zahrnutie do syntaktickej analýzy)

- Súvisiaci problém: vyhľadávanie patternov v texte

Konečné automaty

$A = (Q, \Sigma, \delta, q_0, F)$
Q - konečná množina stavov
Σ - vstupná abeceda
$q_0 \in Q$ - počiatočný stav
$F \subseteq Q$ - množina akceptačných stavov

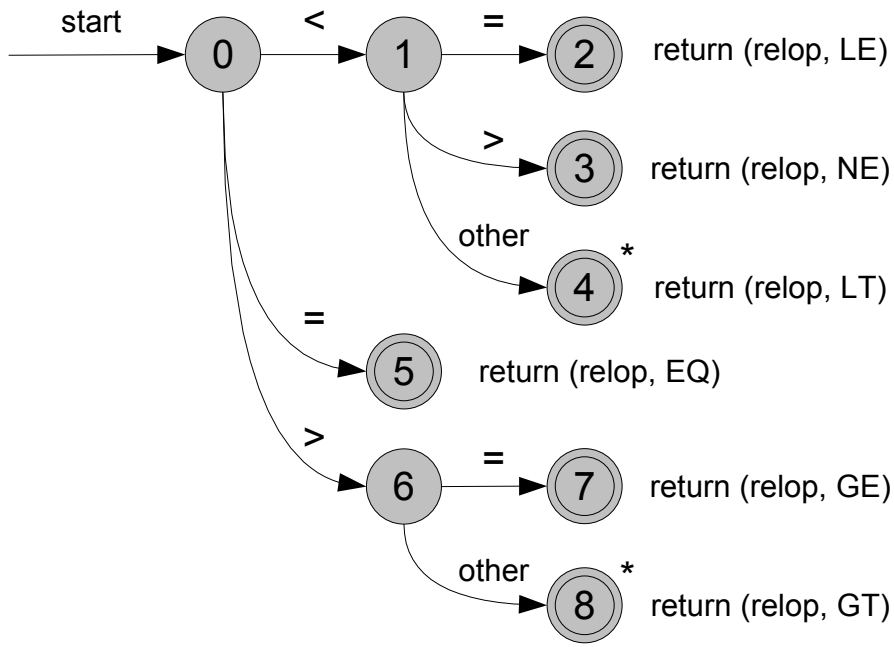
1. DKA: $\delta : K \times \Sigma \to K$
2. NKA: $\delta : K \times \Sigma \to 2^K_{kon}$

- Rozpoznávajú regulárne jazyky

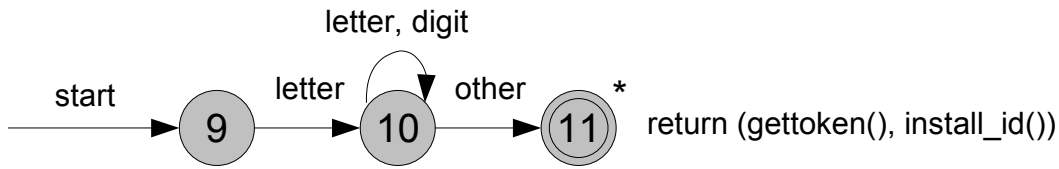
Prechodové diagramy

Špecifikácia patternov

- Je potrebné zostrojiť množinu prech. diagramov, každý špecifikuje skupinu tokenov
- Relačné operátory:



- Identifikátor:



* označuje vrátenie posledne prečítaného znaku na vstup

Prechodové diagramy

Implementácia

- Premenné pre aktuálny stav (*state*) a počiatočný stav aktuálneho prech. diagramu (*start*)
- Hrany sú implementované pomocou prechodovej tabuľky - veľkosť *počet stavov* $\times$ *počet znakov*
- Lookahead sa využíva iba pri ε-prechodoch pri koncových stavoch
- Algoritmus:
  1. Na začiatku máme *start* $:= 0$, *state* $:= 0$
  2. Posúvanie medzi stavmi po hranách podľa prečítaného znaku zo vstupu (mení sa *state*)
  3. Ak sa zasekne, skúša sa ďalší diagram (do *start* aj *state* sa priradí jeho poč. stav)
  4. Ak sa zasekne aj v poslednom diagrame - lexikálna chyba

Thomsonova metóda

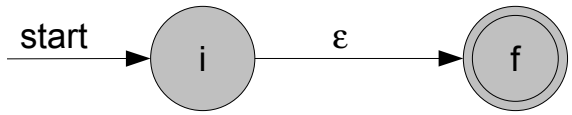
Špecifikácia patternov

- Regulárne výrazy

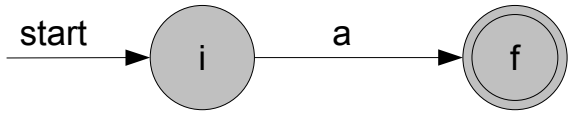
Thomsonova metóda

Zostrojenie NKA k reg. výrazom (1)

- NKA pre ε:

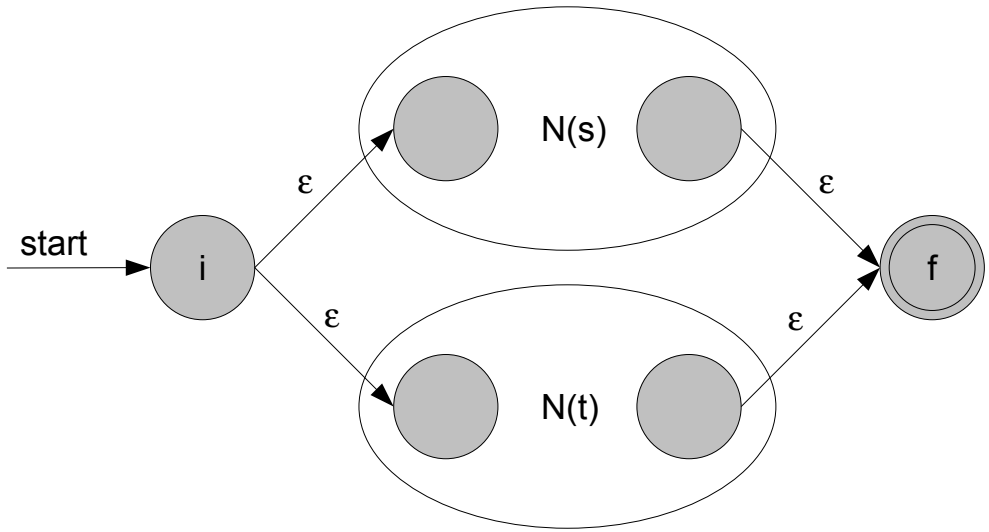


- NKA pre a:



Nech $N(s)$ je NKA pre s a $N(t)$ je NKA pre t.

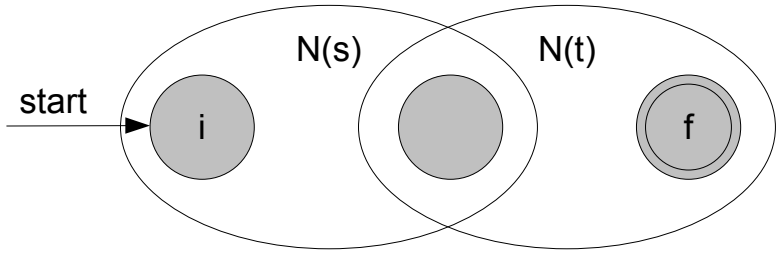
- NKA pre $s|t$:



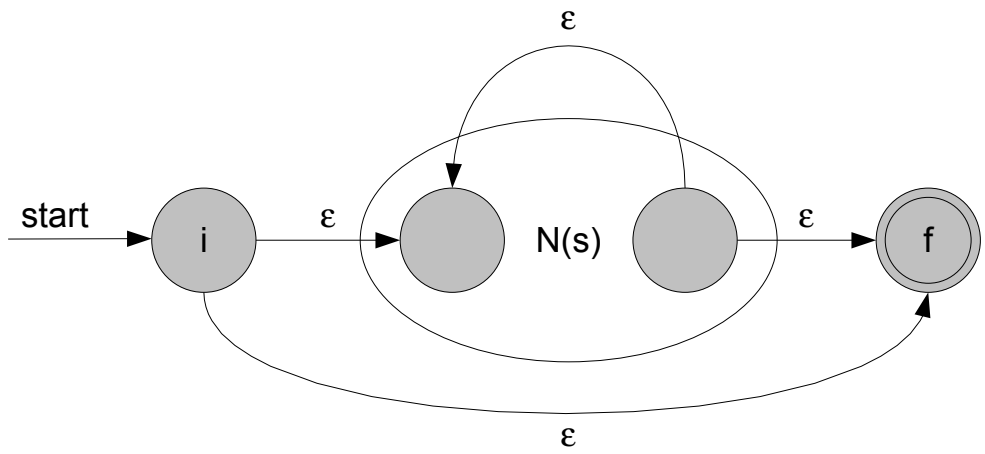
Thomsonova metóda

Zostrojenie NKA k reg. výrazom (2)

- NKA pre st:



- NKA pre s^*:



- NKA pre (s) = NKA pre s, t.j. $N(s)$

Každému pridávanému stavu dáme nové meno.

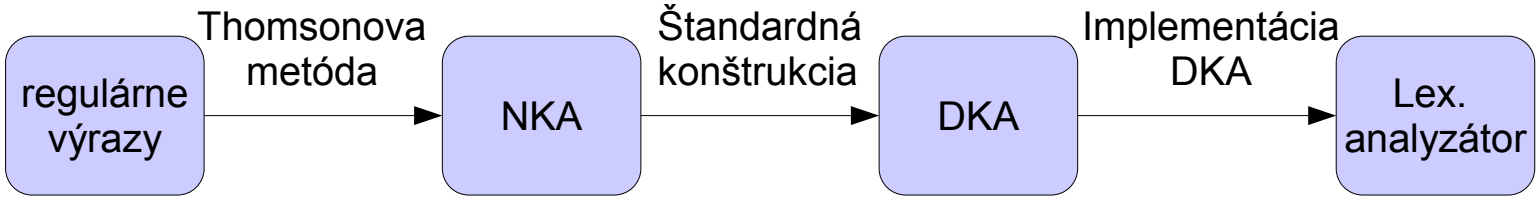
Thomsonova metóda

Implementácia NKA

- 2 prístupy:

1. Priama simulácia zostrojeného NKA
   - Aplikuje sa algoritmus zostrojenia DKA k NKA *za behu*
   - Priestor: $O(|r|)$, čas: $O(|r| \times |x|)$
     ($|r|$ je dĺžka reg. výrazu a $|x|$ je dĺžka vstupného reťazca)

2. Zostrojenie ekvivalentného DKA štandardnou konštrukciou, simulácia DKA:
   - Priestor: $O(2^{|r|})$, čas: $O(|x|)$
   - Zostrojený DKA sa ešte minimalizuje (počet stavov)



Algoritmus konštrukcie DKA k NKA

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**
    $U :=$ ε-*closure*(*move*(q, a));
    if U nie je v *Dstates* **then**
      pridaj U do *Dstates* ako neoznačený stav;
    Dtran$[q, a] := U$;
  end;
end;

- 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;

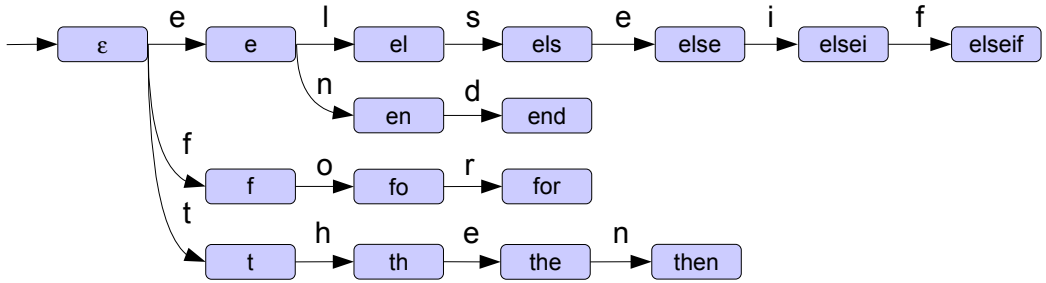
Algoritmus simulácie DKA

Vstup: vstupný reťazec x zakončený **eof**,
DKA $D = (K, \Sigma, \delta, q_0, F)$.
Výstup: Odpoveď "ánoäk D akceptuje x; inak odpoveď "nie"

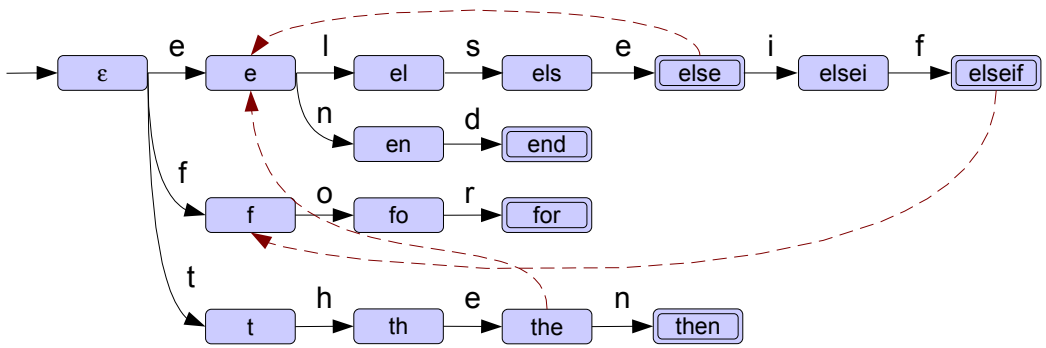
$q := q_0$;
$c :=$ *nextchar*;
while $c \neq$ **eof do begin**
  $q := move(q, c)$; (podľa prechodovej tabuľky)
  $c :=$ *nextchar*;
end;
if $q \in F$ **then**
  return "*yes*"
else return "*no*";

Algoritmus Aho-Corasick

- Lokalizuje prvky konečnej množiny patternov P
- Algoritmus najskôr pre P skonštruuje konečný automat a potom ho použije na vstupný text
- Príklad: $P = \{elseif, else, end, for, then\}$, postup konštrukcie:
  1. Zostrojíme strom $T(P)$, kde hrany sú označené písmenkami a návestia vrcholov sú prefixy prvkov P



  2. Pridáme "fail"hrany (prechody na ε v prípade, ak sa nedá pokračovať žiadnou písmenkovou hranou). Fail hrany existujú v skutočnosti pre každý vrchol - všetky okrem tých, ktoré sú vyznačené na obrázku, vedú do koreňa stromu. Vrcholy zodpovedajúce prvkom P označíme ako koncové stavy a získame prechodový diagram konečného automatu.



- Používa sa napr. vo vírusových databázach a vo funkcii `fgrep` v UNIXe
- Na rozdiel od techník lex. analýzy hľadané reťazce nemusia byť oddelené bielym priestorom (môžu sa aj prekrývať)
- Časová zložitosť je lineárna vzhľadom na dĺžku vzoriek plus dĺžku prehľadávaného textu

Flex - generátor lex. analyzátora