Prednáška 8 - Lexikálna analýza, FLEX

Ing. Viliam Hromada, PhD.

C-510
Ústav informatiky a matematiky
FEI STU

viliam.hromada@stuba.sk

Jazykové procesory a kompilátory

- **Jazykový procesor** je softvér, ktorého úlohou je spracovanie počítačových jazykov. Najčastejšie rieši tieto 2 úlohy:
  - *transformácia* dokumentu (programu) v zdrojovom jazyku do ekvivalentného dokumentu v inom (cieľovom jazyku),
  - *priame spracovanie (interpretácia)* dokumentu (programu) v zdrojovom jazyku.
- Jazykový procesor môže napríklad transformovať dokument pripravený vo formáte LaTeXdo formátu dvi alebo pdf, t.j. takého, ktorý je na nižšej úrovni (bližší jazykom, ktoré vedia interpretovať technické zariadenia).
- Následne sa dvi/pdf konvertuje do jazyka, ktorý je interpretovaný výsledným zariadením. Pritom je jednoduchšie naprogramovať softvér pre takúto konverziu, ako vyvíjať softvér, ktorý by priamo konvertoval LaTeXdo jazyka výstupného zariadenia.

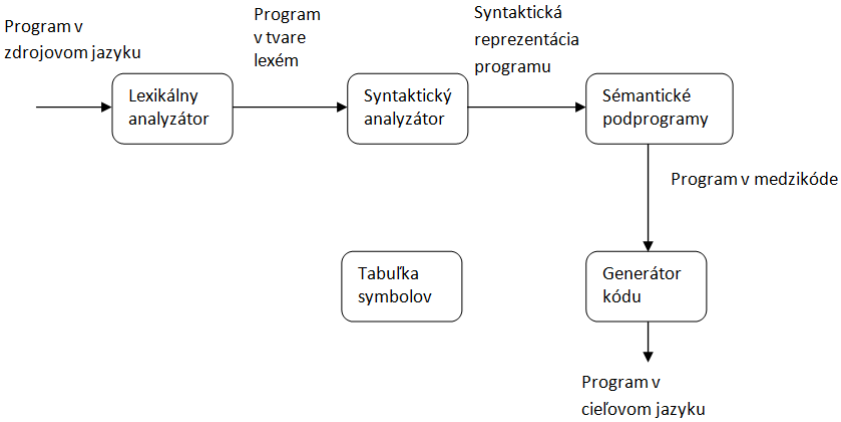
- Taktiež sa jazykové procesory používajú v oblasti tvorby počítačových programov.
- Tu sa dajú rozdeliť na 2 hlavné skupiny:
  - **kompilátory (prekladače)** - transformujú zápis (zdrojový kód) vo vyššom programovacom jazyku do iného jazyka (napr. strojovo-orientovaného jazyka),
  - **interpretátory** - priamo vykonávajú (interpretujú) zápis vo vyššom programovacom jazyku.
- Špeciálny význam má prekladač asemblera, pretože niektoré kompilátory ho využívajú ako medzistupeň medzi vyšším programovacím jazykom a strojovým kódom.

- Okrem **strojového jazyka**, ktorý využíva konkrétne inštrukcie procesora podľa cieľovej platformy môže kompilátor preložiť zdrojový kód aj do **virtuálneho strojového jazyka** - využíva inštrukcie *virtuálneho stroja*
- Pre virtuálny strojový jazyk je jednoduchšie zostrojiť interpretátor alebo kompilátor pre konkrétnu platformu (HW + OS), umožňuje tento prístup jednoduchšiu tvorbu kompilátorov prenositeľných medzi platformami. Príkladom virtuálneho strojového jazyka je Java bytecode.

Cieľový jazyk môžeme uvažovať v nasledovných formátoch:

- *asembler* - tzn. symbolický formát,
- *relokovateľný formát* (.o, .obj) - binárny formát, v ktorom nie sú vyriešené odkazy na externé referencie a lokálne absolútne adresy. Umožňuje vykonávať separátnu kompiláciu jednotlivých modulov programu, ktoré sa neskôr navzájom pospájajú spolu s knižnicami (*zlinkujú* pomocou **linkera**) a vytvorí sa spustiteľný binárny program.
- *pamätový obraz* - výstup z kompilátora sa priamo umiestni do operačnej pamäte a spustí (t.j. výsledný program je priamo v strojom kóde).

Funkčný pohľad na kompilátor



- **Lexikálny analyzátor** - výstupom sú **lexémy** (lexikálne jednotky), ktoré zodpovedajú základným symbolom programovacieho jazyka (identifikátory, kľúčové slová, konštanty,...).
- Tie spracuje **syntaktický analyzátor**, ktorý vytvára syntaktickú reprezentáciu programu, ktorá súvisí s bezkontextovou gramatikou, popisujúcou syntax daného jazyka.

- Výstup syntaktickej analýzy spracúvajú **sémantické podprogramy**, ktoré interpretujú syntaktickú štruktúru kódu a na základe interpretácie generujú kód. Zvyčajne negenerujú priamo strojový kód, ale tzv. *medzikód*; čo je jazyk blízky strojovému, avšak s istou úrovňou abstrakcie - napr. od konkrétneho počtu a typu registrov a režimu adresácie. Na medzikóde sa dajú vykonávať optimalizácie a z neho následne generovať výsledný kód.
- **Tabuľka symbolov** uchováva informácie o atribútoch identifikátorov (mená premenných, hodnoty a typy konštánt) a využívajú ju hlavne sémantické podprogramy.

Príklad - gramatika

Uvažujme gramatiku, nech počiatočný neterminál je <prikazy>:

- <prikazy> → <prikaz><prikazy> | ε
- <prikaz> → **id =** <hodnota> **;**
- <prikaz> → **id =** <hodnota> <operator> <hodnota> **;**
- <prikaz> → **while (** <podmienka> **) { ** <prikazy> **}**
- <podmienka> → <hodnota><komparator><hodnota>
- <komparator> → **>** | **<** | **==** | **!=** | **>=** | **<=**
- <hodnota> → **id** | **konst_int**
- <operator> → **+** | **-** | *****

Príklad - lexikálne elementy (terminály)

Nech lexikálne elementy (terminály gramatiky) sú definované ako:

- **(,), {, }** (oddeľovače ľavá/pravá okrúhla, množinová zátvorka)
- **;** (bodkočiarka)
- **+,-,*** (operátory plus, mínus, krát)
- **>,<,>=,<=,!=,==** (relačné operátory)
- **=** (rovná sa / operátor priradenia)
- **while** (kľúčové slovo while)
- **konst_int** (celočíselná konštanta, predstavuje každé číslo spĺňajúce regex $(0)|(1|...|9)(0|1|...|9)^*$
- **id** (identifikátor, predstavuje ľubovoľný reťazec symbolov spĺňajúci regex $(a|..|z|A|..|Z)(a|..|z|A|..|Z|0|..|9)^*$)

Príklad - zdrojový kód

Uvažujme reťazec symbolov predstavujúci zdrojový kód:

```
i1 = 5;
i2 = 1;
while(i1 >= 1) {
    i2 = i2 * i1;
    i1 = i1 - 1;
}
```

Príklad - lexikálna analýza

V prvej fáze lexikálna analýza **tokenizuje** reťazec (zdrojový kód) a prepisuje ho ako postupnosť lexikálnych elementov (terminálov) gramatiky - podľa toho **ako sú definované lexikálne elementy**, t.j. aké regexy ich predstavujú.

1. **id** (s hodnotou *i1*)
2. **=**
3. **konst_int** (s hodnotou *5*)
4. **;**
5. **id** (s hodnotou *i2*)
6. **=**
7. **konst_int** (s hodnotou *1*)
8. **;**

Príklad - lexikálna analýza

9. **while**
10. **(**
11. **id** (s hodnotou *i1*)
12. **>=**
13. **konst_int** (s hodnotou *1*)
14. **)**
15. **{**
16. **id** (s hodnotou *i2*)
17. **=**
18. **id** (s hodnotou *i2*)
19. *****
20. **id** (s hodnotou *i1*)
21. **;**

Príklad - lexikálna analýza

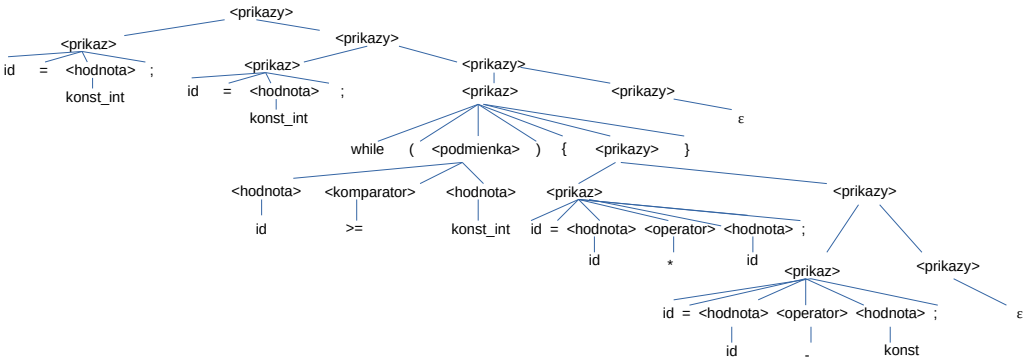
22. **id** (s hodnotou *i1*)
23. **=**
24. **id** (s hodnotou *i1*)
25. **-**
26. **id** (s hodnotou *1*)
27. **;**
28. **}**

Príklad - syntaktická analýza

Zdrojový kód sa teda rozdelí na postupnosť terminálnych symbolov. V ďalšej fáze (syntaktická analýza) **syntaktický analyzátor** zisťuje, či má daná postupnosť terminálnych symbolov deriváciu a ak áno, ako vyzerá derivačný strom.

Derivačný strom je dôležitý, pretože gramatika je spravidla konštruovaná tak, aby sa na základe derivačného stromu dal generovať medzikód pomocou sémantických podprogramov. Napríklad teda musí byť z derivačného stromu zrejmé, kde začínajú / končia vnorené bloky príkazov, čo predstavuje podmienku cyklu / vetvenia, atď.

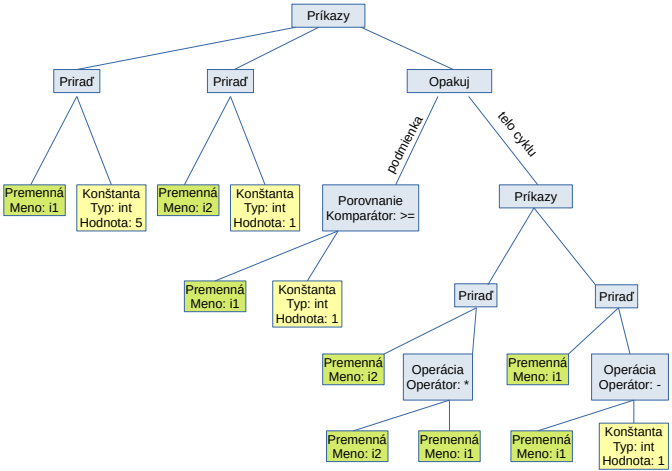
Príklad - derivačný strom



Príklad - sémantická analýza

- Cieľom syntaktickej analýzy je teda zistiť, či je vstupný reťazec **syntakticky** v poriadku a zostrojiť jeho **derivačný strom**.
- Následne sa na základe derivačného stromu vykonáva **sémantická analýza**, v ktorej sa zisťuje, čo program vykonáva, resp. sa vyrobí jeho **ekvivalentná** reprezentácia v tzv. medzikóde.
- Jedným z rôznych foriem medzikódu je aj tzv. abstraktný syntaktický strom - jedná sa o syntaktický strom, v ktorom sa abstrahuje od konkrétnych kľúčových slov a sú v ňom zachytené len informácie dôležité z hľadiska sémantiky (činnosti) programu.

Príklad - medzikód



Na základe medzikódu je potom možné generovať výsledný kód v cieľovom jazyku (asembler, strojový kód, iný programovací jazyk, atď.)

Cieľový jazyk - asembler

Save/Load + Add new... Vim CppInsights Quick-bench C++

```
// Type your code here, or load an example.
#include<stdio.h>

int f() {
    int i1;
    int i2;
    i1 = 5;
    i2 = 1;
    while(i1 >= 1)
    {
        i2 = i2 * i1;
        i1 = i1 - 1;
    }
    return i2;
}
```

x86-64 gcc 10.3 Compiler options...

A Output... Filter... Libraries + Add new...

```
1  f():
2          push    rbp
3          mov     rbp, rsp
4          mov     DWORD PTR [rbp-4], 5
5          mov     DWORD PTR [rbp-8], 1
6  .L3:
7          cmp     DWORD PTR [rbp-4], 0
8          jle     .L2
9          mov     eax, DWORD PTR [rbp-8]
10         imul    eax, DWORD PTR [rbp-4]
11         mov     DWORD PTR [rbp-8], eax
12         sub     DWORD PTR [rbp-4], 1
13         jmp     .L3
14 .L2:
15         mov     eax, DWORD PTR [rbp-8]
16         pop     rbp
17         ret
```

Cieľový jazyk - asembler

urce #1

Save/Load | Add new... | Vim | CppInsights | Quick-bench | C++

```
// Type your code here, or load an example.
#include<stdio.h>

int f() {
    int i1;
    int i2;
    i1 = 5;
    i2 = 1;
    while(i1 >= 1)
    {
        i2 = i2 * i1;
        i1 = i1 - 1;
    }
    return i2;
}
```

x86-64 gcc 10.3 (Editor #1, Compiler #1) C++

x86-64 gcc 10.3 | -O2

A | Output... | Filter... | Libraries | Add new...

```
1  f():
2          mov     eax, 120
3          ret
```

Lexikálna analýza

- **Lexikálny analyzátor** je prvou súčasťou kompilátora.
- Transformuje vstupný program do tvaru postupnosti lexém.
- **Lexémy** predstavujú základné symboly zdrojového jazyka; typicky sú to:
  - identifikátory,
  - konštanty rôznych typov,
  - kľúčové slová,
  - operátory,
  - oddeľovače.

- Každý typ lexémy má svoj pridelený (číselný) kód.
- Pri niektorých lexémach vracia analyzátor len kód pridelený pre danú lexému (operátory, oddeľovače, kľúčové slová).
- Pri identifikátoroch musí analyzátor vrátiť okrem kódu lexémy aj názov identifikátora (napr. meno premennej), inak by prišlo k strate informácie!
- Pri konštantách musí analyzátor vrátiť nielen kód lexémy a hodnotu konštanty, ale aj jej typ.

- Z hľadiska ďalšieho spracovania programu, t.j. syntaktickej analýzy, predstavujú lexémy **terminálne symboly** gramatiky, popisujúcej syntax daného jazyka.
- T.j. namiesto toho, aby gramatika popisovala, **čo všetko** môže byť idenfitikátor, obsahuje len terminálny symbol *identifikátor*.
- To, aké rôzne reťazce môžu predstavovať identifikátor sa následne popisuje mimo gramatiku, v popise **lexikálnych elementov** - najčastejšie regulárnym výrazom.

Lexikálny analyzátor môže plniť aj iné úlohy:

- eliminuje nepotrebné informácie (komentáre, prázdne riadky, medzery, tabulátory...),
- spracúva direktívy pre kompilátor,
- spolupracuje s ostatnými súčasťami kompilátora pri identifikácii chýb, pretože je jedinou súčasťou kompilátora, ktorá je schopná označiť riadok, kde bola zistená chyba.

Príklad výstupu lexikálneho analyzátora

Zdrojový program:

```
begin
 if i <= 53 then
  i := i + 1;
end
```

Postupnosť lexém:

begin
if
identifikátor i
<=
konštanta_int 53
then
identifikátor i
:=
identifikátor i
+
konštanta_int 1
;
end

Konštrukcia lexikálneho analyzátora

- Ako už bolo viackrát spomínané, lexémy vieme popísať pomocou regulárnych výrazov.
- Preto sa nám teraz zídu skúsenosti z konečných automatov a regulárnych jazykov.
- Ak viem zostrojiť pre každú lexému DKA a zároveň viem, že regulárne jazyky sú uzatvorené na zjednotenie, znamená to, že viem zostrojiť DKA, ktorý bude akceptovať práve všetky prípustné lexémy daného programovacieho jazyka, t.j. lexikálny analyzátor.

Ako na to?

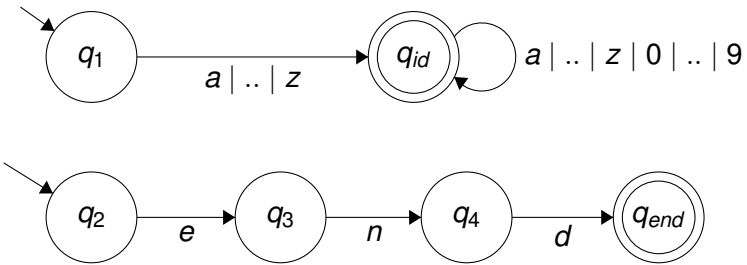
1. Ku každej lexéme zostrojím DKA, ktorý ju rozpoznáva - môžem ho aj minimalizovať vzhľadom na počet stavov. Stavy jednotlivých automatov pre každú lexému musia byť **disjunktné!**.
2. Zostrojím NKA, ktorý bude akceptovať všetky lexémy - všetky DKA z kroku 1. spojím do jedného NKA tak, že vytvorím nový počiatočný stav q_0 a z neho zostrojím ε-prechody do každého počiatočného stavu pôvodných DKA z kroku 1. (t.j. tak, ako sa konštruoval NKA pre zjednotenie regulárnych výrazov).
3. Ku výslednému NKA z kroku 2. zostrojím ekvivalentný DKA.

Príklad

Predpokladajme, že máme 2 typy lexém:

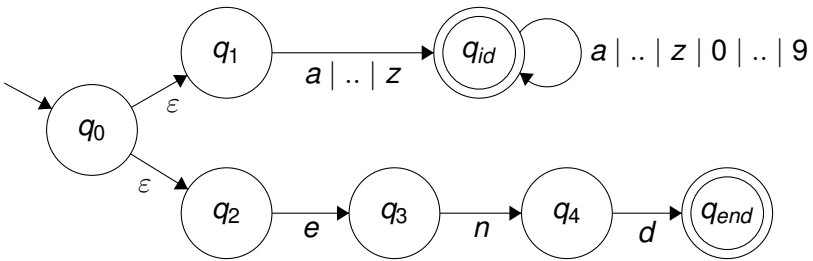
- **identifikátor** - postupnosť malých písmen a číslic; začína písmenom.
- **end** - kľúčové slovo.

1. krok (DKA pre lexémy)



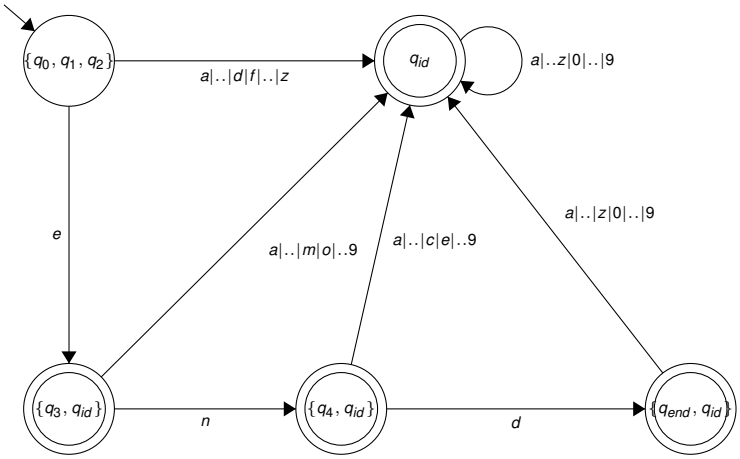
Príklad (pokr.)

2. krok (NKA):



Príklad (pokr.)

3. krok (DKA):



Rozpoznávanie lexém

- Ak lexikálny analyzátor rozpozná lexému, musí vedieť povedať, o akú lexému ide.
- V našom príklade to znamená, že musí vedieť rozpoznať, či prečítal identifikátor alebo kľúčové slovo.
- Ak budú stavy jednotlivých automatov disjunktné, znamená to, že ak lexému rozpozná NKA z 2. kroku, tak na základe výsledného akceptačného stavu je schopný rozpoznať, o ktorú lexému sa jedná (t.j. ktorý pôvodný DKA by ju akceptoval).
- Pri DKA v 3. kroku sa akceptácia lexémy prejaví tak, že automat skončí v niektorom z akceptačných stavov (t.j. v stave, ktorý obsahuje niektorý z pôvodných akceptačných stavov z DKA z 1. kroku).
- Môže nastať situácia, že DKA v 3. kroku obsahuje také stavy, ktoré obsahujú rôzne akceptačné stavy pôvodných automatov (v našom príklade stav $\{q_{end}, q_{id}\}$).

Rozpoznávanie lexém - nejednoznačnosť

- Ak nastane situácia, že DKA v kroku 3. obsahuje stav, ktorý tvoria rôzne akceptačné stavy pôvodných automatov z kroku 1., je potrebné určiť **prioritu**, na základe ktorej sa lexéma klasifikuje.
- Táto priorita sa priradí jednotlivým DKA, ktoré rozpoznávajú jednotlivé lexémy.
- V prípade konfliktu potom automat rozpozná tú lexému, ktorej akceptujúci stav patrí do automatu s najvyššou prioritou.

Rozpoznávanie lexém - nejednoznačnosť

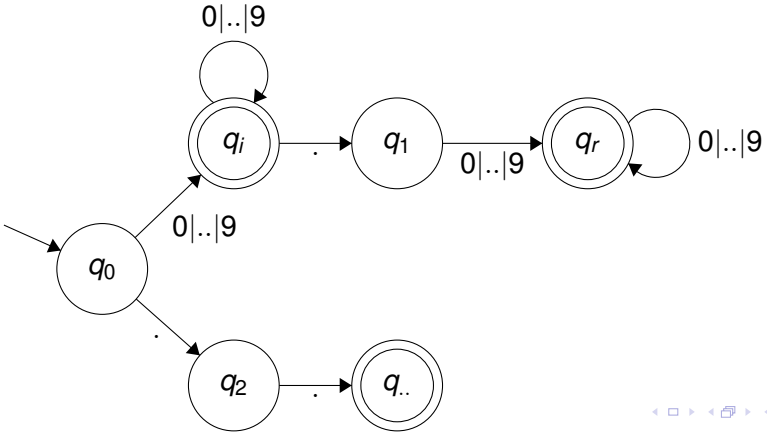
- V našom príklade je takou lexémou reťazec **end**. Takýto reťazec akceptuje aj 1. DKA, aj 2. DKA (pretože takýto reťazec je ak kľúčovým slovom, ale zodpovedá aj identifikátoru).
- Ak dáme kľúčovému slovu **end** prioritu, potom sa lexéma **end** rozpozná ako kľúčové slovo.

Rozpoznávanie lexém - činnosť automatu

1. Automat (analyzátor) číta znaky zo vstupu, kým sa nezasekne. Zároveň využíva vyrovnávaciu pamäť (buffer), do ktorej ukladá všetky prečítané symboly.
2. Ak sa zasekne v akceptačnom stave, vráti kód príslušnej lexémy (podľa príslušného akceptačného stavu). Zároveň sa vo vyrovnávacej pamäti nachádza samotná lexéma, resp. jej reprezentácia (napr. pri identifikátore by to bol názov identifikátora a meno premennej/funkcie/...).
3. Ak sa zasekne v stave, ktorý nie je akceptačný, hľadá najbližší predchádzajúci akceptačný stav. Ak taký neexistuje, nastala **lexikálna chyba**. Ak existuje, automat vráti príslušný kód a lexému (využije pri tom vyrovnávaciu pamäť, pretože príslušná lexéma je **prefixom** toho, čo je v bufferi).
4. Ďalšie spracovanie začína tam, kde skončila posledná lexéma (t.j. v prvom symbole buffera, ktorý nebol súčasťou poslednej identifikovanej lexémy).

Príklad

Uvažujme lexikálny analyzátor pre lexémy **konštanta_int**, **konštanta_real** a operátor **..** (2 bodky). q_i je akceptačný stav pre celé číslo, q_r pre reálne číslo a $q_{..}$ pre operátor **..**



Príklad (pokr.)

Spracovanie vstupu: 10.3

Symbol v bufferi	ε	1	10	10.	10.3
Stav	q_0	q_i	q_i	q_1	q_r

T.j. analyzátor rozpoznal reťazec ako lexému typu **konštanta_real**.

Príklad (pokr.)

Spracovanie vstupu: 10..20

Symbol v bufferi	ε	1	10	10.
Stav	q_0	q_i	q_i	q_1 (automat sa zasekol)

Najbližší akceptačný stav bol q_i pre reťazec 10. Analyzátor teda rozpozná reťazec 10 ako lexému typu **konštanta_int** a prečítanú bodku musíme znovu spracovať:

Symbol v bufferi	ε	.	..
Stav	q_0	q_2	$q_{..}$ (automat sa zasekol)

Automat sa znovu zasekne. Najbližší akceptačný stav bol $q_{..}$ pre reťazec **..** Analyzátor rozpozná tento reťazec ako lexému typu **operátor ..** a pokračuje s ďalšími neprečítanými symbolmi. Tentokrát v bufferi nič nezostalo.

Príklad (pokr.)

Spracovanie zvyšku vstupu:

Symbol v bufferi	ε	2	20
Stav	q_0	q_i	q_i

Reťazec 20 je teda rozpoznaný ako lexéma typu **konštanta_int**. Vstup 10..20 teda analyzátor rozpoznal ako trojicu lexém: **konštanta_int .. konštanta_int**.

Syntaktický analyzátor by následne rozhodol, či takáto trojica terminálnych symbolov je odvoditeľná v gramatike, popisujúcej syntax jazyka.

Flex

- Nástroj Flex (Fast Lexical Analyzer) je program pre generovanie lexikálneho analyzátora.
- `http://gnuwin32.sourceforge.net/packages/flex.htm`
- Pre UNIX systémy existuje balík **`flex`**
- Vstupný súbor (prípona **.l**) obsahuje popis lexém pomocou regulárnych výrazov, spolu s fragmentami kódu v jazyku C, ktorý sa má vykonať, ak bola na vstupe identifikovaná príslušná lexéma.
- Výstupom nástroja je modul lexikálneho analyzátora v jazyku C, štandardne je to súbor **lex.yy.c**. Tento súbor je možné skompilovať a zlinkovať s ďalšími modulmi jazykového procesora (syntaktický analyzátor, sémantické podprogramy, atď.)
- Štandardne sa používa s súčinnosti s nástrojom **Bison**, ktorý predstavuje generátor syntaktického analyzátora (parsera).

Flex

Pre popis lexém využíva Flex regulárne výrazy, pričom sa využívajú rôzne operátory, napr.:

- `[abcz]` - to isté ako `(a | b | c | z)`, t.j. ľubovoľný znak z množiny
- `[a-zA-Z]` - to isté ako `(a|b|..| z | A | B |..| Z)`
- `[^xy]` - ľubovoľný jeden znak okrem `x,y]`
- `r+` - pozitívna iterácia, `r*` - iterácia, `r?` - žiaden alebo jeden výskyt `r`, kde `r` je ľubovoľný regulárny výraz,
- `.` je ľubovoľný znak (bajt) okrem prechodu na nový riadok,
- špeciálne znaky používajú formu známu z jazyku C: `\n` - nový riadok, `\t` - tabulátor, `\"` - úvodzovky

Flex

- `[Bb][Ee][Gg][Ii][Nn]` - popisuje slovo **begin**, pričom nerozlišuje veľké a malé písmeno
- `[a-zA-Z]([a-zA-Z] | [0-9])*` - popisuje identifikátory, t.j. reťazce písmen a čísiel, začínajúce písmenom
- `[+-]?[0-9]+` - popisuje čísla s prípadným znamienkom, môžu začínať viacerými nulami
- `[+-]?[0-9]+[.][0-9]+([Ee][+-]?[0-9]+)?` - desatinné číslo, prípadne v semilogaritmickom tvare
- `\"([^\"\n] | \"\")* \"` - reťazcová konštanta - v jej vnútri sa nesmie nachádzať znak nového reťazca a ak sú tam úvodzovky, tak potom len 2 za sebou

Syntax zdrojového súboru pre Flex

3 časti, oddelené `%%`

1. Definície
2. Pravidlá
3. C-kód používateľa

Syntax zdrojového súboru pre Flex - definície

Prvá časť vstupu obsahuje:

- Programový kód ohraničený `%{ ... }%`, ktorý sa vloží (skopíruje) do výstupného súboru. Môže obsahovať:
  - Definície konštánt predstavujúcich kódy lexém.
  - Deklaráciu používateľom vytvorených premenných a funkcií, používaných v analyzátore.
- Makrá reprezentujúce regulárne výrazy.

Syntax zdrojového súboru pre Flex - pravidlá

Druhá časť vstupu obsahuje dvojice:

regulárny výraz *programový kód*

- *regulárny výraz* predstavuje popis lexémy pomocou regulárneho výrazu
- *programový kód* je program, ktorý sa vykoná, ak bola na vstupe identifikovaná príslušná lexéma
- Reťazec, ktorý bol identifikovaný ako lexéma sa nachádza v premennej **`yytext`**
- Jeho dĺžka v premennej **`yyleng`**
- Ak vstupnému reťazcu zodpovedá viacero lexém, rozpozná sa tá, ktorá je definovaná skorej.

Syntax zdrojového súboru pre Flex - C-kód používateľa

- Tretia časť vstupu obsahuje kód v jazyku C napísaný používateľom, ktorý sa pridá do výstupného súboru.
- Napr. si používateľ môže naprogramovať vlastné funkcie, ktoré sú využívané v programových kódoch v časti 2 vstupného súboru.
- Funkcia samotného lexikálneho analyzátora, ktorý Flex vytvorí, má názov **`yylex()`**. Jej zavolanie spôsobí vyžiadanie nasledujúcej lexémy na vstupe, jej rozpoznanie a vykonanie príslušného kódu zo sekcie 2. Ak sa na vstupe nenachádza žiadna ďalšia lexéma, **`yylex()`** vráti hodnotu 0. Jedno zavolanie **`yylex()`** rozpoznáva lexémy a spúšťa im príslušné kódy dovtedy, kým nenarazí na koniec vstupu alebo na príkaz **`return`**. Potom, ak je to potrebné, je možné **`yylex()`** volať znovu.

Výsledný súbor Flex-u

- Výsledným výstupným súborom je súbor **`lex.yy.c`**, ktorý obsahuje C-implementáciu príslušného lexikálneho analyzátora.
- Jeho skompilovaním dostávame program, ktorý je schopný rozpoznávať lexémy na vstupe a klasifikovať ich podľa príslušných regulárnych výrazov.

Príklad - lexikálny analyzátor vo Flexe č.1

Ako by vyzeral vstupný súbor pre lexikálny analyzátor ku príkladu zo slajdu 29:

Predpokladajme, že máme 2 typy lexém:

- **identifikátor** - postupnosť malých písmen a číslic; začína písmenom.
- **end** - kľúčové slovo.

Príklad - lexikálny analyzátor vo Flexe č.1

Vytvoríme vstupný súbor **`jazyk.l`**. Jeho prvá časť bude obsahovať:

```
%{
#define END 1
#define IDENT 2
#define OSTATNE 3
%}

Cislica [0-9]
Pismeno [a-z]
%%
```

Druhá časť:

```
[ \t\r\n]+ { /*ignoruj*/ }
[Ee][Nn][Dd] {return (END);}
{Pismeno}({Pismeno}|{Cislica})* {return (IDENT);}
. {return (OSTATNE);}
%%
```

Tretia časť:

```
void main()
{
    int c;
    printf("Kod lexemy\tText\t\tDlzka\n");
    while((c = yylex()) > 0)
    {
        printf("%d\t\t%s\t\t%d\n",c,yytext,yyleng);
    }
}
```

- Spustíme program flex so vstupným súborom popísaným na predchádzajúcich 3 slajdoch : **`flex jazyk.l`**
- Flex vygeneruje C-verziu lexikálneho analyzátora: **`lex.yy.c`**
- Skompilujeme, napr. na pre UNIX systémy **`gcc -o la lex.yy.c -lfl`**
- Vo Windowse je zväčša potrebné doplniť na prvý riadok vstupného súboru do programu Flex riadok: **`%option noyywrap`**
- Výsledný program číta zo štandardného vstupu. Ak máme vstup v súbore **`vstup.txt`**, potom si môžeme činnosť analyzátora overiť: **`la < vstup.txt`**

Súbor vstup.txt

```
abeceda
i1 i2 i3
end
start
begin end 2i
```

Výstup z programu **la**

Kod lexemy	Text	Dlzka
2	abeceda	7
2	i1	2
2	i2	2
2	i3	2
1	end	3
2	start	5
2	begin	5
1	end	3
3	2	1
2	i	1

Príklad - lexikálny analyzátor vo Flexe č.2

Uvažujme lexikálny analyzátor pre lexémy **konštanta_int**, **konštanta_real** a operátor **..** (2 bodky). Lexémy sú definované tak, ako na slajde 36.

Prvá časť:

```
%{
#define KONST 1
#define DOTDOT 2
#define OSTATNE 3

#define KONSTINT 1
#define KONSTREAL 2

int typ;
%}

Cislica [0-9]
Cislice {Cislica}+
%%
```

Druhá časť:

```
[ \t\r\n] { /* ignoruj */ }
{Cislice} {typ = KONSTINT; return (KONST);}
{Cislice}[\.]{Cislice} {typ = KONSTREAL;
                        return (KONST);}
\.\. {typ = 0; return (DOTDOT);}
. {return (OSTATNE);}
%%
```

Tretia časť:

```
void main()
{
    int c;
    printf("Kod lexemy\tKod typu\tText\n");
    while((c = yylex()) > 0)
    {
        printf("%d\t\t%d\t\t%s\n",c,typ,yytext);
    }
}
```

Súbor **vstup.txt**

```
10.3
10..20
10..3.10
.10
f1e
```

Výstup z programu **la**

Kod lexemy	Kod typu	Text
1	2	10.3
1	1	10
2	0	..
1	1	20
1	1	10
2	0	..
1	2	3.10
3	2	.
1	1	10
3	1	f
1	1	1
3	1	e

Použitá literatúra

Aho, A., Lam, M., Sethi, R., Ullman, J.: *Compilers: Principles, techniques and tools.*

Dedera, Ľ: *Počítačové jazyky a ich spracovanie*.

Linz, P.: *An Introduction to Formal Languages and Automata*.