

# Przetwarzanie tekstu 7

## Analiza leksykalna

# Analiza leksykalna

- ▶ analiza leksykalna polega na identyfikacji w tekście jednostek, które z punktu widzenia struktury i/lub znaczenia tego tekstu są jednostkami elementarnymi (niepodzielnymi); nazywamy je **jednostkami leksykalnymi**
- ▶ przykład

a 1 f a : = ( b 2 - 16 ) / 1 . 2



analiza leksykalna

alfa := ( b2 - 16 ) / 1.2

# Analiza leksykalna

## jednostki leksykalne

we fragmencie tekstu programu

```
if q=1 then ret=0.5 end
```

rozpoznamy następujące jednostki leksykalne

if	słowo kluczowe 'if'
q	identyfikator
=	operator porównania
1	stała całkowita
then	słowo kluczowe 'then'
ret	identyfikator
=	operator przypisania
0.5	stała zmiennopozycyjna
end	słowo kluczowe 'end'

# Analiza leksykalna

## lex

- ▶ `lex` – typowe, powszechnie używane od ponad 35 lat (M. E. Lesk and E. Schmidt, *Lex - A Lexical Analyzer Generator*, Bell Laboratories, 1975) narzędzie programistyczne do budowania analizatorów leksykalnych (oraz innych programów przetwarzających tekst)
- ▶ `flex` – darmowa implementacja `lex`'a na licencji GNU, prawie całkowicie zgodna z oryginałem

# lex

## zasada działania

- ▶ program `lex (flex)` kompiluje specyfikację analizatora napisaną w języku `lex` na kod definicji funkcji w języku C
- ▶ wynikowa funkcja nazywa się standardowo `yylex( )`
- ▶ funkcja `yylex( )` czyta tekst z wejścia (domyślnie używane jest standardowe wejście) i po rozpoznaniu jednostki leksykalnej wykonuje odpowiednią **akcję**

# lex

## budowa programu

- ▶ program w języku lex składa się z trzech sekcji oddzielonych liniami zawierającymi separator %%

*sekcja deklaracji i definicji*

%%

*sekcja reguł*

%%

*sekcja podprogramów użytkownika*

- ▶ sekcja podprogramów może nie wystąpić, wtedy można pominąć też poprzedzający ją separator %%
- ▶ sekcja deklaracji może być pusta

- ▶ reguła ma postać

wzorzec    akcja

- ▶ wzorzec musi rozpoczynać się od pierwszego znaku linii i nie może zawierać odstępów
- ▶ akcja musi rozpoczynać się w tej samej linii
- ▶ pomiędzy wzorcem i akcją musi być ciąg odstępów
- ▶ wzorzec jest (w najprostszej postaci) wyrażeniem regularnym opisującym budowę jednostki leksykalnej
- ▶ akcja jest ciągiem instrukcji/wywołań makr w języku C
- ▶ jeśli kod akcji zajmuje więcej niż jedną linię, musi być ujęty w parę nawiasów {    }

# lex

## przykład 1

- ▶ program lexowy

przykład1.l

```
%%  
[A-Za-z]* printf("slovo");  
[0-9]*    printf("liczba");
```

- ▶ przetworzy tekst w następujący sposób:

```
Ala ma 3 koty.  
Ola ma 44 psy.
```



```
slovo slovo liczba slovo.  
slovo slovo liczba slovo.
```



# lex

## działanie funkcji `yylex()`

- ▶ cyklicznie, aż do napotkania końca pliku lub instrukcji `return`, wybierana jest jedna spośród reguł przetwarzania, których wzorzec da się dopasować do niepustego przedrostka pliku wejściowego, wykonywana jest akcja i przesuwany wskaźnik pliku wejściowego bezpośrednio za dopasowany przedrostek
- ▶ jeżeli kilka wzorców daje się dopasować, wybierany jest ten, który da się dopasować do najdłuższego przedrostka (**zasada najdłuższego dopasowania**)
- ▶ jeżeli kilka wzorców daje się dopasować do najdłuższego przedrostka, wybierany jest ten, który w pliku programu występuje jako pierwszy (**zasada pierwszego dopasowania**)
- ▶ jeżeli żaden wzorzec nie daje się dopasować, wykonywana jest **akcja domyślna**, polegająca na przepisaniu do pliku wyjściowego jednego znaku i przesunięciu wskaźnika pliku za ten znak

# lex

wartość funkcji `yylex()`

- ▶ funkcja `yylex()` zwraca wartość całkowitą
- ▶ po wyczerpaniu danych wejściowych zwraca 0
- ▶ w typowym zastosowaniu funkcji `yylex()` – jako analizatora leksykalnego stanowiącego komponent np. kompilatora – wartość zwracana określa typ wykrytej jednostki leksykalnej
- ▶ *w przykładach do tego wykładu nie wykorzystujemy wartości zwracanej, ponieważ używamy `lex`a samodzielnie*

# lex

program z funkcją `yylex()`

- ▶ aby móc wykorzystać funkcję `yylex()` w programie, konieczne jest zdefiniowanie jeszcze co najmniej dwóch funkcji:

`main()` oczywiste

`yywrap()` wywoływana przez funkcję `yylex()` w momencie napotkania końca pliku wejściowego

- ▶ najprostsza funkcja `main()`

```
int main()
{
    yylex();
    return 0;
}
```

- ▶ najprostsza funkcja `yywrap()`

```
int yywrap()
{
    return 1;
}
```

# lex

program z funkcją `yylex()` c.d.

- ▶ definicje funkcji `main()` i `yywrap()` można umieścić w sekcji podprogramów użytkownika

```
...  
  
%%  
  
...  
  
%%  
  
int main()  
{  
    yylex();  
    return 0;  
}  
  
int yywrap()  
{  
    return 1;  
}
```

# lex

program z funkcją `yylex()` c.d.

- ▶ definicje funkcji `main()` i `yywrap()` można też pobrać z biblioteki (w przypadku `flexa`, kompilacja z parametrem `-lf1`)

# lex

## przykład 2

- ▶ program

przyklad2.l

```
%%  
a      printf("1");  
aa     printf("2");  
a*     printf("3");
```

- ▶ przetworzy tekst w następujący sposób:

a aa aaa aaaa



1 2 3 3

# lex

## przykład 3

- ▶ program (te same reguły, zmieniona kolejność)

przykład3.pl

```
%%  
a*    printf("1");  
a      printf("2");  
aa     printf("3");
```

- ▶ przetworzy tekst w następujący sposób:

a aa aaa aaaa



1 1 1 1

# lex

zmienne, makra, funkcje

## predefiniowane zmienne

char*	<b>yytext</b>	przedrostek dopasowany do wzorca
int	<b>yylen</b>	długość dopasowania
FILE*	<b>yyin</b>	wskaźnik do aktualnego pliku wejściowego
FILE*	<b>yyout</b>	wskaźnik do aktualnego pliku wyjściowego



# lex

zmienne, makra, funkcje

## predefiniowane makra i funkcje

<b>ECHO</b>	wypisz wartość zmiennej <code>yytext</code> na wyjście
<b>REJECT</b>	przejdź do wykonywania 'kolejnej najlepszej' reguły
<b>yyomore( )</b>	w kolejnym dopasowany przedrostek dopisz do aktualnej wartości zmiennej
<b>yyless(<i>n</i>)</b>	prześlij zwrótnie do pliku wejściowego wszystkie poza <i>n</i> początkowymi znakami zapisanymi w zmiennej <code>yytext</code>
<b>BEGIN <i>s</i></b>	przejdź do stanu <i>s</i>
<b>input( )</b>	pobierz kolejny znak z pliku wejściowego
<b>unput(<i>c</i>)</b>	zwróć znak <i>c</i> do pliku wejściowego

# lex

## wstawianie kodu w języku C

```
%{  
                                     (a)  
%}  
  
%%  
                                     (b)  
    reguły  
%%  
    (c)
```

- (a) kod zawarty między ogranicznikami `%{` i `%}` (muszą wystąpić same w linii) w sekcji deklaracji i definicji zostaje skopiowany na początek pliku wynikowego, przed definicję funkcji `yylex()`
- (b) wcięty kod na początku sekcji reguł jest kopiowany na początek ciała funkcji `yylex()` (tylko `flex`)
- (c) cały kod z sekcji podprogramów użytkownika jest kopiowany na koniec pliku wynikowego

# lex

## operator prawego kontekstu (podgląd)

- ▶ wzorzec może mieć postać:

$r/s$

gdzie  $r$  i  $s$  są wyrażeniami regularnymi

- ▶ wtedy przedrostek pliku wejściowego musi zostać dopasowany do wyrażenia  $rs$ , przy czym wartością `yytext` staje się tylko początkowa jego część dopasowana do  $r$
- ▶ wzorzec taki możemy czytać:  $r$  pod warunkiem, że występuje po nim  $s$
- ▶ w algorytmie wyboru reguły brana jest pod uwagę długość całego fragmentu dopasowanego do  $rs$

# lex

## operator prawego kontekstu – przykład

- ▶ program

```
%{  
    #include <stdio.h>  
}%  
%%  
[\\t\\ ]*                putchar(' ');  
[\\t\\ ]*/[,\\.]  
;
```

- ▶ przetworzy tekst w następujący sposób

Człowiek , o którego pytales , wyjechał .



Człowiek, o którego pytales, wyjechał.

# lex

## stany (uwzględnienie lewego kontekstu)

- ▶ zestaw reguł, które mają być w danym momencie uwzględnione, można uzależnić od historii przetwarzania (lewego kontekstu); wykorzystujemy do tego mechanizm stanów
- ▶ predefiniowany stan, w którym **lex** zaczyna pracę to stan 0 (we **flexie**: **INITIAL**)
- ▶ deklaracje dodatkowych stanów umieszcza się w sekcji deklaracji i definicji, ma ona postać:

```
%start stan stan ... stan (lex)
```

```
%s stan stan ... stan (flex)
```

# lex

## stany (uwzględnienie lewego kontekstu)

- ▶ reguła o wzorcu postaci:

$w$  jest aktywna zawsze

$< * > w$  jest aktywna zawsze

$< stan_i, stan_j, \dots > w$  jest aktywna tylko gdy aktualnym stanem jest jeden z wymienionych

- ▶ przejście do stanu  $s$  następuje wskutek wykonania makropolecenia **BEGIN**(  $s$  ).
- ▶ we **flexie** można definiować też stany tzw. exclusive (deklaracja `%x stan`); w tych stanach aktywne są tylko reguły, w których wzorcach stan ten jawnie jest wymieniony lub użyto wyrażenia `< * >`.

# lex

## stany – przykład

- poniższy program w liniach rozpoczynających się wielką literą zamienia wszystkie litery na wielkie, w liniach rozpoczynających się małą literą zamienia wszystkie litery na małe, a linie nie rozpoczynające się literą pozostawia bez zmian (działa tu reguła domyślna)

zamianaliter.l

```
%{
    #include <ctype.h>
    #include <stdio.h>
}%

%s MALE WIELKIE

%%

<INITIAL>^[a-z]      ECHO; BEGIN(MALE);
<MALE>[A-Z]          putchar(tolower(*yytext));
<INITIAL>^[A-Z]      ECHO; BEGIN(WIELKIE);
<WIELKIE>[a-z]       putchar(toupper(*yytext));
\n                   ECHO; BEGIN(INITIAL);
```

# lex

## funkcja `yywrap()`

- ▶ funkcja `wrap()` jest wywoływana automatycznie w momencie osiągnięcia końca pliku wejściowego; umożliwia ona określenie następnego pliku wejściowego (np. przez przypisanie wartości zmiennej `yyin`)
- ▶ jeśli to nastąpi, wartością zwracaną powinno być 0
- ▶ gdy wartość jest niezerowa, `lex` domniemuje, że nowy plik wejściowy nie został określony i kończy pracę.

```
%%  
    ...  
%%  
  
int yywrap()  
{  
    yyin = nastepny_plik;  
    return 0;  
}
```



# lex

## funkcja yywrap( )

- ▶ funkcję yywrap( ) można też wykorzystać wtedy, gdy chcemy wykonać pewne działania po zakończeniu przetwarzania pliku
- ▶ poniższy program drukuje na wyjściu sumę wszystkich liczb znalezionych w pliku wejściowym

sumaliczb.l

```
%{  
    #include <stdio.h>  
  
    float suma=0;  
}%  
  
%%  
  
-?[0-9]+(\\.[0-9]+)?      suma += atof(yytext);  
.|\\n                    ;  
  
%%  
  
int yywrap() { printf("%.2f\\n", suma); }
```

# lex

definiowanie nazw dla wyrażeń regularnych

przyklad1\_inaczej.l

```
DIGIT [0-9]
LETTER [a-zA-Z]

%%

{DIGIT}* printf("liczba");
{LETTER}* printf("slovo");
```

# lex

## wzorzec końca pliku

- ▶ `<<EOF>>` to wzorzec końca pliku, może wystąpić samodzielnie lub poprzedzony stanami
- ▶ program z przykładu [sumaliczb.l](#) możemy sformułować też:

[sumaliczb\\_inaczej.l](#)

```
%{
    #include <stdio.h>

    float suma=0;
}%

%%

-?[0-9]+(\\.[0-9]+)?      suma += atof(yytext);
.|\\n                      ;
<<EOF>>                   {
                           printf("%.2f\\n",suma);
                           return 0;
                           }
```

- ▶ cudzysłów (podwójny) pełni rolę ogranicznika ciągu znaków, znoszącego specjalne znaczenie wszystkich znaków specjalnych poza " i \
- ▶ wzorzec

```
"[a-z]* \"+"
```

dopasuje się do ciągu znaków

```
[a-z]* "+
```

# Analiza leksykalna

## literatura

- ▶ A. V. Aho, R. Sethi, J. D. Ullman, *Kompilatory. Reguły, metody i narzędzia*, Wydawnictwo Naukowo-Techniczne, 2002, **(rozdział 3.)**