Metody kompilacji

Wykład 11

Y A C C

Y A C C

Generator analizatorów składniowych

Literatura

- *A Compact Guide to Lex & Yacc*
  by Tom Niemann
  http://epaperpress.com/lexandyacc/

- *The Lex & Yacc Page*
  http://dinosaur.compilertools.net/

Wstęp

Autor oryginalnej wersji YACC'a:

Stephen C. Johnson, 1975.

Narzędzia pokrewne:

– lex, yacc (AT&T)

– bison (GNU)

– BSD yacc

– PCYACC (Abraxas Software)

GENERATOR YACC

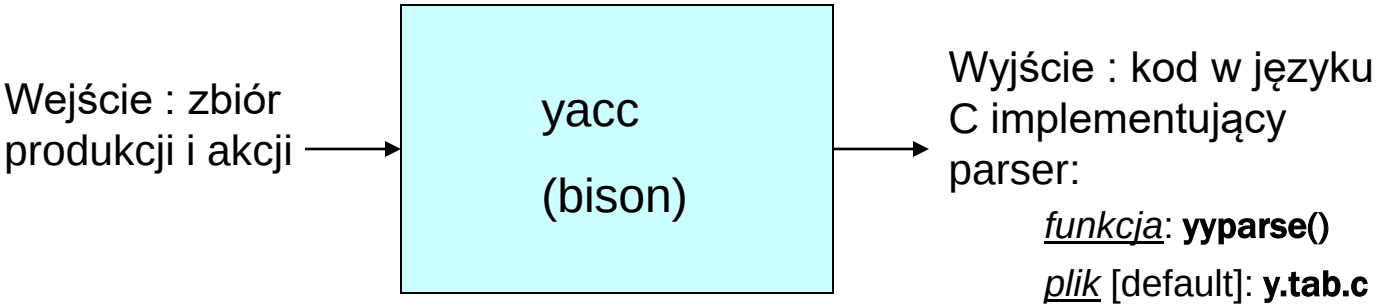
- Zadaniem generatora YACC jest wygenerowanie kodu źródłowego analizatora składniowego w języku C.

- Kod źródłowy generowany jest przez YACC'a w oparciu o plik ze specyfikacją.

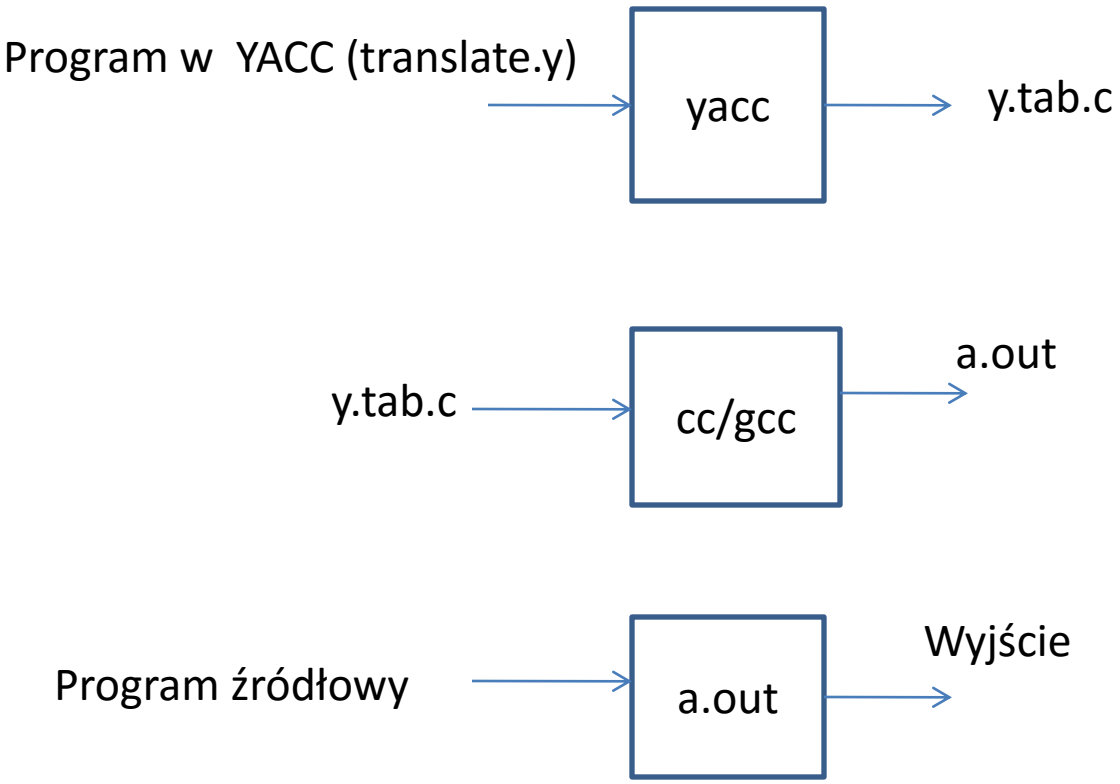
GENERATOR **YACC**

– Pobiera specyfikację gramatyki bezkontekstowej

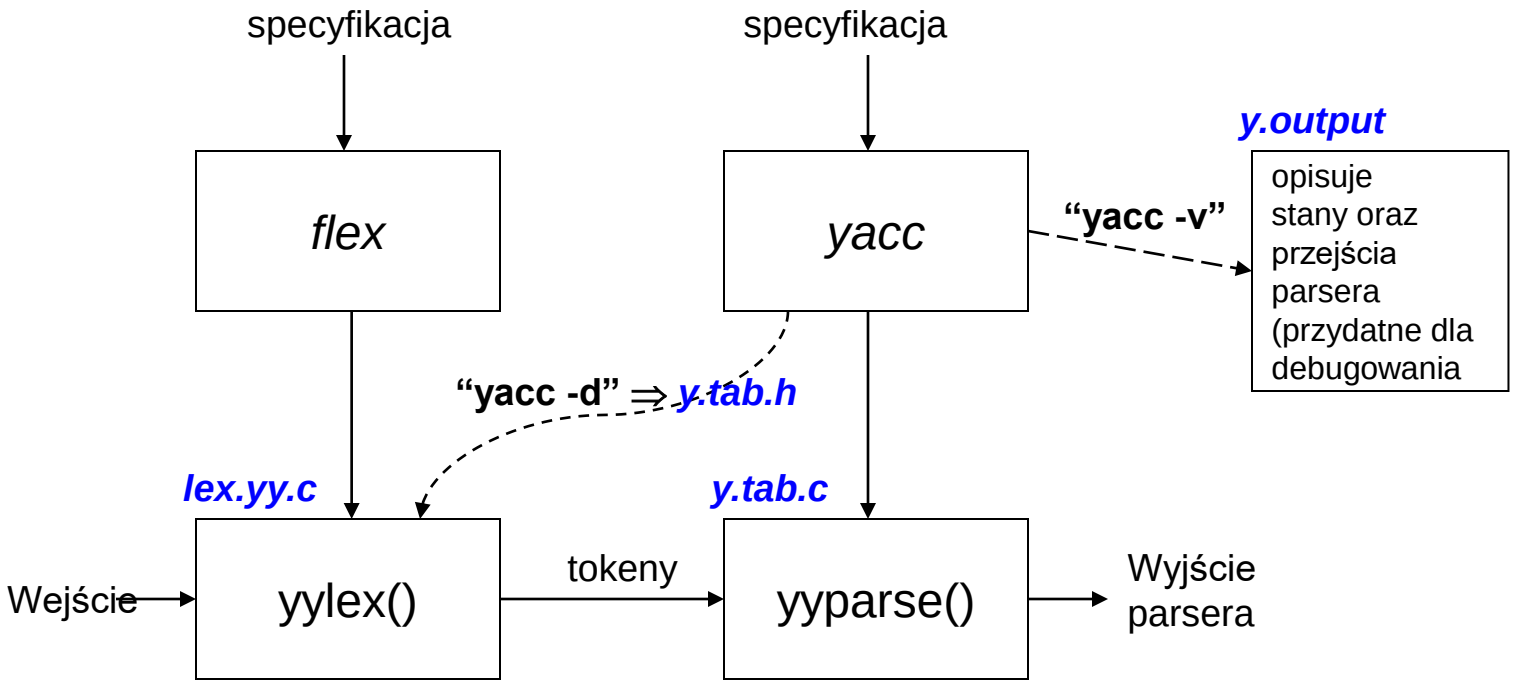
– Generuje kod dla parsera.



Jak korzystamy z YACC'a



Zastosowanie YACCA



GENERATOR **YACC**

Zasady działania programu:

- Wygenerowany przez program YACC analizator redukujący działa w oparciu o tablicę LALR(1).

GENERATOR **YACC**

Zasady działania programu:

- Jako symbol startowy gramatyki, przyjmowany jest, przez domniemanie, nieterminal znajdujący się po lewej stronie pierwszej produkcji.

GENERATOR YACC

- Wygenerowany parser ma postać funkcji

  int yyparse(). Aby uruchomić parser zawarty w tej funkcji potrzebujemy dwóch innych funkcji: main() i yylex().

- Funkcja main() wywołuje funkcję yyparse().

GENERATOR **YACC**

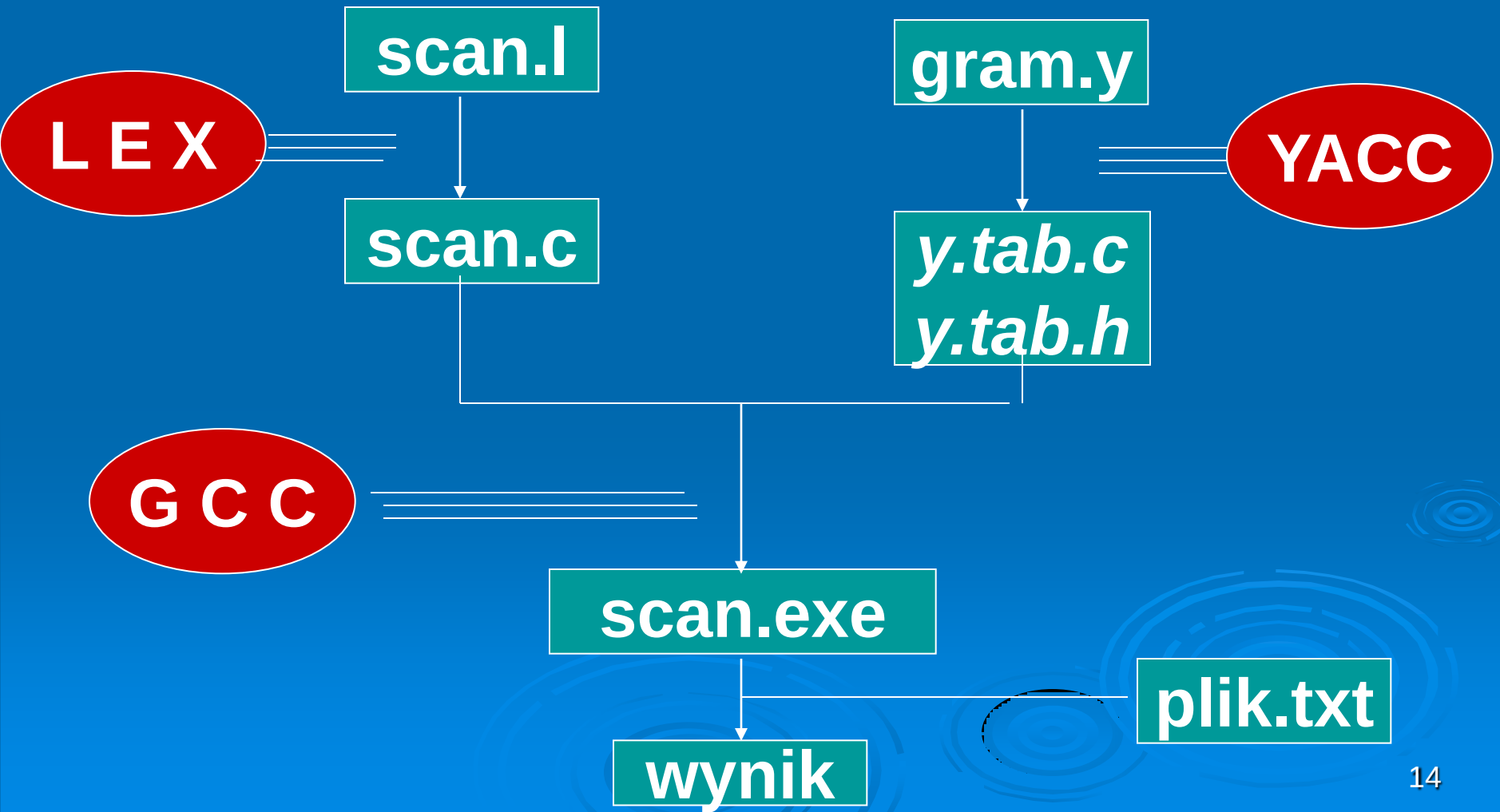
- Parser wywołuje funkcję yylex() celem pobrania tokena z wejścia.
- Yacc definiuje nazwy tokenów w parserze jako nazwy preprocesora C w pliku y.tab.h, więc yylex() może je użyć.

GENERATOR **YACC**

- Gdy yylex() znajdzie token, zwraca do parsera jego numeryczną wartość, umieszczając ją w zmiennej yyval;

GENERATOR **YACC**

Schemat organizacji działania Yacc'a:



TWORZENIE PLIKU SPECYFIKACJI

- Każdy plik ze specyfikacją dla programu YACC powinien składać się z trzech sekcji.

- Pierwsza sekcja to **sekcja definicji**,

  gdzie umieszczamy, definicje i deklaracje zmiennych, stałych, deklaracje stanów oraz makra preprocesora.

TWORZENIE PLIKU SPECYFIKACJI

- Sekcja definicji może zawierać fragment kodu, który zostanie uwzględniony przez analizator składniowy.

- Kod ten musi być odpowiednio „opakowany”: otwarcie fragmentu powinno być poprzedzone znacznikiem **%{**, natomiast jej zamknięcie znacznikiem **%}**;

TWORZENIE PLIKU SPECYFIKACJI

- Przykład budowy sekcji definicji:

```
%{
        #include <iostream.h>
        int zmienna;
        int zmienna_druga=1;
%}
```

TWORZENIE PLIKU SPECYFIKACJI

- Druga sekcja to **sekcja przetwarzania**.

- W sekcji przetwarzania umieszczamy wszelkie reguły przetwarzania, zgodnie z którymi wygenerowany będzie analizator.

TWORZENIE PLIKU REGUŁ

- Budowa reguły przetwarzania opiera się na dwóch zasadniczych częściach: produkcji i operacji.

- Produkcja jest zapisana w notacji programu YACC. Strzałka w produkcji jest zastąpiona znakiem „ : ”.
- Kolejne ciała produkcji, których lewa strona jest taka sama, oddzielamy znakiem '|‘.
- Operacja jest blokiem instrukcji języka C.

TWORZENIE PLIKU SPECYFIKACJI

Jeśli w gramatyce mamy produkcję

S->T+T,

to reguła produkcji może wyglądać następująco:

```
S  :  T  '+'  T        {printf("liczba + liczba");}
   ;
```

NOTACJA YACC'a

- Rozważmy gramatykę, w której zbiór produkcji jest następujący:

  E -> E + T;

  E -> T;

  T -> T * F;

  T -> F;

  F -> (E);

  F -> num;

Produkcje w notacji YACC'a:

```
%token num
%%
E : E '+' T
  | T
  ;
T : T '*' F
  | F
  ;
F : '(' E ')'
  | num
  ;
```

E->E+T
E->T
T->T*F
T->F
F->(E)
F->num

NOTACJA YACC'a

- Niech G będzie gramatyką z produkcjami T->(T) i T->ε. Wtedy ε-produkcję możemy zapisać w notacji YACC'a następująco:

```
%%
T  :  '(' T ')'
      |
      ;
```

AKCJE SEMANTYCZNE

- Akcja semantyczna YACC'a jest sekwencją instrukcji w C.
- Symbol $$ odwołuje się do wartości atrybutu skojarzonej z nieterminalem po lewej stronie.

AKCJE SEMANTYCZNE

- Symbol $i odwołuje się do wartości skojarzonej z i-tym symbolem gramatyki po prawej stronie.
- Akcja semantyczna wywoływana jest zawsze, gdy redukujemy według związanej z nią produkcji.

AKCJE SEMANTYCZNE

Rozważmy dwie produkcje: E->E+T | T;

Notacja YACC'a:

```
wyr : wyr '+' term  {$$ = $1 + $3;}
                | term
                ;
```

Przykład

```
%{
#include<ctype.h>
%}
%token DIGIT
%%

line:expr '\n'          {printf("%D\n",$1);}
            ;
Expr        : expr '+' term         {$$=$1+$3}
            | term
            ;
Term        : term '*' factor        {$$=$1*$3}
            | factor
            ;
Factor      : '(' expr ')'          {$$=$2}
            | DIGIT
            ;
%%
```

Przykład

```
%{
#include< ctype.h>
%}
%token DIGIT
%%

line  :expr '\n'     {printf("%d\n",$1);}
                ;
expr : expr '+' term          {$$=$1+$3}
                | term
                ;
term : term '*' factor         {$$=$1*$3}
                | factor
                ;
factor          : '(' expr ')'                  {$$=$2}
                | DIGIT
                ;
%%
```

```
yylex()
{
        int c;

c=getchar();
if isdigit(c )
{
 yylval=c-'0';
return DIGIT;
}
return c;
}
```

Prosty kalkulator

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line    :  expr '\n'          { printf("%d\n", $1); }
        ;
expr    :  expr '+' term      { $$ = $1 + $3; }
        |  term
        ;
term    :  term '*' factor    { $$ = $1 * $3; }
        |  factor
        ;
factor  :  '(' expr ')'       { $$ = $2; }
        | DIGIT
        ;
%%

yylex() { int c; c=getchar();
          if (isdigit(c)) {yylval=c-'0'; return
DIGIT;}
          return c;}
```

Deklaruje isdigit m.in.

Deklaruje token DIGIT

Akcja semantyczna

Domyślna akcja $$ = $1 dla produkcji, których prawa strona zawiera jeden symbol

#include "lex.yy.c" aby korzystać z procedury *yylex utworzonej przez* Lexa

'yylval' przechowuje atrybut

Konflikty

- W celu rozwiązania konfliktu YACC stosuje dwie reguły:

  1. Konflikt reduce/reduce jest rozwiązywany przez wybór produkcji, która jest umiejscowiona wcześniej w specyfikacji YACC'a.

Konflikty

2. Domyślnie, konflikt shift/reduce zostaje zawsze rozwiązany na rzecz shift.

Konflikty

- Ponieważ ta ostatnia zasada nie zawsze może być właściwa, YACC zawiera ogólny mechanizm rozwiązywania konfliktów shift/reduce.

Konflikty

- W deklaracjach, możemy przypisać pierwszeństwo i łączność do terminali.
- Deklaracja

  %left ‘+’ ‘-’

  czyni, że operatory ‘+’ i ‘-’ będą miały takie same pierwszeństwo i będą łączne lewostronnie.

Konflikty

- Możemy zdeklarować operator łączny prawostronnie jak

  %right ‘-’

Konflikty

- Pierwszeństwo terminali i produkcji określa kolejność, w której one się pojawiają w części deklaracyjnej.

Konflikty

Jeśli YACC ma wybrać między przesunięciem symbolu *a* a redukcją w oparciu o produkcję

A -> α,

to wybiera redukcję jeśli pierwszeństwo produkcji jest większe od pierwszeństwa symbolu *a*.

- Inaczej wybiera przesunięcie.

Jeszcze jeden prosty kalkulator - Pliki

- calc.l
  - Zawiera specyfikację dla LEXa.
- calc.y
  - Zawiera specyfikację dla YACCa oraz wywołanie funkcji yylex .

Współpraca między LEX i YACC

scanner.l

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
id          [_a-zA-Z][_a-zA-Z0-9]*
%%
int         { return INT; }
char        { return CHAR; }
float       { return FLOAT; }
{id}        { return ID;}
```

parser.y

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token  CHAR, FLOAT, ID, INT
%%
```

```
yacc -d xxx.y
```

- **Produced**

```
y.tab.h

# define CHAR 258
# define FLOAT 259
# define ID 260
# define INT 261
```

Wywołanie YACCa z flagą „-d” tworzy plik nagłówkowy zawierający liczbowe identyfikatory tokenów

Plik dla YACCa

- %{
- #include <stdio.h>
- int regs[26];
- int base;
- %}
- %start list
- %token DIGIT LETTER
- %left '|'
- %left '&'
- %left '+' '-'
- %left '*' '/' '%'
- %left UMINUS /*supplies precedence for unary minus */

Plik dla YACCa

```
%%                  /* beginning of rules section */
list:                 /*empty */
        |
        list stat '\n'
        |
        list error '\n'
        {
          yyerrok;
        }
        ;
stat:    expr
        {
          printf("%d\n",$1);
        }
        |
        LETTER '=' expr
        {
          regs[$1] = $3;
        }
        ;
```

Akcja semantyczna jest wypisana po produkcji

```
expr:    '(' expr ')'
        {
          $$ = $2;
        }
        |
        expr '*' expr
        {
          $$ = $1 * $3;
        }
        |
        expr '/' expr
        {
          $$ = $1 / $3;
        }
        |
        expr '%' expr
        {
          $$ = $1 % $3;
        }
        |
        expr '+' expr
        {
          $$ = $1 + $3;
        }
        |
        expr '-' expr
        {
          $$ = $1 - $3;
        }
        |
        expr '&' expr
        {
          $$ = $1 & $3;
        }
        |       expr '|' expr       {  $$ = $1 | $3;
        }         |
```

Plik dla YACCa

```
        '-' expr %prec UMINUS
        {
          $$ = -$2;
        }
        |
        LETTER
        {
          $$ = regs[$1];
        }
        |
        number
        ;
number:  DIGIT
        {
          $$ = $1;
          base = ($1==0) ? 8 : 10;
        }       |
        number DIGIT
        {
          $$ = base * $1 + $2;
        }
        ;
```

Plik dla YACCa

```
%%
main()
{
 return(yyparse());
}
yyerror(s)
char *s;
{
  fprintf(stderr, "%s\n",s);
}
yywrap()
{
  return(1);
}
```

Plik dla Lexa

```
%{

#include <stdio.h>
#include "y.tab.h"
int c;
extern int yylval;
%}
%%
" "       ;
[a-z]     {
          c = yytext[0];
          yylval = c - 'a';
          return(LETTER);
          }
[0-9]     {
          c = yytext[0];
          yylval = c - '0';
          return(DIGIT);
          }
[^a-z0-9\b]    {
               c = yytext[0];
               return(c);
               }
```

Kompilacja i wykonanie

- bison -d –y calc.y
  - create y.tab.c and y.tab.h
- flex calc.l
  - create lex.yy.c
- gcc -g lex.yy.c y.tab.c -o calc
  - Create execution file
- ./calc
  - Run the calculator

Dziękuję za uwagę