Podstawy Kompilatorów

Laboratorium 1

Celem laboratorium jest zapoznanie się ze środowiskiem i sprawdzenie poprawności działania narzędzi. Nazwy programów i rezultaty będą takie same w systemie operacyjnym Linux i systemie Cygwin (o ile nie zaznaczono, że jest inaczej).

Zadanie 1:
Testowanie działania kompilatora języka C.

Proszę do dowolnego edytora tekstowego wprowadzić poniższy program:

```
#include <stdio.h>
int main(void)
{
  printf("\nTest\n");
  return 0;
}
```

i zapisać go w pliku *test.c*

Program kompilujemy wydając polecenie:

```
gcc test.c
```

jeżeli kompilacja przebiegnie poprawnie powinniśmy otrzymać program wykonywalny o nazwie:

```
a.out (Cygwin: a.exe)
```

Po wywołaniu:

```
./a.out (Cygwin: ./a.exe)
```

powinniśmy otrzymać rezultat:

```
Test
```

Zadanie 2:
Testowanie działania generatora Lex.

Proszę do dowolnego edytora tekstowego wprowadzić poniższą specyfikację:

```
%{
  #include <stdio.h>
  int yywrap(void);
  int yylex(void);
%}
%%
a      { printf("b"); }
%%
int yywrap(void) { return 1; }
int main(void) { return yylex(); }
```

i zapisać ją w pliku *scan.l*

Analizator leksykalny generujemy wydając polecenie:

```
flex -l scan.l
```

opcja –l przełącza flexa w tryb maksymalnej zgodności z oryginalnym generatorem LEX. W większości dystrybucji Linuxa istnieje symboliczne łącze wywołujące od razu flexa z parametrem –l, wtedy wystarczy wydać polecenie:

```
lex scan.l
```

Jeżeli generacja przebiegnie poprawnie powinniśmy otrzymać plik *lex.yy.c*

Program kompilujemy wydając polecenie:

```
gcc lex.yy.c
```

jeżeli kompilacja przebiegnie poprawnie powinniśmy otrzymać program wykonywalny o nazwie:

```
a.out (Cygwin: a.exe)
```

Proszę utworzyć za pomocą dowolnego edytora tekstowego plik o poniższej zawartości:

```
Ala ma kota
```

i zapisać go w pliku *test.in*

Po wywołaniu:

```
./a.out < test.in (Cygwin: ./a.exe < test.in)
```

powinniśmy otrzymać rezultat:

```
Alb mb kotb
```

Uwaga: Na etapie nauki niewskazane jest uruchamianie programów w trybie interaktywnym. Zachowanie bardziej złożonych analizatorów, ze względu na buforowanie tekstu i działanie podglądu, może się różnić w zależności od tego, czy są uruchomione w trybie interaktywnym czy wsadowym.

Zadanie 3:
Testowanie działania generatora LLgen (Linux).

Proszę do dowolnego edytora tekstowego wprowadzić poniższą specyfikację:

```
%{
  #include <stdio.h>
  int yywrap(void);
  int yylex(void);
  #include "Lpars.h"
%}
%%
a   { return 'a'; }
b   { return 'b'; }
%%
int yywrap(void) { return 1; }
```

i zapisać ją w pliku *scan.l*

Proszę do dowolnego edytora tekstowego wprowadzić poniższą specyfikację:

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

%start parse, spec ;

spec : 'a' spec { printf("1"); }
     | 'b' spec { printf("2"); }
     |          { printf("3"); }
     ;

{
int main(void)
{
  printf("\n");
  parse();
  printf("\n");
  return 0;
}
LLmessage(int tk)
{
  printf("blad (%d)",tk);
}
}
```

i zapisać ją w pliku *gram.g*

Analizator leksykalny generujemy, tak jak w poprzednim zadaniu, wydając polecenie:

```
flex -l scan.l
```

Jeżeli generacja przebiegnie poprawnie powinniśmy otrzymać plik *lex.yy.c*

Analizator składniowy generujemy poleceniem:

```
LLgen gram.g
```

Jeżeli generacja przebiegnie poprawnie powinniśmy otrzymać pliki: *Lpars.c* i *Lpars.h*

Kompletny analizator kompilujemy poleceniem:

```
gcc lex.yy.c Lpars.c gram.c
```

Jeżeli kompilacja przebiegnie poprawnie otrzymamy plik o nazwie:

```
a.out
```

Proszę utworzyć za pomocą dowolnego edytora tekstowego plik o poniższej zawartości:

```
aabb
```

i zapisać go w pliku *test.in*

Po wywołaniu:

```
./a.out < test.in
```

powinniśmy otrzymać rezultat:

```
32211
```

Uwaga: Na etapie nauki niewskazane jest uruchamianie programów w trybie interaktywnym. Zachowanie bardziej złożonych analizatorów, ze względu na buforowanie tekstu i działanie podglądu, może się różnić w zależności od tego, czy są uruchomione w trybie interaktywnym czy wsadowym.

Zadanie 4:
Testowanie działania generatora YACC.

Proszę do dowolnego edytora tekstowego wprowadzić poniższą specyfikację:

```
%{
  int yywrap(void);
  int yylex(void);
  #include "y.tab.h"
%}
%%
a          { return 'a'; }
b          { return 'b'; }
%%
int yywrap(void) { return 1; }
```

i zapisać ją w pliku *scan.l*

Proszę do dowolnego edytora tekstowego wprowadzić poniższą specyfikację:

```
%{
  int yylex(void);
  void yyerror(const char*, ...);
  int yyparse(void);
  #include <stdio.h>
  extern int yylineno;
%}
%%
S : { printf("\n"); }
    L
    { printf("\n"); }
  ;
L : 'a' L { printf("1"); }
  | 'b' L { printf("2"); }
  |       { printf("3"); }
  ;
%%
void yyerror(const char *fmt,...) {
   printf("%s in line %d\n", fmt, yylineno); }
int main(void) { return yyparse(); }
```

i zapisać ją w pliku *gram.y*

Analizator leksykalny generujemy tak, jak w poprzednich zadaniach, wydając polecenie:

```
flex -l scan.l
```

Jeżeli generacja przebiegnie poprawnie powinniśmy otrzymać plik *lex.yy.c*

Analizator składniowy generujemy, w zależności od tego, który generator jest zainstalowany w systemie, poleceniem:

```
bison -y -d gram.y
```

albo

```
byacc -d gram.y
```

w obu przypadkach, jeżeli kompilacja przebiegnie poprawnie, powinniśmy otrzymać pliki *y.tab.c* i *y.tab.h*
Zazwyczaj w dystrybucjach Linuxa istnieje symboliczne łącze wywołujące od razu YACCa z ewentualnymi odpowiednimi parametrami, wtedy wystarczy wydać polecenie:

```
yacc -d gram.y
```

Kompletny analizator kompilujemy poleceniem:

```
gcc y.tab.c lex.yy.c
```

Jeżeli kompilacja przebiegnie poprawnie otrzymamy plik o nazwie:

```
a.out (Cygwin: a.exe)
```

Proszę utworzyć za pomocą dowolnego edytora tekstowego plik o poniższej zawartości:

```
aabb
```

i zapisać go w pliku *test.in*

Po wywołaniu:

```
./a.out < test.in (Cygwin: ./a.exe < test.in)
```

powinniśmy otrzymać rezultat:

```
32211
```

Uwaga: Na etapie nauki niewskazane jest uruchamianie programów w trybie interaktywnym. Zachowanie bardziej złożonych analizatorów, ze względu na buforowanie tekstu i działanie podglądu, może się różnić w zależności od tego, czy są uruchomione w trybie interaktywnym czy wsadowym.