Exercise 1

Specify in Lex the analyzer of the lexical symbols of a language where each phrase is composed of one or more definitions, as in the following example:

```
int x = 3, num = 100;
string A = "alpha", B = "beta";
boolean ok = true, end = false;
```

Exercise 1

```
%{
#include <stdlib.h>
#include “def.h”
Lexval lexval;  /* typedef union {int ival; char *sval; } Lexval; */
%}
delimiter       [ \t\n]
spacing         {delimiter}+
letter          [A-Za-z]
digit           [0-9]
intconst        {digit}+
strconst        \"([^\"])*\”
boolconst       false|true
id              {letter}({letter}|{digit})*
%%
{spacing}       ;
int             {return(INT);}
string          {return(STRING);}
boolean         {return(BOOLEAN);}
{intconst}      {lexval.ival = atoi(yytext); return(INTCONST);}
{strconst}      {lexval.sval = newstring(yytext); return(STRCONST);}
{boolconst}     {lexval.ival = (yytext[0] == 'f' ? 0 : 1); return(BOOLCONST);}
{id}            {lexval.sval = newstring(yytext); return(ID);}
=               {return(ASSIGN};}
,               {return(COMMA);}
;               {return(SEMICOLON);}
.               {return(ERROR);}
%%
char *newstring(char *s)
{
  char *p = malloc(strlen(s)+1);
  strcpy(p,s);
  return(p);
}
```

```
int x = 3, num = 100;
string A = "alpha", B = "beta";
boolean ok = true, end = false;
```

Exercise 2

Specify in Lex the lexical analyzer of the language relevant to the following EBNF:

program → { *stat* **;**}$^{+}$
stat → *def-stat* | *if-stat* | *while-stat* | *assign-stat*
def-stat → **def id :** *type*
type → **integer** | **real** | **character**
if-stat → **if** *cond* **then** *stat* [**else** *stat*]
cond → **id** *relop const*
relop → = | **!=** | > | < | >= | <=
const → **intconst** | **realconst** | **charconst**
while-stat → **while** *cond* **do** *stat*
assign-stat → **id :=** *const*

(example of phrase)

```
def x: integer;
def y: real;
def z: character;
x := 25;
y := 14.358;
z := 'a'
```

Exercise 2

Specify in Lex the lexical analyzer of the language relevant to the following EBNF:

program → { *stat* ;}$^{+}$
stat → *def-stat* | *if-stat* | *while-stat* | *assign-stat*
def-stat → **def id :** *type*
type → **integer** | **real** | **character**
if-stat → **if** *cond* **then** *stat* [**else** *stat*]
cond → **id** *relop const*
relop → = | != | > | < | >= | <=
const → **intconst** | **realconst** | **charconst**
while-stat → **while** *cond* **do** *stat*
assign-stat → **id :=** *const*

(example of phrase)

```
def x: integer;
def y: real;
def z: character;
x := 25;
y := 14.358;
z := 'a'
```

```
%{
#include <stdio.h>
#include "def.h"  /* Token e typedef Lexval */
Lexval lexval;   /* Fields: ival,rval,cval,sval */
%}

delimiter   [ \t\n]
spacing     {delimiter}+
letter      [A-Za-z]
digit       [0-9]
intconst    digit}+
realconst   {intconst}\.{intconst}
charconst   '(.|\n)'
id          {letter}({letter}|{digit}).

%%
```

```
{spacing}  ;
def             {return(DEF);}
if              {return(IF);}
then            {return(THEN);}
else            {return(ELSE);}
integer         {return(INTEGER);}
real            {return(REAL);}
character       {return(CHARACTER);}
while           {return(WHILE);}
do              {return(DO);}
";"             {return(SEMICOLON);}
":"             {return(COLON);}
":="            {return(ASSIGN);}
"="             {return(EQ);}
"!="            {return(NE);}
">"             {return(GT);}
"<"             {return(LT);}
">="            {return(GE);}
"<="            {return(LE);}
{intconst}      {lexval.ival = atoi(yytext);
                 return(INTCONST);}
{realconst}     {lexval.rval = atof(yytext);
                 return(REALCONST);}
{charconst}     {lexval.cval = yytext[1];
                 return(CHARCONST);}
{id}            {lexval.sval = newstring(yytext);
                 return(ID);}
.               {return(ERROR);}

%%
```

```
char *newstring(char *s)
{
  char *p;

  p = malloc(strlen(s)+1);
  strcpy(p, s);
  return(p);
}
```

Exercise 3

Specify in Lex the lexical analyzer of the language defined by the following EBNF:

program → { *stat* **;** }$^{+}$
stat → *def-stat* | *assign-stat* | *if-stat*
def-stat → *type* **id** { **,** **id** }
type → **int** | **bool**
assign-stat → **id** **:=** *const*
const → **intconst** | **boolconst**
if-stat → **if** **id** **then** *block* [**else** *block*] **endif**
block → **begin** { *stat* **;** }$^{+}$ **end**

based on the following assumptions:

- Lexical strings can be separated by spacing.
- Identifiers includes letters only.
- Type **bool** is defined on the domain { **true**, **false** }.

Exercise 3

```
%{
#include <stdlib.h>
#include "def.h"  /* encoding of lexical symbols */
int lexval;
%}
delimiter     [ \t\n]
spacing       {delimiter}+
letter        [A-Za-z]
digit         [0-9]
id            {letter}+
num           {digit}+
%%
{spacing}     ;
if            {return(IF);}
then          {return(THEN);}
else          {return(ELSE);}
endif         {return(ENDIF);}
begin         {return(BEGIN);}
end           {return(END);}
int           {return(INT);}
bool          {return(BOOL);}
true          {lexval = 0; return(BOOLCONST);}
false         {lexval = 1; return(BOOLCONST);}
{num}         {lexval = atoi(yytext); return(INTCONST);}
{id}          {lexval = assign_id(); return(ID);}
","           {return(COMMA);}
";"           {return(SEMICOLON);}
":="          {return(ASSIGN);}
.             {return(ERROR);}
%%
assign_id()  /* symbol table without keywords */
{ int line;
  if((line=lookup(yytext)) == 0) line = insert(yytext);
  return(line);
}
```

program → { *stat* **;** }$^{+}$
stat → *def-stat* | *assign-stat* | *if-stat*
def-stat → *type* **id** { **,** **id** }
type → **int** | **bool**
assign-stat → **id** := *const*
const → **intconst** | **boolconst**
if-stat → **if id then** *block* [**else** *block*] **endif**
block → **begin** { *stat* **;** }$^{+}$ **end**

Exercise 4

Specify in Lex the lexical analyzer of the language defined by the following EBNF:

program → { *stat* **;** }$^{+}$
stat → *def-stat* | *select-stat*
def-stat → **def id (** *attr-def* { **,** *attr-def* } **)**
attr-def → **id :** *domain*
domain → **integer** | **string** | **bool**
select-stat → **select** *id-list* **from** *id-list* [**where** *predicate*]
id-list → **id** { **,** **id** }
predicate → *condition* { (**and** | **or**) *condition* }
condition → *comparison* | *membership*
comparison → *elem comp-op elem*
elem → **id** | **intconst** | **strconst** | **boolconst**
comp-op → = | > | <
membership → **in (***elem* **,** *select-stat* **)**

Exercise 4

```
%{
#include <stdlib.h>
#include "def.h"  /* encoding of lexical symbols */
Lexval lexval;
%}
delimiter       [ \t\n]
spacing         {delimiter}+
letter          [A-Za-z]
digit           [0-9]
id              {letter}({letter}|{digit})*
num             {digit}+
%%
{spacing}       ;
def             {return(DEF);}
integer         {return(INTEGER);}
string          {return(STRING);}
bool            {return(BOOL);}
select          {return(SELECT);}
from            {return(FROM);}
where           {return(WHERE);}
and             {return(AND);}
or              {return(OR);}
in              {return(IN);}
{num}           {lexval.ival = atoi(yytext); return(INTCONST);}
\"([^\"])*\"    {lexval.sval = newstring(); return(STRCONST);}
false           {lexval.ival = 0; return(BOOLCONST);}
true            {lexval.ival = 1; return(BOOLCONST);}
{id}            {lexval.sval = assign_id(); return(ID);}
","             {return(COMMA);}
";"             {return(SEMICOLON);}
":"             {return(COLON);}
"("             {return(LPAR);}
")"             {return(RPAR);}
"="             {return(EQUAL);}
">"             {return(GREATER);}
"<"             {return(LESS);}
.               {return(ERROR);}
%%
assign_id(){ ... }  /* symbol table without keywords */
```

program → { *stat* **;** }$^{+}$
stat → *def-stat* | *select-stat*
def-stat → **def id (** *attr-def* { **,** *attr-def* } **)**
attr-def → **id :** *domain*
domain → **integer** | **string** | **bool**
select-stat → **select** *id-list* **from** *id-list* [**where** *predicate*]
id-list → **id** { **,** **id** }
predicate → *condition* { (**and** | **or**) *condition* }
condition → *comparison* | *membership*
comparison → *elem comp-op elem*
elem → **id** | **intconst** | **strconst** | **boolconst**
comp-op → = | > | <
membership → **in (***elem* **,** *select-stat* **)**

Exercise 5

Specify in Lex the lexical analyzer of the language defined by the following BNF:

program → *def-table select-op*
def-table→ **table id (** *type-list* **)**
type-list → *type-list* **,** *type* | *type*
type → **string** | **bool**
select-op→ **select id where numattr =** *const*
const → **strconst** | **boolconst**

(example of phrase)

```
table T (string, bool)
select T where 1 = "alpha"
```

Exercise 5

```
%{
#include <stdlib.h>
#include "def.h"  /* encoding of lexical symbols */
Lexval lexval;
%}
delimiter    [ \t\n]
spacing      {delimiter}+
letter       [A-Za-z]
digit        [0-9]
id           {letter}({letter}|{digit})*
numattr      {digit}+
strconst     \"([^\"])*\"
boolconst    false | true
sugar        [(),=]
%%
{spacing}    ;
{sugar}      {return(yytext[0]);}
table        {return(TABLE);}
string       {return(STRING);}
bool         {return(BOOL);}
select       {return(SELECT);}
where        {return(WHERE);}
{numattr}    {lexval.ival = atoi(yytext); return(NUMATTR);}
{strconst}   {lexval.sval = newstring(yytext); return(STRCONST);}
{boolconst}  {lexval.ival = (yytext[0]=='f' ? 0 : 1); return(BOOLCONST);}
{id}         {lexval.sval = newstring(yytext); return(ID);}
.            {return(ERROR);}
%%
char *newstring(char *s)
{
  char *p = malloc(strlen(s)+1);
  strcpy(p,s);
  return(p);
}
```

program → *def-table select-op*
def-table → **table id (** *type-list* **)**
type-list → *type-list* **,** *type* | *type*
type → **string** | **bool**
select-op → **select id where numattr =** *const*
const → **strconst** | **boolconst**

Exercise 6

Specify in Lex the lexical analyzer of the language defined by the following BNF:

program → *stat-list*
stat-list → *stat* **;** *stat-list* | **ε**
stat → *def-stat* | *if-stat* | **display**
def-stat → **id :** *type*
type → **int** | **string**
if-stat → **if** *expr* **then** *stat* **else** *stat*
expr → **boolconst**

assuming that:

- An identifier is a sequence of lowercase letters, possibly separated by an (unique) underscore, as in the following example: **alpha_beta_gamma**;
- A boolean constant can be either **true** or **false**.

Exercise 6

```
%{
#include <stdlib.h>
#include "def.h"  /* encoding of lexical symbols */
Lexval lexval;
%}
delimiter     [ \t\n]
spacing       {delimiter}+
lowercase     [a-z]
id            {lowercase}+('_'{lowercase}+)*
boolconst     false | true
sugar         [;:]
%%
{spacing}     ;
{sugar}       {return(yytext[0]);}
display       {return(DISPLAY);}
int           {return(INT);}
string        {return(STRING);}
if            {return(IF);}
then          {return(THEN);}
else          {return(ELSE);}
{boolconst}   {lexval.ival = (yytext[0]=='f' ? 0 : 1); return(BOOLCONST);}
{id}          {lexval.sval = newstring(yytext); return(ID);}
.             {return(ERROR);}
%%
char *newstring(char *s)
{
  char *p = malloc(strlen(s)+1);
  strcpy(p,s);
  return(p);
}
```

program → *stat-list*
stat-list → *stat* **;** *stat-list* | **ε**
stat → *def-stat* | *if-stat* | **display**
def-stat → **id :** *type*
type → **int** | **string**
if-stat → **if** *expr* **then** *stat* **else** *stat*
expr → **boolconst**

Exercise 7

Specify in Lex the lexical analyzer of the language defined by the following EBNF:

program → { *stat* **;** }$^{+}$
stat → *def-stat* | *procedure-call*
def-stat → **id** { **,** **id** } **:** *type*
type → **int** | **string** | **bool** | *structured-type*
structured-type → **matrix** **[** **intconst** { **,** **intconst** } **]** **of** *type*
procedure-call → **call** **id** **(** [*parameters*] **)**
parameters → *param* { **,** *param* }
param → **intconst** | **stringconst** | **boolconst** | **id**

assuming that:

- An identifier is a (nonempty) list of uppercase letters, followed by zero or more digits;
- An integer constant cannot start with digit 0;
- A string constant is a (possibly empty) sequence of alphanumeric characters enclosed between apexes.
- A boolean constant is either **true** or **false**.

Exercise 7

```
%{
#include <stdlib.h>
#include "def.h"  /* encoding of lexical symbols */
Lexval lexval;
%}
delimiter       [ \t\n]
spacing         {delimiter}+
uppercase       [A-Z]
lowercase       [a-z]
letter          {uppercase} | {lowercase}
digit           [0-9]
initial_digit   [1-9]
alphanum        {letter} | {digit}
intconst        {initial_digit}{digit}*
boolconst       false | true
strconst        \"{alphanum}*\"
id              {uppercase}+{digit}*
sugar           [,;:()\[\]]
%%
{spacing}       ;
{sugar}         {return(yytext[0]);}
matrix          {return(MATRIX);}
int             {return(INT);}
string          {return(STRING);}
bool            {return(BOOL);}
of              {return(OF);}
call            {return(CALL);}
{intconst}      {lexval.ival = atoi(yytext); return(INTCONST);}
{strconst}      {lexval.sval = newstring(yytext); return(STRCONST);}
{boolconst}     {lexval.ival = (yytext[0]=='f' ? 0 : 1); return(BOOLCONST);}
{id}            {lexval.ival = assign_id(yytext); return(ID);}
.               {return(ERROR);}
%%
char *newstring(char *s){
  char *p = malloc(strlen(s)+1);
  strcpy(p,s);
  return(p);}

int assign_id(){...}
```

program → { *stat* **;** }$^{+}$
stat → *def-stat* | *procedure-call*
def-stat → **id** { **,** **id** } **:** *type*
type → **int** | **string** | **bool** | *structured-type*
structured-type → **matrix [intconst** { **,** **intconst** } **]** **of** *type*
procedure-call → **call id (** [*parameters*] **)**
parameters → *param* { **,** *param* }
param → **intconst** | **stringconst** | **boolconst** | **id**

Exercise 8

Specify in Lex the lexical analyzer of the language defined by the following BNF:

program → *def-list*
def-list → *def* **;** *def-list* | **ε**
def → *type-def* | *function-def*
type-def → **type id** = *domain*
domain → **int** | **string** | **[** *domain* **]**
function-def → **function id (** *param-list* **) :** *domain*
param-list → *param* **,** *param-list* | **ε**
param → **id :** *domain*

assuming that:

- An identifier is a (nonempty) sequence of lowercase letters;
- An identifier may include one or more underscores '_';
- An identifier neither starts not ends with an underscore '_';
- An identifier cannot contain sequences of two or more underscores '_';

Exercise 8

```
%{
#include <stdlib.h>
#include "def.h"  /* encoding of lexical symbols */
Lexval lexval;
%}
delimiter     [ \t\n]
spacing       {delimiter}+
lowercase     [a-z]
underscore     \_
id            {lowercase}+{underscore {lowercase}+}*
sugar         [,;:=()\[\]]
%%
{spacing}     ;
{sugar}       {return(yytext[0]);}
type          {return(TYPE);}
int           {return(INT);}
string        {return(STRING);}
function      {return(FUNCTION);}
{id}          {lexval.ival = assign_id(yytext); return(ID);}
.             {return(ERROR);}
%%
int assign_id(char *s)
{
  int line;
  if((line = lookup(s)) == 0) line = insert(s);
  return(line);
}
```

program → *def-list*
def-list → *def* **;** *def-list* | **ε**
def → *type-def* | *function-def*
type-def → **type id** = *domain*
domain → **int** | **string** | **[** *domain* **]**
function-def → **function id (** *param-list* **) :** *domain*
param-list → *param* **,** *param-list* | **ε**
param → **id :** *domain*

Exercise 9

Specify in Lex the lexical analyzer of the language defined by the following EBNF:

program → { *stat* **;** }$^{+}$
stat → *def-stat* | *assign-stat* | *if-stat*
def-stat → **def id** { **,** **id** } **:** *type*
type → **int** | **string** | **bool** | *record-type*
record-type → **record** **(** **id :** *type* { **,** **id :** *type* } **)**
assign-stat → **id :=** (**id** | *const*)
const → (**intconst** | **strconst** | **boolconst**)
if-stat → **if** *cond* **then** *stat* [*elsif-part*] **otherwise** *stat*
elsif-part → { **elsif** *cond* **then** *stat* }$^{+}$
cond → **id =** *const*

assuming that:

- An identifier is a sequence of three alphanumeric characters, starting with a letter;
- An integer constant is a nonempty sequence of digits, which cannot start with zero;
- A string constant is a (possibly empty) sequence of alphanumeric characters, possibly separated by blank spaces, enclosed in double apexes;
- A boolean constant is either `true` or `false`.

Exercise 9

```
%{
#include <stdlib.h>
#include "def.h"  /* encoding of lexical symbols */
Lexval lexval;
%}
blank           ' '
delimiter       {blank} | \t | \n
spacing         {delimiter}+
letter          [A-Za-z]
digit           [0-9]
initial_digit   [1-9]
id              {letter}({letter}|{digit})({letter}|{digit})
intconst        {initial_digit}{digit}*
strconst        \" ({blank} | {letter} | {digit})* \"
boolconst       true | false
sugar           [,;:=()]
%%
{spacing}       ;
{sugar}         {return(yytext[0]);}
:=              {return(ASSIGN):}
def             {return(DEF);}
int             {return(INT);}
string          {return(STRING);}
bool            {return(BOOL);}
record          {return(RECORD);}
if              {return(IF);}
then            {return(THEN);}
elsif           {return(ELSIF);}
otherwise       {return(OTHERWISE);}
{intconst}      {lexval.ival = atoi(yytext); return(INTCONST);}
{strconst}      {lexval.sval = copy(yytext); return(STRCONST);}
{boolconst}     {lexval.ival = (yytext[0] == 'f' ? 0 : 1); return(BOOLCONST);}
{id}            {lexval.ival = assign_id(yytext); return(ID);}
.               {return(ERROR);}
%%
```

program → { *stat* **;** }$^{+}$
stat → *def-stat* | *assign-stat* | *if-stat*
def-stat → **def id** { **,** **id** } **:** *type*
type → **int** | **string** | **bool** | *record-type*
record-type → **record** **(** **id :** *type* { **,** **id :** *type* } **)**
assign-stat → **id :=** (**id** | *const*)
const → (**intconst** | **strconst** | **boolconst**)
if-stat → **if** *cond* **then** *stat* [*elsif-part*] **otherwise** *stat*
elsif-part → { **elsif** *cond* **then** *stat* }$^{+}$
cond → **id =** *const*

```
int assign_id(char *s;)
{
  int line;
  if((line=lookup(s))==0) line = insert(s);
  return(line);
}

char *copy(char *s)
{ char *p = malloc(strlen(s)+1);
  strcpy(p, s);
  return(p);
}
```

Exercise 10

Specify in Lex the lexical analyzer of the language defined by the following BNF:

program → *stat*
stat → *def-stat* | *assign-stat*
def-stat → **relation id :** *rel-type*
rel-type → **[** *attr-list* **]**
attr-list → *attr-def attr-list* | **ε**
attr-def → **id :** *type*
type → *atomic-type* | *rel-type*
atomic-type → **int** | **string**
assign-stat → **id :=** *const*
const → **intconst** | **strconst**

assuming that:

- An identifier is a sequence of alphanumeric characters starting with a letter;
- The longest length of an identifier is three characters;
- An integer constant is a sequence of digits, which cannot start with zero;
- A string constant is a (possibly empty) sequence of letters, possibly separated by one or more blank spaces, enclosed in double apexes.

Exercise 10

```
%{
#include <stdlib.h>
#include "def.h"  /* encoding of lexical symbols */
Lexval lexval;
%}
blank           ' '
delimiter       {blank} | \t | \n
spacing         {delimiter}+
letter          [A-Za-z]
digit           [0-9]
alphanum        {letter}|{digit}
initial_digit   [1-9]
id              {letter}{alphanum}?{alphanum}?
intconst        {initial_digit}{digit}*
strconst        \" ({letter}|{blank})* \"
sugar           [\[:\]]
%%
{spacing}     ;
{sugar}       {return(yytext[0]);}
:=            {return(ASSIGN):}
relation      {return(RELATION);}
int           {return(INT);}
string        {return(STRING);}
{intconst}    {lexval.ival = atoi(yytext); return(INTCONST);}
{strconst}    {lexval.sval = copy(yytext); return(STRCONST);}
{id}          {lexval.ival = assign_id(yytext); return(ID);}
.             {return(ERROR);}
%%
int assign_id(char *s){...}
char *copy(char *s){...}
```

program → *stat*
stat → *def-stat* | *assign-stat*
def-stat → **relation id :** *rel-type*
rel-type → **[** *attr-list* **]**
attr-list → *attr-def attr-list* | **ε**
attr-def → **id :** *type*
type → *atomic-type* | *rel-type*
atomic-type → **int** | **string**
assign-stat → **id :=** *const*
const → **intconst** | **strconst**

Exercise 11

Specify in Lex the lexical analyzer of the language defined by the following EBNF:

program → **program id** [*var-section*] *body* **.**
var-section → **var** { *decl-list* **;** }$^{+}$
decl-list → **id** { **,** **id** } **:** *type*
type → **int** | **real** | **string**
body → **begin** *block* **end**
block → { *stat* **;** }$^{+}$
stat → *assign-stat* | *if-stat*
assign-stat → **id** **:=** (*const* | **id**)
const → **intconst** | **realconst** | **stringconst**
if-stat → **if** *cond* **then** *block* { **elsif** *cond* **then** *block* } [**otherwise** *block*] **endif**
cond → **id** (= | **!=** | >| < | >=| <=) (*const* | **id**)

assuming that:

- An identifier is a sequence of three alphanumeric characters, starting with a letter;
- An integer constant is a sequence of digits , which cannot start with zero;
- A real constant includes an integer part and a decimal part (separated by a dot);
- A string constant is a (possibly empty) sequence of alphanumeric characters, possibly separated by (one or more) blank spaces, enclosed in double apexes.

Exercise 11

```
%{
#include <stdlib.h>
#include "def.h"  /* encoding of lexical symbols */
Lexval lexval;
%}
blank           ' '
delimiter       {blank} | \t | \n
spacing         {delimiter}+
letter          [A-Za-z]
digit           [0-9]
alphanum        {letter}|{digit}
initial_digit   [1-9]
id              {letter}{alphanum}{alphanum}
realconst       {intconst}\.{intconst}
intconst        {initial_digit}{digit}*
strconst        \" ({alphanum}|{blank})* \"
sugar           [\.;,=><]
%%
{spacing}    ;
{sugar}      {return(yytext[0]);}
:=           {return(ASSIGN):}
!=           {return(NE):}
>=           {return(GE):}
<=           {return(LE):}
program      {return(PROGRAM);}
int          {return(INT);}
real         {return(REAL);}
string       {return(STRING);}
begin        {return(BEGIN);}
end          {return(END);}
if           {return(IF);}
then         {return(THEN);}
elsif        {return(ELSIF);}
otherwise    {return(OTHERWISE);}
endif        {return(ENDIF);}
{intconst}   {lexval.ival = atoi(yytext); return(INTCONST);}
{realconst}  {lexval.rval = atof(yytext); return(REALCONST);}
{strconst}   {lexval.sval = copy(yytext); return(STRCONST);}
{id}         {lexval.ival = assign_id(yytext); return(ID);}
```

program → **program id** [*var-section*] *body* **.**
var-section → **var** { *decl-list* **;** }$^{+}$
decl-list → **id** { **,** **id** } **:** *type*
type → **int** | **real** | **string**
body → **begin** *block* **end**
block → { *stat* **;** }$^{+}$
stat → *assign-stat* | *if-stat*
assign-stat → **id** **:=** (*const* | **id**)
const → **intconst** | **realconst** | **stringconst**
if-stat → **if** *cond* **then** *block* { **elsif** *cond* **then** *block* } [**otherwise** *block*] **endif**
cond → **id** (= | **!=** | >| < | >=| <=) (*const* | **id**)

```
.       {return(ERROR);}

%%

int assign_id(char *s){...}
char *copy(char *s){...}
```

Exercise 12

Specify in Lex the lexical analyzer of the language of the following fragment of code:

```
int i = 10, j = -7, k = +55;                    -- counters
string n_of_file2 = "test.log";                 -- file must exist
real x = 10.22, y = -35.06, z = +15.000, w = 0.01;
while i > j do
    alpha(i);
end;
```

assuming that:

- A comment starts with '--' and ends with a newline;
- An integer constant long more than one digit cannot start with zero;
- A real constant is expressed by an integer part and a decimal part, both mandatory;
- In a real constant, the integer part long more than one digit cannot start with zero;
- Both integer and real constants are possibly qualified with a sign (which is part of the constant);
- A string constant is enclosed in double apexes and can contain any character, except newline;
- An identifier starts with an alphabetic character and is followed by a (possibly empty) sequence of alphanumeric characters, possibly separated by underscores '_';
- An identifier neither includes two or more consecutive underscores nor ends with an underscore.

Exercise 12

```
%{
#include <stdlib.h>
#include "def.h"  /* encoding of lexical symbols */
Lexval lexval;   /* ival, rval, sval */
%}
blank          " "
delimiter      {blank}|\t|\n
spacing        {delimiter}+
comment        "--"(.)*\n
letter         [A-Za-z]
digit          [0-9]
alphanum       {letter}|{digit}
initial_digit  [1-9]
sign            + | -
id             {letter}{alphanum}*(_{alphanum}+)*
intconst       {sign}?({initial_digit} {digit}* | 0)
realconst      {intconst}\.{digit}+
strconst       \"[^\n\"]*\"
sugar          [=;,>()]

%%
```

```
{spacing} ;
{comment}    ;
{sugar}     {return(yytext[0]);}
int         {return(INT);}
string      {return(STRING);}
real        {return(REAL);}
while       {return(BEGIN);}
do          {return(DO);}
end         {return(END);}
{intconst}  {lexval.ival = atoi(yytext); return(INTCONST);}
{realconst} {lexval.rval = atof(yytext); return(REALCONST);}
{strconst}  {lexval.sval = copy(yytext); return(STRCONST);}
{id}        {lexval.ival = assign_id(yytext); return(ID);}
.           {return(ERROR);}

%%

int assign_id(char *s)
{ int line;
  if((line = lookup(s)) == 0) line = insert(s);
  return(line);
}

char *copy(char *s)
{
  char *p = malloc(strlen(s)+1);
  strcpy(p, s);
  return(p);
}
```

Exercise 13

Specify in Lex the lexical analyzer of the language defined by the following EBNF:

program→ {*stat*}$^+$
stat → (*var-decl* | *proc-decl* | *if-stat* | *while-stat* | *assign-stat* | *call*) **;**
var-decl → *type* **id** {**,** **id**}
type → **int** | **string** | **bool**
proc-decl → **procedure id (** **id** {**,** **id**} **)** {*stat*}$^+$ **end**
if-stat → **if** *expr* **then** {*stat*}$^+$ { **elsif** *expr* **then** {*stat*}$^+$ } [**else** {*stat*}$^+$] **end**
expr → **id** | **boolconst** | **intconst** | **strconst**
while-stat → **while** *expr* **do** {*stat*}$^+$ **end**
assign-stat → **id =** *expr*
call → **call id (** *expr* {**,** *expr*} **)**

assuming that:

- An identifier is an alphanumeric string, long at most 4 characters, starting with a letter;
- An integer constant cannot start with zero;
- A (possibly empty) string constant is enclosed in double apexes and contains any character different from newline;
- A boolean constant is either **`true`** or **`false`**;
- It is possible to include comments, starting with symbol **#** and ending with a newline.

Exercise 13

program→ {*stat*}$^+$
stat → (*var-decl* | *proc-decl* | *if-stat* | *while-stat* | *assign-stat* | *call*) **;**
var-decl → *type* **id** {**,** **id**}
type → **int** | **string** | **bool**
proc-decl → **procedure id (id** {**,** **id**} **)** {*stat*}$^+$ **end**
if-stat → **if** *expr* **then** {*stat*}$^+$ { **elsif** *expr* **then** {*stat*}$^+$ } [**else** {*stat*}$^+$] **end**
expr → **id** | **boolconst** | **intconst** | **strconst**
while-stat → **while** *expr* **do** {*stat*}$^+$ **end**
assign-stat → **id =** *expr*
call → **call id (** *expr* {**,** *expr*} **)**

```
%{
#include <stdlib.h>
#include "def.h"  /* encoding of lexical symbols */
Lexval lexval;
%}
blank        ' '
delimiter    {blank} | \t | \n
spacing      {delimiter}+
letter       [A-Za-z]
digit         [0-9]
alphanum      {letter}|{digit}
initial_digit [1-9]
id           {letter}{alphanum}?{alphanum}?{alphanum}?
intconst      {initial_digit}{digit}* | 0
strconst      \".*\"
boolconst     true|false
comment       #[^\"\n]*\n
sugar           [;,()=]
%%
```

```
{spacing} ;
{comment} ;
{sugar}      {return(yytext[0]);}
program      {return(PROGRAM);}
int          {return(INT);}
string       {return(STRING);}
bool         {return(BOOL);}
end          {return(END);}
if           {return(IF);}
then         {return(THEN);}
elsif        {return(ELSIF);}
else         {return(ELSE);}
while        {return(WHILE);}
do           {return(DO);}
procedure    {return(PROCEDURE);}
call         {return(CALL);}
{intconst}   {lexval.ival = atoi(yytext); return(INTCONST);}
{strconst}   {lexval.sval = copy(yytext); return(STRCONST);}
{boolconst}  {lexval.ival = (yytext[0]=='t' ? 1 : 0);}
{id}         {lexval.ival = assign_id(yytext); return(ID);}
.            {return(ERROR);}
%%
int assign_id(char *s){...}
char *copy(char *s){...}
```

Exercise 14

Specify in *Lex* a program that takes as input a text file and prints the text lines (sequences of characters terminated by a newline) which are composed of exactly three **w** letters separated between them by other characters.

Exercise 14

Specify in *Lex* a program that takes as input a text file and prints the text lines (sequences of characters terminated by a newline) which are composed of exactly three **w** letters separated between them by other characters.

```
%{
#include <stdio.h>
%}
notw        [^w\n]
line        {notw}*w{notw}+w{notw}+w{notw}*\n
%%
{line}  ECHO;
.       ;
%%
main(){yylex();}
```

Exercise 15

Specify in *Lex* a complete program that takes as input a text file and prints the content of the file where integer numbers with sign (either plus or minus) are replaced by the same number without sign.

Exercise 15

Specify in *Lex* a complete program that takes as input a text file and prints the content of the file where integer numbers with sign (either plus or minus) are replaced by the same number without sign.

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%option noyywrap
snum [\-+][0-9]+
%%
{snum} {printf("%s",yytext+1);
%%
main(){yylex();}
```