```
/*
 * Paul Heidelman
 * Case Western Reserve University
 * EECS 337 Fall 2009
 * Homework 2
 */

######
FILES
######

 Makefile
  Used to compile the project with the 'make' command.
 lex.l
  Contains the code for the lexical analyzer.
 linked_list.c
  Contains the updated and modified code for the linked list problem in HW1.
 linked_list_hw2.c
  Code that is used in the lexical analyzer to make a linked list for the symbol table.
 linked_list_hw2.h
  Header file for linked list functions/structures.
 readme
  This file.
 test
  The test sample code that is run through the final lexical analyzer as a test.

######
REGULAR EXPRESSIONS
######

 id:  [a-zA-Z_][a-zA-Z0-9_]*
 basic: (int|float)
 num: [+-]?[0-9]+
 real: [+-]?[0-9]*\.?[0-9]+(E[+-]?[0-9]+)?

######
SYMBOL TABLE
######

Modified Linked List Ouptut:

 paul[heidelmanp2]$ gcc linked_list.c
 linked_list.c: In function 'main':
 linked_list.c:173: warning: passing argument 1 of 'list_init' from incompatible pointer type
 paul[heidelmanp2]$ ./a.out
 Creating double_list...
 found: 2.000000
 found: 4.000000
 found: 6.000000
 not found
 found: 4.000000


 Creating string_list...
 found: 2.000000
 found: 6.000000
 not found
 not found
 found: 6.000000

######
LEXICAL ANALYZER
######

To compile:
 Be in the directory "heidelmanp2"
 Run "make"
```

```
To run:
 Be in the directory "heidelmanp2"
 Run "./myprog test" where test is the file you wish to input.
 If no file is specified, it defaults to stdin.

Output:

 paul[heidelmanp2]$ make
 flex lex.l
 gcc -c lex.yy.c -o lex.yy.o
 lex.l: In function 'yylex':
 lex.l:84: warning: assignment makes pointer from integer without a cast
 lex.l: In function 'installID':
 lex.l:129: warning: return makes integer from pointer without a cast
 lex.l:129: warning: function returns address of local variable
 gcc -c linked_list_hw2.c -o linked_list_hw2.o
 gcc lex.yy.o linked_list_hw2.o -o myprog
 paul[heidelmanp2]$ ./myprog test

 Using file test
 Search: list is empty!
 { comment basic <id, i>; basic <id, j>; basic <id, v>; basic <id, x>; basic[<num, 100>] <id, a>;
  while( true ) {
   do <id, i> = <id, i><num, +1>; while( <id, a>[<id, i>] < <id, v>);
   do <id, j> = <id, j><num, -1>; while( <id, a>[<id, j>] > <id, v>);
   if( <id, i> >= <id, j> ) break;
   <id, x> = <id, a>[<id, i>]; <id, a>[<id, i>] = <id, a>[<id, j>]; <id, a>[<id, j>] = <id, x>;
  }
 }
 id's:
 i
 j
 v
 x
 a
 paul[heidelmanp2]$

Notes:
 It was difficult to find an acceptable value for yylval and the return statements.  Because they won't
be needed until Homework 3, I am not too woried about them being missing/incorrect. I will correct them
when I can actually use and test them in HW3.
 list_search() outputs an error on the first run, because we search for something on an empty list.

######
SOURCE CONTROL
######
 I set up a git repo on unfuddle.com




To Print:
 design notes
 how to compile and regex
 linked_list.h file
 linked_list test file
  output from ^
 lex file
 output form lex file
 text file
```

```
#/*
# * Paul Heidelman
# * Case Western Reserve University
# * EECS 337 Fall 2009
# * Homework 2
# */

CC = gcc
OBJ = lex.yy.o linked_list_hw2.o
#all: myprog ./test

%.o: %.c
 $(CC) -c $< -o $@

myprog: $(OBJ)
 $(CC) $(OBJ) -o myprog

lex.yy.c: lex.l
 flex lex.l

clean:
 rm -f *.o lex.yy.c myprog a.out

.PHONY: all clean
```

```c
/*
 * Paul Heidelman
 * Case Western Reserve University
 * EECS 337 Fall 2009
 * Homework 1
 * Modified for Homework 2
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>


typedef struct list_entry list_entry_t;

struct list_entry {
 void *key;
 void *value;
 list_entry_t *next;
};

typedef struct {
 list_entry_t *list;
 int (*cmp)(const void *, const void *); /* Comparison function */
} list_head_t;


list_head_t *list_init(int (*cmp)(const void *, const void *)){
 //assign cmp
 list_head_t *n;
 n = (list_head_t *) malloc(sizeof(list_head_t));
 if(n == NULL)
 {
  fprintf(stderr, "Error creating list_head_t\n");
  return NULL;
 }
 n->cmp = cmp;
 return n;
};

list_entry_t *list_insert(list_head_t *head, void *key, void *value){
 if(head == NULL)
 {
  fprintf(stderr, "Invalid list_head_t\n");
  return NULL;
 }
 if(key == NULL)
 {
  fprintf(stderr, "Invalid key\n");
  return NULL;
 }
 if(value == NULL)
 {
  fprintf(stderr, "List is empty!\n");
  return NULL;
 }

 list_entry_t *new_entry;
 new_entry = (list_entry_t *) malloc(sizeof(list_entry_t));
 new_entry->key = key;
 new_entry->value = value;
 list_entry_t *n;
 n = (list_entry_t *) malloc(sizeof(list_entry_t));

 if(head->list == NULL)
 {//then the list is empty, so add a first entry
```

```
  //printf("new");
  head->list = new_entry;
 }
 else
 {
  //printf("adding");
  n = head->list;
  while(n->next != NULL)
   n = n->next;
  n->next = new_entry;
 }
 return new_entry;
};

void *list_search(list_head_t *head, void *key){
 if(head == NULL)
 {
  fprintf(stderr, "Invalid list_head_t\n");
  return NULL;
 }
 if(key == NULL)
 {
  fprintf(stderr, "Invalid key\n");
  return NULL;
 }
 if(head->list == NULL)
 {
  fprintf(stderr, "List is empty!\n");
  return NULL;
 }

 list_entry_t *n;
 n = head->list;
 while(n != NULL)
 {
  //printf("not %ld\n",n->value);
  if(head->cmp(n->key, key) == 0)
   return n->value;
  else
  n = n->next;
 }
 return NULL;
};

void list_delete(list_head_t *head){
 assert(head != NULL);
 list_entry_t *temp1;
 temp1 = (list_entry_t *) malloc(sizeof(list_entry_t));
 temp1 = head->list;
 while (temp1 != NULL)
 {
  list_entry_t *temp2;
  temp2 = (list_entry_t *) malloc(sizeof(list_entry_t));
  temp2 = temp1->next;
  free(temp1);
  temp1 = temp2;
 }
/*
 list_entry_t *o;
 o = (list_entry_t *) malloc(sizeof(list_entry_t));
 list_entry_t *p;
 p = (list_entry_t *) malloc(sizeof(list_entry_t));
 o = head->list;
 free(head);
 while(o != NULL)
 {
  //list_entry_t *p = n->next;
```

```
  p = o->next;
  free(o);
  o = p;
 }
 free(p);
*/
 return;
};

int cmp_double(double *a, double *b){
 if(*a == *b)
  return 0;
 else
  return 1;
};

int cmp_string(const void *a, const void *b){
 assert(a != NULL);
 assert(b != NULL);
 return strcmp(a, b);
};

int main(){

 /*
 insert(1,2)
 insert(3,4)
 insert(5,6)
 search(1)
 search(3)
 search(5)
 insert(3,7)
 search(8)
 search(3)
 */

 printf("Creating double_list...\n");
 list_head_t *double_list;
 double_list = list_init(cmp_double);

 double a = 1;
 double b = 2;
 list_insert(double_list, &a, &b);

 double c = 3;
 double d = 4;
 list_insert(double_list, &c, &d);

 double e = 5;
 double f = 6;
 list_insert(double_list, &e, &f);

 double s = 1;
 double *result;
 result = list_search(double_list, &s);
 if(result != NULL)
  printf("found: %lf\n", *result);

 else
  printf("not found\n");

 s = 3;
 result = list_search(double_list, &s);
 if(result != NULL)
  printf("found: %lf\n", *result);

 else
```

```
 printf("not found\n");

s = 5;
result = list_search(double_list, &s);
if(result != NULL)
 printf("found: %lf\n", *result);

else
 printf("not found\n");

double g = 3;
double h = 7;
list_insert(double_list, &g, &h);

s = 8;
result = list_search(double_list, &s);
if(result != NULL)
 printf("found: %lf\n", *result);

else
 printf("not found\n");


s = 3;
result = list_search(double_list, &s);
if(result != NULL)
 printf("found: %lf\n", *result);

else
 printf("not found\n");

list_delete(double_list);



/*
 *insert("one",2)
 *insert(two",4)
 *insert(three,6)
 *search(one)
 *search(three)
 *search(five)
 *insert(three,7)
 *search(eight)
 *search(three)
 */

printf("\n\nCreating string_list...\n");
list_head_t *string_list;
string_list = list_init(cmp_string);

char str1[] = "one";
double dbl1 = 2;
list_insert(string_list, &str1, &dbl1);

char str2[] = "two";
double dbl2 = 4;
list_insert(string_list, &str2, &dbl2);

char str3[] = "three";
double dbl3 = 6;
list_insert(string_list, &str3, &dbl3);

char search_str[] = "one";
result = list_search(string_list, &search_str);
if(result != NULL)
 printf("found: %lf\n", *result);
```

```
 else
  printf("not found\n");

 strcpy(search_str, "three");
 result = list_search(string_list, &search_str);
 if(result != NULL)
  printf("found: %lf\n", *result);

 else
  printf("not found\n");

 strcpy(search_str, "five");
 result = list_search(string_list, &search_str);
 if(result != NULL)
  printf("found: %lf\n", *result);

 else
  printf("not found\n");

 char str4[] = "three";
 double dbl4 = 7;
 list_insert(string_list, &str4, &dbl4);

 strcpy(search_str, "eight");
 result = list_search(string_list, &search_str);
 if(result != NULL)
  printf("found: %lf\n", *result);

 else
  printf("not found\n");

 strcpy(search_str, "three");
 result = list_search(string_list, &search_str);
 if(result != NULL)
  printf("found: %lf\n", *result);

 else
  printf("not found\n");


 list_delete(string_list);
 return 0;
};
```

```c
/*
 * Paul Heidelman
 * Case Western Reserve University
 * EECS 337 Fall 2009
 * Homework 2
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>


typedef struct list_entry list_entry_t;

struct list_entry {
 void *key;
 void *value;
 list_entry_t *next;
};

typedef struct {
 list_entry_t *list;
 int (*cmp)(const void *, const void *); /* Comparison function */
} list_head_t;


list_head_t *list_init(int (*cmp)(const void *, const void *)){
 //assign cmp
 list_head_t *n;
 n = (list_head_t *) malloc(sizeof(list_head_t));
 if(n == NULL)
 {
  fprintf(stderr, "Error creating list_head_t\n");
  return NULL;
 }
 n->cmp = cmp;
 return n;
};

list_entry_t *list_insert(list_head_t *head, void *key, void *value){
 if(head == NULL)
 {
  fprintf(stderr, "Invalid list_head_t\n");
  return NULL;
 }
 if(key == NULL)
 {
  fprintf(stderr, "Invalid key\n");
  return NULL;
 }

 list_entry_t *new_entry;
 new_entry = (list_entry_t *) malloc(sizeof(list_entry_t));
 new_entry->key = key;
 new_entry->value = value;
 list_entry_t *n;
 n = (list_entry_t *) malloc(sizeof(list_entry_t));

 if(head->list == NULL)
 {//then the list is empty, so add a first entry
  //printf("new");
  head->list = new_entry;
 }
 else
 {
  //printf("adding");
```

```
  n = head->list;
  while(n->next != NULL)
   n = n->next;
  n->next = new_entry;
 }
 return new_entry;
};

void *list_search(list_head_t *head, void *key){
 if(head == NULL)
 {
  fprintf(stderr, "Invalid list_head_t\n");
  return NULL;
 }
 if(key == NULL)
 {
  fprintf(stderr, "Invalid key\n");
  return NULL;
 }
 if(head->list == NULL)
 {
  fprintf(stderr, "Search: list is empty!\n");
  return NULL;
 }

 list_entry_t *n;
 n = head->list;
 while(n != NULL)
 {
  if(head->cmp(n->key, key) == 0)
  {
   //printf("found!");
   return n->key;
  }
  else
  {
   //printf("key %s is not %s\n",key, n->key);
   n = n->next;
  }
 }
 return NULL;
};

void list_delete(list_head_t *head){
 assert(head != NULL);
 list_entry_t *temp1;
 temp1 = (list_entry_t *) malloc(sizeof(list_entry_t));
 temp1 = head->list;
 while (temp1 != NULL)
 {
  list_entry_t *temp2;
  temp2 = (list_entry_t *) malloc(sizeof(list_entry_t));
  temp2 = temp1->next;
  free(temp1);
  temp1 = temp2;
 }
/*
 list_entry_t *o;
 o = (list_entry_t *) malloc(sizeof(list_entry_t));
 list_entry_t *p;
 p = (list_entry_t *) malloc(sizeof(list_entry_t));
 o = head->list;
 free(head);
 while(o != NULL)
 {
  //list_entry_t *p = n->next;
  p = o->next;
```

```
  free(o);
  o = p;
 }
 free(p);
*/
 return;
};

int cmp_double(double *a, double *b){
 if(*a == *b)
  return 0;
 else
  return 1;
};

int cmp_string(const void *a, const void *b){
 assert(a != NULL);
 assert(b != NULL);
 return strcmp(a, b);
};
```

```c
/*
 * Paul Heidelman
 * Case Western Reserve University
 * EECS 337 Fall 2009
 * Homework 2
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
typedef struct list_entry list_entry_t;

struct list_entry {
 void *key;
 void *value;
 list_entry_t *next;
};

typedef struct {
 list_entry_t *list;
 int (*cmp)(const void *, const void *); /* Comparison function */
} list_head_t;

list_head_t *list_init(int (*cmp)(const void *, const void *));

list_entry_t *list_insert(list_head_t *head, void *key, void *value);

void *list_search(list_head_t *head, void *key);

void list_delete(list_head_t *head);

int cmp_double(double *a, double *b);

int cmp_string(const void *a, const void *b);
```

```
/*
 * Paul Heidelman
 * Case Western Reserve University
 * EECS 337 Fall 2009
 * Homework 2
 */


%{
 #include "linked_list_hw2.h"
 /* definitions of manifest constants
 LT, LE, EQ, NE, GT, GE,
 IF, THEN, ELSE, ID, NUMBER, RELOP */
 #define LT 1
 #define LE 2
 #define EQ 3
 #define NE 4
 #define GT 5
 #define GE 6
 #define IF 7
 #define THEN 8
 #define ELSE 9
 #define ID 10
 #define NUMBER 11
 #define RELOP 12
 #define CO 13
 #define BO 14
 #define BC 15
 #define WHILE 16
 #define DO 17
 #define BREAK 18
 #define BASIC 19
 #define TRUE 20
 #define FALSE 21

 void* yylval;
 int int_var;
 int float_var;
 int true_var;
 int false_var;
 int line_commented;
 int block_commented;
 list_head_t *id_list;

%}

/* regular definitions */
delim [\t\n]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
number [+-]?{digit}+
real [+-]?{digit}+(\.{digit}+)?(E[+-]?{digit}+)?
comment (("//".*\n)|("/*".*"*/"))
modifier const|extern|signed|static|unsigned|volatile
custom struct|enum|union|typedef
basic   int|float
type double|long|short|void|char|bool
tf (TRUE|true)|(FALSE|false)

%%

{ws} {/* no action and no return */printf("%s", yytext);}
if {printf("if"); return(RELOP);}
then {printf("then"); return(RELOP);}
else {printf("else"); return(RELOP);}
```

```
while {printf("while"); return(RELOP);}
do {printf("do"); return(RELOP);}
for {printf("for"); return(RELOP);}
break {printf("break"); return(RELOP);}
continue {printf("continue"); return(RELOP);}
switch {printf("switch"); return(RELOP);}
case {printf("case"); return(RELOP);}
default {printf("default"); return(RELOP);}
goto {printf("goto"); return(RELOP);}
return {printf("return"); return(RELOP);}
{modifier} {printf("modifier");}
{custom} {printf("custom");}
{type} {printf("type");}
{comment} {printf("comment");}
{basic} {printf("basic"); if(strcmp(yytext, "int") == 0) yylval = (void*) &int_var; else yylval = (void*)
&float_var; return(RELOP);}
{tf} {if(strcmp(yytext, "true") == 0){printf("true"); yylval = (void*) &true_var;} else {printf("false");
yylval = (void*) &false_var;} return(RELOP);}
{id} {yylval = (int) installID(); printf("<id, %s>", yytext); return(RELOP);}
{number} {printf("<num, %s>", yytext); int* n = malloc(sizeof(atoi(yytext))); *n = atoi(yytext); yylval =
(void*)n; return(RELOP);}
{real} {printf("<real, %s>", yytext); int* n = malloc(sizeof(atoi(yytext))); *n = atoi(yytext); yylval =
(void*)n; return(RELOP);}
"<" {printf("<"); return(RELOP) ;}
"+" {printf("+"); return(RELOP) ;}
"-" {printf("-"); return(RELOP) ;}
"<=" {printf("<="); return(RELOP) ;}
"=" {printf("="); return(RELOP) ;}
"<>" {printf("<>"); return(RELOP) ;}
">" {printf(">"); return(RELOP) ;}
">=" {printf(">="); return(RELOP) ;}
"!=" {printf("!="); return(RELOP) ;}
"==" {printf("=="); return(RELOP) ;}
"&" {printf("&"); return(RELOP) ;}
"&&" {printf("&&"); return(RELOP) ;}
"||" {printf("||"); return(RELOP) ;}
"|" {printf("|"); return(RELOP) ;}
"*" {printf("*"); return(RELOP) ;}
"/" {printf("/"); return(RELOP) ;}
"^" {printf("^"); return(RELOP) ;}
"!" {printf("!"); return(RELOP) ;}
"+=" {printf("+="); return(RELOP) ;}
"-=" {printf("-="); return(RELOP) ;}
"*=" {printf("*="); return(RELOP) ;}
"/=" {printf("/="); return(RELOP) ;}
"++" {printf("++"); return(RELOP) ;}
"--" {printf("--"); return(RELOP) ;}

%%

int installID() {
 /* function to install the lexeme, whose
 first character is pointed to by yytext,
 and whose length is yyleng, into the
 symbol table and return a pointer
 thereto */
 char *str1 = malloc(sizeof(yytext));
 strcpy(str1, yytext);
 //printf("|%s|", str1);
 //printf("%s", list_search(id_list, yytext));
 if(list_search(id_list, yytext) == NULL)
 {
  list_insert(id_list, (void*)str1, NULL);
 }
 return &str1;
}
```

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

main( argc, argv )
int argc;
char **argv;
{

 id_list = list_init(cmp_string);

 ++argv, --argc; /* skip over program name */
 if ( argc > 0 )
 {
  yyin = fopen( argv[0], "r" );
  printf("\nUsing file %s\n", argv[0]);
 }
 else
  yyin = stdin;

 while(yylex() != 0)
  yylex();


 printf("id's:\n");
 list_entry_t *cur;
 cur = id_list->list;
 while (cur != NULL)
 {
  printf("%s\n",(char*)cur->key);
  cur = cur->next;
 }

 list_delete(id_list);

 return 0;
}
```

```
{ //File test
 int i; int j; float v; float x; float[100] a;
 while( true ) {
  do i = i+1; while( a[i] < v);
  do j = j-1; while( a[j] > v);
  if( i >= j ) break;
  x = a[i]; a[i] = a[j]; a[j] = x;
 }
}
```