CS 347 - Assignment 1

Group - 6

Grammar

```
statements          -->          statement statements | statement
statement           -->          ID:=expr1;
                                 |        IF expr1 then statement
                                 |        while expr1 do statement
                                 |        begin  statements end
expr1               -->          expression
                                 |        expression < expression
                                 |        expression = expression
                                 |        expression > expression
expression          ->           term expression'
expression'         ->           + term expression'
                                 |        - term expression'
                                 |        epsilon
term                ->           factor term'
term'               ->           * factor term'
                                 |        / factor term'
                                 |        epsilon
factor              ->           NUM_OR_ID
                                 |        ( expression )
```

Main.c-

```
#include <stdio.h>
#include <stdlib.h>
#include "code_gen.c"

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

Code_gen.c-

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "lex.c"
#include "name.c"
#include "code_gen.h"
#include "symtab.c"
#include "labelstack.c"

int label = 0;

char *mapper(char *tempvar) {
   int index = 7 - (tempvar[1] - '0');
   return Registers[index];
}

void statements(){
   /*  statements -> statement statements | statement  */
   inter = fopen("main.ic", "w");
   assembly = fopen("main.asm", "w");
   fprintf(assembly, "ORG 100H\n");

   while(!match(EOI)){
       statement();
   }
   fclose(inter);
```

```
    map *temp = head;
    fprintf(assembly, "RET\n");
    while(temp != NULL){
        fprintf(assembly, "%s DB ?\n", temp->var);
        temp = temp->next;
    }
    fprintf(assembly, "END\n");
    fclose(assembly);
}

void statement(){
    char *tempvar;
    char var[100];
    int i=0;
    while(i<yyleng){
        var[i] = *(yytext+i);
        i++;
    }
    var[i] = '\0';
    if(match(ID)){
        advance();
        if(match(ASS)){
            advance();
            tempvar = expr1(0);
            if(!match(SEMI)){
                fprintf(stderr, "%d: ';' expected\n", yylineno);
            }
            else{
                char var2[100];
                sprintf(var2, "_%s", var);
                insert(var2);

                fprintf(inter, "    %s <- %s\n", var2, tempvar);
                fprintf(assembly, "MOV %s, %s\n", var2,
mapper(tempvar));
                advance();
            }
            freename(tempvar);
        }
    }
```

```
else if(match(WHILE)){
    advance();
    label++;
    push(label);
    fprintf(assembly, "WHILELABEL%d: \n", top->data);
    tempvar = expr1(1);
    if(match(DO)){
        advance();
        fprintf(inter, "while (%s) do { \n", tempvar);
        statement();
        fprintf(inter, "\n}");
        fprintf(assembly, "JMP WHILELABEL%d\n", top->data);
        fprintf(assembly, "LABEL%d: \n", top->data);
        pop();
    }
    freename(tempvar);
}
else if(match(IF)){
    advance();
    label++;
    push(label);
    tempvar = expr1(1);
    if(match(THEN)){
        advance();
        fprintf(inter, "if (%s) then { \n", tempvar);
        statement();
        fprintf(inter, "\n}");
        fprintf(assembly, "LABEL%d: \n", top->data);
        pop();
    }
    freename(tempvar);
}
else if(match(BEG)){
    advance();
    fprintf(inter, "begin { \n");
    while (!match(END)){
        statement();
    }
    fprintf(inter, "\n} end \n");
    advance();
```

```
   }
   else {
       fprintf(stderr, "%d: Syntax error\n", yylineno);
       terminate();
   }
}

char *expr1(int flag){
   char *tempvar, *tempvar1, *tempvar2;
   tempvar = expression();
   // advance();
   if(match(EQU) && flag > 0){
       advance();
       tempvar2 = newname();
       tempvar1 = expression();
       fprintf(inter, "    %s <- %s = %s\n", tempvar2, tempvar,
tempvar1);
       fprintf(assembly, "CMP %s, %s\n", mapper(tempvar),
mapper(tempvar1));
       fprintf(assembly, "JNE LABEL%d\n", top->data);
       freename(tempvar);
       freename(tempvar1);
       return tempvar2;
   }
   else if(match(LT) && flag > 0){
       advance();
       tempvar2 = newname();
       tempvar1 = expression();
       fprintf(inter, "    %s <- %s < %s\n", tempvar2, tempvar,
tempvar1);
       fprintf(assembly, "CMP %s, %s\n", mapper(tempvar),
mapper(tempvar1));
       fprintf(assembly, "JGE LABEL%d\n", top->data);
       freename(tempvar);
       freename(tempvar1);
       return tempvar2;
   }
   else if(match(GT) && flag > 0){
       advance();
       tempvar2 = newname();
       tempvar1 = expression();
       fprintf(inter, "    %s <- %s > %s\n", tempvar2, tempvar,
tempvar1);
       fprintf(assembly, "CMP %s, %s\n", mapper(tempvar),
mapper(tempvar1));
       fprintf(assembly, "JLE LABEL%d\n", top->data);
       freename(tempvar);
       freename(tempvar1);
       return tempvar2;
   }
   if (flag > 0) {
       fprintf(assembly, "CMP %s, 0\n", mapper(tempvar));
       fprintf(assembly, "JLE LABEL%d\n", top->data);
   }
   return tempvar;
}

char *expression(){
   /* expression -> term expression'
    * expression' -> PLUS term expression' |  epsilon
    */
   char  *tempvar, *tempvar1;
   tempvar = term();
   while(1){
       if(match(PLUS)){
           advance();
           tempvar1 = term();
           fprintf(inter, "    %s += %s\n", tempvar, tempvar1);
           fprintf(assembly, "ADD %s, %s\n", mapper(tempvar),
mapper(tempvar1));
           freename(tempvar1);
       }
       else if(match(MINUS)){
           advance();
           tempvar1 = term();
           fprintf(inter, "    %s -= %s\n", tempvar, tempvar1);
           fprintf(assembly, "SUB %s, %s\n", mapper(tempvar),
mapper(tempvar1));
           freename(tempvar1);
       }
```

```
        else break;
    }
    return tempvar;
}
char *term() {
    char  *tempvar, *tempvar1 ;
    tempvar = factor();

    int flag = 0;
    if (match(TIMES) || match(DIV)) {
        flag = 1;
    }

    if (flag > 0 && strcmp(mapper(tempvar),"AL") != 0) {
        fprintf(assembly, "XCHG AL, %s\n", mapper(tempvar));
    }
    while(1){
        if(match(TIMES)){
            advance();
            tempvar1 = term();
            fprintf(inter, "    %s *= %s\n", tempvar, tempvar1);
            fprintf(assembly, "MOV AH, 0\n");
            fprintf(assembly, "MUL %s\n", mapper(tempvar1));
            freename(tempvar1);
        }
        else if(match(DIV)){
            advance();
            tempvar1 = term();
            fprintf(inter, "    %s /= %s\n", tempvar, tempvar1);
            fprintf(assembly, "MOV AH, 0\n");
            fprintf(assembly, "DIV %s\n", mapper(tempvar1));
            freename(tempvar1);
        }
        else break;
    }
    if (flag > 0 && strcmp(mapper(tempvar),"AL") != 0) {
        fprintf(assembly, "XCHG AL, %s\n", mapper(tempvar));
    }
    return tempvar;
}
```

```
char *factor(){
   char *tempvar;

   if(match(NUM) || match(ID)){
    /* Print the assignment instruction. The %0.*s conversion is a
form of
     * %X.Ys, where X is the field width and Y is the maximum
number of
     * characters that will be printed (even if the string is
longer). I'm
     * using the %0.*s to print the string because it's not \0
terminated.
     * The field has a default width of 0, but it will grow the
size needed
     * to print the string. The ".*" tells printf() to take the
maximum-
     * number-of-characters count from the next argument (yyleng).
     */
       char var[100];
       int i=0;
       var[0] = '_';
       while(i<yyleng){
           var[i+1] = *(yytext+i);
           i++;
       }
       var[i+1] = '\0';
       map *temp = search(var);
       if (match(ID) && temp == NULL) {
           fprintf(stderr, "%d: Undeclared identifier\n",
yylineno);
       }

       if (match(ID)) {
           fprintf(inter, "    %s = %s\n", tempvar = newname(),
var);
           fprintf(assembly, "MOV %s, %s\n", mapper(tempvar),
var);
       }
       else if (match(NUM)) {
           fprintf(inter, "    %s = %s\n", tempvar = newname(),
(var+1));
           fprintf(assembly, "MOV %s, %s\n", mapper(tempvar),
(var+1));
       }
       advance();
   }
   else if(match(LP)){
       advance();
       tempvar = expression();
       if(match(RP))
           advance();
       else
           fprintf(stderr, "%d: Mismatched parenthesis\n",
yylineno);
   }
   else
        fprintf(stderr, "%d: Number or identifier expected\n",
yylineno);
   return tempvar;
}
```

Code_gen.h-

```
extern char *newname();
extern void freename(char *name);

void statements();
void statement();
char *expr1(int );
char *expression();
char *term();
char *factor();
```

Lex.h-

```
#define EOI 0      /* ~ */
#define SEMI 1      /* ; */
#define PLUS 2      /* + */
#define MINUS 3     /* - */
```

```
#define TIMES 4     /* * */
#define DIV 5      /* / */
#define GT 6       /* > */
#define LT 7       /* < */
#define EQU 8      /* = */
#define LP 9       /* ( */
#define RP 10      /* ) */
#define IF 11      /* if condition */
#define THEN 12     /* then */
#define ELSE 13     /* else */
#define WHILE 14    /* while loop */
#define DO 15      /* do */
#define BEG 16      /* begin */
#define END 17      /* end */
#define NUM 18  /* Decimal Number or Identifier */
#define ASS 19      /* assignment operator */
#define ID 20

extern char *yytext;
extern int yyleng;
extern int yylineno;
```

Lex.c-

```
#include "lex.h"
#include <stdio.h>
#include <string.h>
#include <ctype.h>

char *yytext = ""; /* Lexeme (not '\0' terminated) */
int yyleng = 0;    /* Lexeme length */
int yylineno = 0;  /* Input line number */


void terminate(){
   remove("main.ic");
   remove("main.asm");
   exit(0);
}

int lex(void){
```

```
static char input_buffer[1024];
char *current;

current = yytext + yyleng; /* Skip current lexeme */

while(1){                 /* Get the next one */
    while(!*current ){
        /* Get new lines, skipping any leading
        * white space on the line,
        * until a nonblank line is found.
        */
        current = input_buffer;
        if(!fgets(input_buffer, 1024, stdin)){
            *current = '\0' ;
            return EOI;
        }
        ++yylineno;
        while(isspace(*current))
            ++current;
    }
    for(; *current; ++current){ /* Get the next token */

        yytext = current;
        yyleng = 1;
        switch( *current ){
        case ';':
            return SEMI;
        case '+':
            return PLUS;
        case '-':
            return MINUS;
        case '*':
            return TIMES;
        case '/':
            return DIV;
        case '>':
            return GT;
        case '<':
            return LT;
        case '=':
                return EQU;
            case '(':
                return LP;
            case ')':
                return RP;
            case '\n':
            case '\t':
            case ' ' :
                break;
            default:
                if(!isalnum(*current)){
                    char arr[20];
                    int i=0;
                    while(i<2){
                        arr[i] = *current;
                        ++current;
                        i++;
                    }
                    arr[i] = '\0';
                    if(strcmp(arr, ":=") == 0){
                        yyleng = 2;
                        return ASS;
                    }
                    fprintf(stderr, "%d: Syntax error\n",
yylineno);
                    terminate();
                }
                else{
                    char arr[20];
                    int i=0;
                    while(isalnum(*current)){
                        arr[i] = *current;
                        ++current;
                        i++;
                    }
                    arr[i] = '\0';
                    i = 0;
                    if (isdigit(arr[0])) {
                        while (arr[i] != '\0') {
                            if (!isdigit(arr[i])) {
                                fprintf(stderr, "%d: Invalid
 identifier name\n", yylineno);
                                terminate();
                            }
                            i++;
                        }
                        yyleng = current - yytext;
                        return NUM;
                    }
                    if(strcmp(arr, "if") == 0){
                        yyleng = 2;
                        return IF;
                    }
                    if(strcmp(arr, "then") == 0){
                        yyleng = 4;
                        return THEN;
                    }
                    if(strcmp(arr, "else") == 0){
                        yyleng = 4;
                        return ELSE;
                    }
                    if(strcmp(arr, "while") == 0){
                        yyleng = 5;
                        return WHILE;
                    }
                    if(strcmp(arr, "do") == 0){
                        yyleng = 2;
                        return DO;
                    }
                    if(strcmp(arr, "begin") == 0){
                        yyleng = 5;
                        return BEG;
                    }
                    if(strcmp(arr, "end") == 0){
                        yyleng = 3;
                        return END;
                    }
                    yyleng = current - yytext;
                    return ID;
                }
```

```
                break;
            }
        }
    }
}

static int Lookahead = -1; /* Lookahead token  */

int match(int token){
   /* Return true if "token" matches the current lookahead symbol.
*/
   if(Lookahead == -1)
       Lookahead = lex();
   return token == Lookahead;
}

void advance(void){
/* Advance the lookahead to the next input symbol. */
   Lookahead = lex();
}
```

Name.c-

```
#include <stdio.h>

char  *Names[] = { "t0", "t1", "t2", "t3", "t4", "t5", "t6", "t7"
};
char  *Registers[] = { "AL", "AH", "BL", "BH", "CL", "CH", "DL",
"DH" };
char  **Namep  = Names;

char *newname(){
   if(Namep >= &Names[sizeof(Names)/sizeof(*Names)]){
       fprintf(stderr, "%d: Expression too complex\n", yylineno);
       exit(1);
   }
   return(*Namep++);
}

void freename(char *s){
   if(Namep > Names)
```

```
       *--Namep = s;
   else
       fprintf(stderr, "%d: (Internal error) Name stack
underflow\n", yylineno);
}
```

Symtab.c-

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

FILE *inter, *assembly;

typedef struct _map{
   char var[20];
   struct _map *next;
} map;

map *head = NULL;
map *last = NULL;

map *create_node(char *arr){
   map *entry = (map*)malloc(sizeof(map));
   entry->next = NULL;
   strcpy(entry->var, arr);
   return entry;
}


map *search(char *entry){
   map *temp = head;
   while(temp != NULL){
       if(strcmp(temp->var, entry) == 0){
           return temp;
       }
       temp = temp->next;
   }
   return NULL;
}
```

```
void insert(char *arr){
   map *temp = search(arr);
   if(temp==NULL){
       if (head == NULL) {
           map *entry = create_node(arr);
           head = entry;
           last = head;
       }
       else {
           map *entry = create_node(arr);
           last->next = entry;
           last = last->next;
       }
   }
}
```

Labelstack.c-

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* link;
};
struct Node* top = NULL;

void push(int data){
    struct Node* temp;
    temp = (struct Node*)malloc(sizeof(struct Node));
    if (!temp) {
        exit(1);
    }
    temp->data = data;
    temp->link = top;
    top = temp;
}

void pop(){
    struct Node* temp;
    if (top == NULL) {
        printf("\nStack Underflow");
        exit(1);
    }
    else {
        temp = top;
        top = top->link;
        temp->link = NULL;
        free(temp);
    }
}
```

Improved_parser.c-

```
/* Revised parser  */

#include <stdio.h>
#include "lex.h"

void factor (void);
void term (void);
void expression (void);

statements(){
   /*  statements -> expression SEMI |  expression SEMI statements
*/
   while(!match(EOI)){
       expression();

       if(match(SEMI))
           advance();
       else
           fprintf(stderr, "%d: Inserting missing semicolon\n",
yylineno);
   }
}

void expression(){
   /* expression  -> term expression'
    * expression' -> PLUS term expression' |  epsilon
    */
   if( !legal_lookahead(NUM,ID, LP, 0))
```

```
        return;
   term();
   while(match(PLUS)){
       advance();
       term();
   }
}

void term(){
   if(!legal_lookahead(NUM,ID, LP, 0 ))
        return;
   factor();
   while(match(TIMES)){
       advance();
       factor();
   }
}

void factor(){
   if(!legal_lookahead(NUM,ID, LP, 0 ))
        return;
   if(match(NUM)|| match(ID))
       advance();
   else if(match(LP)){
       advance();
       expression();
       if(match(RP))
           advance();
       else
           fprintf( stderr, "%d: Mismatched parenthesis\n",
yylineno );
   }
   else
        fprintf( stderr, "%d: Number or identifier expected\n",
yylineno );
}

#include <stdarg.h>

#define MAXFIRST 16
```

```
#define SYNCH   SEMI

int legal_lookahead(  first_arg )
int first_arg;
{
   /* Simple error detection and recovery. Arguments are a
0-terminated list of
    * those tokens that can legitimately come next in the input.
If the list is
    * empty, the end of file must come next. Print an error
message if
    * necessary. Error recovery is performed by discarding all
input symbols
    * until one that's in the input list is found
    *
    * Return true if there's no error or if we recovered from the
error,
    * false if we can't recover.
    */

   va_list args;
   int tok;
   int lookaheads[MAXFIRST], *p = lookaheads, *current;
   int error_printed = 0;
   int rval = 0;

   va_start(args, first_arg);

   if(!first_arg){
       if(match(EOI)){
           rval = 1;
       }
       else{
           *p++ = first_arg;
           while((tok = va_arg(args, int)) &&
p<&lookaheads[MAXFIRST]){
               *p++ = tok;
           }
           while(!match(SYNCH)){
               for(current=lookaheads; current<p ; ++current){
                   if(match(*current)){
                       rval = 1;
                       goto exit;
                   }
               }
               if(!error_printed){
                   fprintf( stderr, "Line %d: Syntax error\n",
yylineno );
                   error_printed = 1;
               }
               advance();
           }
       }
   }

   exit:
       va_end(args);
       return rval;
}
```

Basic_parser.c-

```
/* Basic parser, shows the structure but there's no code
generation */

#include <stdio.h>
#include "lex.h"

statements(){
   /*  statements -> expression SEMI
    *             |  expression SEMI statements
    */
   expression();
   if(match(SEMI))
        advance();
   else
       fprintf(stderr, "%d: Inserting missing semicolon\n",
yylineno);
   if( !match(EOI) )
       statements();           /* Do another statement. */
}
```

```
expression(){
   /* expression -> term expression' */
   term();
   expr_prime();
}

expr_prime(){
   /* expression' -> PLUS term expression'
    *              | epsilon
    */
   if(match(PLUS)){
       advance();
       term();
       expr_prime();
   }
}

term(){
   /* term -> factor term' */
   factor();
   term_prime();
}

term_prime(){
   /* term' -> TIMES factor term'
    *       |   epsilon
    */
   if(match(TIMES)){
       advance();
       factor();
       term_prime();
   }
}

factor(){
   /* factor   ->    NUM_OR_ID
    *          |     LP expression RP
    */
   if(match(ID) || match(NUM))
```

```
          advance();

      else if(match(LP)){
          advance();
          expression();
          if( match(RP) )
              advance();
          else
              fprintf( stderr, "%d: Mismatched parenthesis\n",
yylineno);
      }
      else
           fprintf( stderr, "%d Number or identifier expected\n",
yylineno );
}
```