软件学院 **2022-2023** 学年度第一学期

《软件设计综合实践》实验报告

班级：	
学号：	
姓名：	

目录

1. 实验内容和要求

基本内容（基本评分要求）：

1. 用 C--语言写一个函数计算第 n（从标准输入获得）个斐波那契数，并将计算结果输出到屏幕；
2. 用 C 语言设计和实现 C--语言的词法分析器，并对输入的 C--代码输出词法分析扫描结果;
3. 书写 C--语言的 Lex 输入文件，并用 Lex 生成 C--语言的词法分析器；
4. 参照实验参考书中附录 A 中给出的 C Minus 语言的 BNF 语法，给出 C--语言的 BNF 语法描述；
5. 用 C 语言设计和实现 C--语言的递归下降语法分析器，并对输入的 C--代码输出语法树。

可选内容（加分环节，不强制要求）：

1. 用 C 语言设计和实现对 C--语言进行符号表构造和类型检查的语义分析程序。

2. 斐波拉契数代码

```
int fibonacci(int n)
{
    int cnt;
    int firstFib;
    int secondFib;
    int fib;

    firstFib = 1;
    secondFib = 1;
    cnt = 2; /* n = 1 或 n = 2 时特判 */

    if (n == 1)
        return 1;
    else if (n == 2)
        return 1;
    else
    {
        while (cnt < n)
        {
            fib = firstFib + secondFib;
            firstFib = secondFib;
            secondFib = fib;
            cnt = cnt + 1;
```

```
        }
    }
    return fib;
}

void main(void)
{
    int n;
    n = input();
    output(fibonacci(n));
}
```

3. 词法分析程序

3.1 主要设计和实现思路

采用 enum 进行状态码的描述，具体如下： START 代表开始状态；INNUM 代表输入数字；INID 代表输入 identifier；INEQ 代表第一次输入 = , 后续可能是 = 或 ==；INLT 代表输入 < , 后续可能是 < 或 <=；INGT 代表输入 > , 后续可能是 > 或 >=；INDIV 代表输入 / ；INNE 代表输入 ! , 后续可能是 ! 或 ！ =；INCOMMENT 代表输入 /*；ENDCOMMENT 代表输入 */；DONE 代表结束状态

```
typedef enum
{
  START,        // 开始状态
  INNUM,        // 输入数字
  INID,         // 输入 identifier
  INEQ,         // 第一次输入 = , 可能是 = 或 ==
  INLT,         // 输入 < ,
  INGT,         // 输入 >
  INDIV,        // 输入 /
  INNE,         // 输入 !
  INCOMMENT,    // 输入 /*
  ENDCOMMENT,   // 输入 */
  DONE          // 结束状态
} StateType;
```

struct reservedWords 结构用来存储预置词，包含有"if" 、 "else" 、"while" 、"int" 、"void" 、 "return" 六个预置词。

```
static struct
{
  char *str;
  TokenType tok;
```

```
} reservedWords[MAXRESERVED] = {{(char *)"if", IF}, {(char *)"else", ELSE},
                               {(char *)"while", WHILE}, {(char *)"int", INT},
                               {(char *)"void", VOID}, {(char *)"return", RETURN}};
```

static int getNextChar(void) 用于读取下个字符， static int ungetNextChar(void)函数用于撤销上述步骤。

TokenType getToken(void) 函数用于获取当前的 token：

- 如果状态码是 START，进行一次 getNextChar()：
  - 对于一元操作符，如 '+', '-', '*', '(', ')', ';', '{', '}' 等字符，直接给出 currentToken
  - 对于可能存在的二元（多元）操作符，如"= || ==", "< || <=", "> || >=", "! || !=", "/ || /*"等状态，给定对应的状态码，进行进一步的判断
  - break；
- 如果状态码是 INEQ，则读取字符并判断是=还是==
- 如果状态码是 INLT，则读取字符并判断是<还是<=
- 如果状态码是 INGT，则读取字符并判断是>还是>=
- 如果状态码是 INNE，则读取字符并判断是!还是!=
- 如果状态码是 INDIV，则读取字符并判断是/*还是/*
  - 为了防止注释中可能的*进行干扰，采用 INCOMMENT 和 ENDCOMMENT 两种状态进行判断
- 如果状态码是 INNUM 或者 INID，则进行读取
- 否则，输出错误信息
- 随后，对新的 ID 进行存储。
- 如果 TraceScan == true ，则调用 UTIL.c 中的 printToken()函数，输出词法分析的结果
- 返回当前的 Token

调用 printToken()函数进行 token 的打印，函数如下：

```
void printToken(TokenType token, const char* lexeme)
{
    switch(token)
    {
        case IF:
        case ELSE:
        case INT:
        case RETURN:
        case VOID:
        case WHILE:
              fprintf(listing, "reserved word \"%s\"", lexeme);
              break;
        case PLUS:    fprintf(listing, "+"); break;
        case MINUS:   fprintf(listing, "-"); break;
        case TIMES:   fprintf(listing, "*"); break;
        case DIVIDE:  fprintf(listing, "/"); break;
        case LT:      fprintf(listing, "<"); break;
        case GT:      fprintf(listing, ">"); break;
        case ASSIGN:  fprintf(listing, "="); break;
        case NE:      fprintf(listing, "!=");break;
        case SEMI:    fprintf(listing, ";"); break;
        case COMMA:   fprintf(listing, ","); break;
        case LPAREN:  fprintf(listing, "("); break;
        case RPAREN:  fprintf(listing, ")"); break;
        case LBRACE:  fprintf(listing, "{"); break;
        case RBRACE:  fprintf(listing, "}"); break;
        case LTE:     fprintf(listing, "<="); break;
        case GTE:     fprintf(listing, ">="); break;
        case EQ:      fprintf(listing, "=="); break;
        case NUM:
              fprintf(listing, "NUM, value = %s", lexeme);
              break;
        case ID:
              fprintf(listing, "ID, name = \"%s\"", lexeme);
              break;
        case ENDOFFILE:
              fprintf(listing, "EOF");
              break;
        case ERROR:
              fprintf(listing, "<<<ERROR>>> %s", lexeme);
              break;
        default:
              fprintf(listing, "<<<UNKNOWN TOKEN>>> %d", token);
    }
}
```

3.2 词法分析程序代码

```c
#include "globals.h"
#include "util.h"
#include "scan.h"

/* states in scanner DFA */
typedef enum
{
  START,
  INNUM,     // input number
  INID,      // input identifier
  INEQ,      // input = , maybe = or ==
  INLT,      // input < ,
  INGT,      // input >
  INDIV,      // input /
  INNE,      // input !
  INCOMMENT,      // input /*
  ENDCOMMENT,      // input */
  DONE
} StateType;

/* lexeme of identifier or reserved word */
char tokenString[MAXTOKENLEN + 1];

/* BUFLEN = length of the input buffer for source code lines */
#define BUFLEN 256

static char lineBuf[BUFLEN]; /* holds the current line */
static int linepos = 0;      /* current position in LineBuf */
static int bufsize = 0;      /* current size of buffer string */
static int EOF_flag = FALSE; /* corrects ungetNextChar behavior on EOF */

/* getNextChar fetches the next non-blank character from lineBuf, reading in a new line
 * if lineBuf is exhausted */
static int getNextChar(void)
{
  if (!(linepos < bufsize))
  {
    lineno++;
    if (fgets(lineBuf, BUFLEN - 1, source))
    {
      if (EchoSource)
        fprintf(listing, "%4d: %s", lineno, lineBuf);
      bufsize = strlen(lineBuf);
      linepos = 0;
      return lineBuf[linepos++];
    }
    else
    {
      EOF_flag = TRUE;
```

```
      return EOF;
    }
  }
  else
    return lineBuf[linepos++];
}

/* ungetNextChar backtracks one character in lineBuf */
static void ungetNextChar(void)
{
  if (!EOF_flag)
    linepos--;
}

/* lookup table of reserved words */
static struct
{
  char *str;
  TokenType tok;
} reservedWords[MAXRESERVED] = {{(char *)"if", IF}, {(char *)"else", ELSE},
                                {(char *)"while", WHILE}, {(char *)"int", INT},
                                {(char *)"void", VOID}, {(char *)"return", RETURN}};

/* lookup an identifier to see if it is a reserved word */
/* uses linear search */
static TokenType reservedLookup(char *s)
{
  int i;
  for (i = 0; i < MAXRESERVED; i++)
    if (!strcmp(s, reservedWords[i].str))
      return reservedWords[i].tok;
  return ID;
}

/* function getToken returns the next token in source file */
TokenType getToken(void)
{ /* index for storing into tokenString */
  int tokenStringIndex = 0;
  /* holds current token to be returned */
  TokenType currentToken;
  /* current state - always begins at START */
  StateType state = START;
  /* flag to indicate save to tokenString */
  int save;
  while (state != DONE)
  {
    int c = getNextChar();
    save = TRUE;
    switch (state)
    {
    case START:
      if (isdigit(c))
        state = INNUM;
```

```
else if (isalpha(c))
  state = INID;
else if (c == '=')
  state = INEQ;
else if (c == '<')
  state = INLT;
else if (c == '>')
  state = INGT;
else if (c == '!')
  state = INNE;
else if (c == '/')
  state = INDIV;
else if ((c == ' ') || (c == '\t') || (c == '\n'))
  save = FALSE;
else
{
  state = DONE;
  switch (c)
  {
    case EOF:
      save = FALSE;
      currentToken = ENDFILE;
      break;
    case '+':
      currentToken = PLUS;
      break;
    case '-':
      currentToken = MINUS;
      break;
    case '*':
      currentToken = TIMES;
      break;
    case '(':
      currentToken = LPAREN;
      break;
    case ')':
      currentToken = RPAREN;
      break;
    case ';':
      currentToken = SEMI;
      break;
    case '{':
      currentToken = LBRACE;
      break;
    case '}':
      currentToken = RBRACE;
      break;
    default:
      currentToken = ERROR;
      break;
  }
}
break;
```

```
case INEQ:
  state = DONE;
  if (c == '=')
    currentToken = EQ;
  else
  { /* backup in the input */
    ungetNextChar();
    save = FALSE;
    currentToken = ASSIGN;
  }
  break;
case INLT:
  state = DONE;
  if (c == '=')
    currentToken = LTE;
  else
  {
    ungetNextChar();
    save = FALSE;
    currentToken = LT;
  }
  break;
case INGT:
  state = DONE;
  if (c == '=')
    currentToken = GTE;
  else
  {
    ungetNextChar();
    save = FALSE;
    currentToken = GT;
  }
  break;
case INNE:
  state = DONE;
  if (c == '=')
    currentToken = NEQ;
  else
  {
    ungetNextChar();
    save = FALSE;
    currentToken = ERROR;
  }
  break;
case INDIV:
  if (c == '*')
  {
    save = FALSE;
    state = INCOMMENT;
    tokenStringIndex -= 1;
  }
  else
```

```
  {
    ungetNextChar();
    save = FALSE;
    state = DONE;
    currentToken = DIVIDE;
  }
  break;

case INCOMMENT:
  save = FALSE;
  if(c == '*')
    state = ENDCOMMENT;
  break;

case ENDCOMMENT:
  save = FALSE;
  if (c == '/')
    state = START;
  else if (c == '*')
    state = ENDCOMMENT;
  else
    state = INCOMMENT;
  break;

case INNUM:
  if (!isdigit(c))
  { /* backup in the input */
    ungetNextChar();
    save = FALSE;
    state = DONE;
    currentToken = NUM;
  }
  break;
case INID:
  if (!isalpha(c))
  { /* backup in the input */
    ungetNextChar();
    save = FALSE;
    state = DONE;
    currentToken = ID;
  }
  break;
case DONE:
default: /* should never happen */
  fprintf(listing, "Scanner Bug: state= %d\n", state);
  state = DONE;
  currentToken = ERROR;
  break;
}
// append new id to token string
if ((save) && (tokenStringIndex <= MAXTOKENLEN))
  tokenString[tokenStringIndex++] = (char)c;
if (state == DONE)
```

```
    {
      tokenString[tokenStringIndex] = '\0';
      if (currentToken == ID)
        currentToken = reservedLookup(tokenString);
    }
  }
  if (TraceScan)
  {
    fprintf(listing, "\t%d: ", lineno);
    printToken(currentToken, tokenString);
  }
  return currentToken;
} /* end getToken */
```

3.3 实验演示

1. 求三个整数中的最大值

```
in Line:  2: get reserved word "int"
in Line:  2: get ID, name = "max"
in Line:  2: get (
in Line:  2: get reserved word "int"
in Line:  2: get ID, name = "x"
in Line:  2: get ,
in Line:  2: get reserved word "int"
in Line:  2: get ID, name = "y"
in Line:  2: get ,
in Line:  2: get reserved word "int"
in Line:  2: get ID, name = "z"
in Line:  2: get )
in Line:  3: get {
in Line:  4: get reserved word "int"
in Line:  4: get ID, name = "biggest"
in Line:  4: get ;
in Line:  5: get ID, name = "biggest"
in Line:  5: get =
in Line:  5: get ID, name = "x"
in Line:  5: get ;
in Line:  6: get reserved word "if"
in Line:  6: get (
in Line:  6: get ID, name = "y"
in Line:  6: get >
in Line:  6: get ID, name = "biggest"
in Line:  6: get )
in Line:  7: get ID, name = "biggest"
in Line:  7: get =
in Line:  7: get ID, name = "y"
in Line:  7: get ;
in Line:  8: get reserved word "if"
in Line:  8: get (
```

```
in Line:  8: get ID, name = "z"
in Line:  8: get >
in Line:  8: get ID, name = "biggest"
in Line:  8: get )
in Line:  9: get ID, name = "biggest"
in Line:  9: get =
in Line:  9: get ID, name = "z"
in Line:  9: get ;
in Line: 10: get reserved word "return"
in Line: 10: get ID, name = "biggest"
in Line: 10: get ;
in Line: 11: get }
in Line: 12: get reserved word "void"
in Line: 12: get ID, name = "main"
in Line: 12: get (
in Line: 12: get reserved word "void"
in Line: 12: get )
in Line: 13: get {
in Line: 14: get reserved word "int"
in Line: 14: get ID, name = "x"
in Line: 14: get ;
in Line: 15: get reserved word "int"
in Line: 15: get ID, name = "y"
in Line: 15: get ;
in Line: 16: get reserved word "int"
in Line: 16: get ID, name = "z"
in Line: 16: get ;
in Line: 17: get reserved word "int"
in Line: 17: get ID, name = "biggest"
in Line: 17: get ;
in Line: 18: get ID, name = "x"
in Line: 18: get =
in Line: 18: get ID, name = "input"
in Line: 18: get (
in Line: 18: get )
in Line: 18: get ;
in Line: 19: get ID, name = "y"
in Line: 19: get =
in Line: 19: get ID, name = "input"
in Line: 19: get (
in Line: 19: get )
in Line: 19: get ;
in Line: 20: get ID, name = "z"
in Line: 20: get =
in Line: 20: get ID, name = "input"
in Line: 20: get (
in Line: 20: get )
in Line: 20: get ;
in Line: 21: get ID, name = "biggest"
in Line: 21: get =
in Line: 21: get ID, name = "max"
in Line: 21: get (
in Line: 21: get ID, name = "x"
```

```
in Line: 21: get ,
in Line: 21: get ID, name = "y"
in Line: 21: get ,
in Line: 21: get ID, name = "z"
in Line: 21: get )
in Line: 21: get ;
in Line: 22: get ID, name = "output"
in Line: 22: get (
in Line: 22: get ID, name = "biggest"
in Line: 22: get )
in Line: 22: get ;
in Line: 23: get }
in Line: 24: get EOF
```

2. 给定 N，求 1 到 N 之和

```
in Line:  1: get reserved word "int"
in Line:  1: get ID, name = "sum"
in Line:  1: get (
in Line:  1: get reserved word "int"
in Line:  1: get ID, name = "n"
in Line:  1: get )
in Line:  2: get {
in Line:  3: get reserved word "int"
in Line:  3: get ID, name = "result"
in Line:  3: get ;
in Line:  4: get reserved word "int"
in Line:  4: get ID, name = "i"
in Line:  4: get ;
in Line:  5: get ID, name = "i"
in Line:  5: get =
in Line:  5: get NUM, value = 1
in Line:  5: get ;
in Line:  6: get ID, name = "result"
in Line:  6: get =
in Line:  6: get NUM, value = 0
in Line:  6: get ;
in Line:  7: get reserved word "while"
in Line:  7: get (
in Line:  7: get ID, name = "i"
in Line:  7: get <=
in Line:  7: get ID, name = "n"
in Line:  7: get )
in Line:  8: get {
in Line:  9: get ID, name = "result"
in Line:  9: get =
in Line:  9: get ID, name = "result"
in Line:  9: get +
in Line:  9: get ID, name = "i"
in Line:  9: get ;
in Line: 10: get ID, name = "i"
in Line: 10: get =
in Line: 10: get ID, name = "i"
```

```
in Line: 10: get +
in Line: 10: get NUM, value = 1
in Line: 10: get ;
in Line: 11: get }
in Line: 12: get reserved word "return"
in Line: 12: get ID, name = "result"
in Line: 12: get ;
in Line: 13: get }
in Line: 15: get reserved word "void"
in Line: 15: get ID, name = "main"
in Line: 15: get (
in Line: 15: get reserved word "void"
in Line: 15: get )
in Line: 16: get {
in Line: 17: get reserved word "int"
in Line: 17: get ID, name = "n"
in Line: 17: get ;
in Line: 18: get reserved word "int"
in Line: 18: get ID, name = "s"
in Line: 18: get ;
in Line: 19: get ID, name = "n"
in Line: 19: get =
in Line: 19: get ID, name = "intput"
in Line: 19: get (
in Line: 19: get )
in Line: 19: get ;
in Line: 20: get ID, name = "s"
in Line: 20: get =
in Line: 20: get ID, name = "sum"
in Line: 20: get (
in Line: 20: get ID, name = "n"
in Line: 20: get )
in Line: 20: get ;
in Line: 21: get ID, name = "output"
in Line: 21: get (
in Line: 21: get ID, name = "s"
in Line: 21: get )
in Line: 21: get ;
in Line: 22: get }
in Line: 23: get EOF
```

3. 计算第 n 个斐波那契数

```
in Line:  1: get reserved word "int"
in Line:  1: get ID, name = "fibonacci"
in Line:  1: get (
in Line:  1: get reserved word "int"
in Line:  1: get ID, name = "n"
in Line:  1: get )
in Line:  2: get {
in Line:  3: get reserved word "int"
in Line:  3: get ID, name = "cnt"
in Line:  3: get ;
```

```
in Line:  4: get reserved word "int"
in Line:  4: get ID, name = "firstFib"
in Line:  4: get ;
in Line:  5: get reserved word "int"
in Line:  5: get ID, name = "secondFib"
in Line:  5: get ;
in Line:  6: get reserved word "int"
in Line:  6: get ID, name = "fib"
in Line:  6: get ;
in Line:  8: get ID, name = "firstFib"
in Line:  8: get =
in Line:  8: get NUM, value = 1
in Line:  8: get ;
in Line:  9: get ID, name = "secondFib"
in Line:  9: get =
in Line:  9: get NUM, value = 1
in Line:  9: get ;
in Line: 10: get ID, name = "cnt"
in Line: 10: get =
in Line: 10: get NUM, value = 2
in Line: 10: get ;
in Line: 12: get reserved word "if"
in Line: 12: get (
in Line: 12: get ID, name = "n"
in Line: 12: get ==
in Line: 12: get NUM, value = 1
in Line: 12: get )
in Line: 13: get reserved word "return"
in Line: 13: get NUM, value = 1
in Line: 13: get ;
in Line: 14: get reserved word "else"
in Line: 14: get reserved word "if"
in Line: 14: get (
in Line: 14: get ID, name = "n"
in Line: 14: get ==
in Line: 14: get NUM, value = 2
in Line: 14: get )
in Line: 15: get reserved word "return"
in Line: 15: get NUM, value = 1
in Line: 15: get ;
in Line: 16: get reserved word "else"
in Line: 17: get {
in Line: 18: get reserved word "while"
in Line: 18: get (
in Line: 18: get ID, name = "cnt"
in Line: 18: get <
in Line: 18: get ID, name = "n"
in Line: 18: get )
in Line: 19: get {
in Line: 20: get ID, name = "fib"
in Line: 20: get =
in Line: 20: get ID, name = "firstFib"
in Line: 20: get +
```

```
in Line: 20: get ID, name = "secondFib"
in Line: 20: get ;
in Line: 21: get ID, name = "firstFib"
in Line: 21: get =
in Line: 21: get ID, name = "secondFib"
in Line: 21: get ;
in Line: 22: get ID, name = "secondFib"
in Line: 22: get =
in Line: 22: get ID, name = "fib"
in Line: 22: get ;
in Line: 23: get ID, name = "cnt"
in Line: 23: get =
in Line: 23: get ID, name = "cnt"
in Line: 23: get +
in Line: 23: get NUM, value = 1
in Line: 23: get ;
in Line: 24: get }
in Line: 25: get }
in Line: 26: get reserved word "return"
in Line: 26: get ID, name = "fib"
in Line: 26: get ;
in Line: 27: get }
in Line: 29: get reserved word "void"
in Line: 29: get ID, name = "main"
in Line: 29: get (
in Line: 29: get reserved word "void"
in Line: 29: get )
in Line: 30: get {
in Line: 31: get reserved word "int"
in Line: 31: get ID, name = "n"
in Line: 31: get ;
in Line: 32: get ID, name = "n"
in Line: 32: get =
in Line: 32: get ID, name = "input"
in Line: 32: get (
in Line: 32: get )
in Line: 32: get ;
in Line: 33: get ID, name = "output"
in Line: 33: get (
in Line: 33: get ID, name = "fibonacci"
in Line: 33: get (
in Line: 33: get ID, name = "n"
in Line: 33: get )
in Line: 33: get )
in Line: 33: get ;
in Line: 34: get }
in Line: 35: get EOF
```

4. Lex 文件

4.1 Lex 输入文件代码

```
%{
#include "globals.h"
#include "util.h"
#include "scan.h"
/* lexeme of identifier or reserved word */
char tokenString[MAXTOKENLEN+1];
void printToken( TokenType token, const char* tokenString );
%}

digit       [0-9]
number      {digit}+
letter      [a-zA-Z]
identifier  {letter}+
newline     \n
whitespace  [ \t]+

%%

"if"            {return IF;}
"while"         {return WHILE;}
"else"          {return ELSE;}
"return"        {return RETURN;}
"int"           {return INT;}
"void"          {return VOID;}
"="             {return ASSIGN;}
"=="            {return EQ;}
"<"             {return LT;}
"<="            {return LTE;}
">"             {return GT;}
">="            {return GTE;}
"!="            {return NE;}
"+"             {return PLUS;}
"-"             {return MINUS;}
"*"             {return TIMES;}
"/"             {return DIVIDE;}
"("             {return LPAREN;}
")"             {return RPAREN;}
";"             {return SEMI;}
"{"             {return LBRACE;}
"}"             {return LBRACE;}
{number}        {return NUM;}
{identifier}    {return ID;}
{newline}       {lineno++;}
{whitespace}    {/* skip whitespace */}
"/*"             {
```

```
                     char c;
                     int flag = 1;
                    do
                    { c = input();
                      if (c == EOF) break;
                      if (c == '\n') lineno++;
                      if (c == '*')
                      {
                        c = input();
                        if (c == '/') flag = 0;
                      }
                    } while (flag);
                  }
.                 {return ERROR;}

%%
TokenType getToken(void)
{
  static int firstTime = TRUE;
  TokenType currentToken;
  if (firstTime)
  { firstTime = FALSE;
    lineno++;
    yyin = source;
    yyout = listing;
  }
  currentToken = yylex();
  strncpy(tokenString,yytext,MAXTOKENLEN);
  if (TraceScan) {
    fprintf(listing,"\t%d: ",lineno);
    printToken(currentToken,tokenString);
  }
  return currentToken;
}

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

4.2 实验演示

- 安装 flex 并将其添加到 PATH 中，打开 cmm.l 所在目录，使用 `flex cmm.l` 命令生成 `lex.yy.c`
- 将 `main.c lex.yy.c util.c globals.h util.h scan.h` 放入一个项目中，编译生成 `lexScanner.exe`

- 使用 `lexScanner .\fibonacci.cmm` 命令进行词法分析，结果如下：

```
in Line:  1: get reserved word "int"
in Line:  1: get ID, name = "fibonacci"
in Line:  1: get (
in Line:  1: get reserved word "int"
in Line:  1: get ID, name = "n"
in Line:  1: get )
in Line:  2: get {
in Line:  3: get reserved word "int"
in Line:  3: get ID, name = "cnt"
in Line:  3: get ;
in Line:  4: get reserved word "int"
in Line:  4: get ID, name = "firstFib"
in Line:  4: get ;
in Line:  5: get reserved word "int"
in Line:  5: get ID, name = "secondFib"
in Line:  5: get ;
in Line:  6: get reserved word "int"
in Line:  6: get ID, name = "fib"
in Line:  6: get ;
in Line:  8: get ID, name = "firstFib"
in Line:  8: get =
in Line:  8: get NUM, value = 1
in Line:  8: get ;
in Line:  9: get ID, name = "secondFib"
in Line:  9: get =
in Line:  9: get NUM, value = 1
in Line:  9: get ;
in Line: 10: get ID, name = "cnt"
in Line: 10: get =
in Line: 10: get NUM, value = 2
in Line: 10: get ;
in Line: 12: get reserved word "if"
in Line: 12: get (
in Line: 12: get ID, name = "n"
in Line: 12: get ==
in Line: 12: get NUM, value = 1
in Line: 12: get )
in Line: 13: get reserved word "return"
in Line: 13: get NUM, value = 1
in Line: 13: get ;
in Line: 14: get reserved word "else"
in Line: 14: get reserved word "if"
in Line: 14: get (
in Line: 14: get ID, name = "n"
in Line: 14: get ==
in Line: 14: get NUM, value = 2
in Line: 14: get )
in Line: 15: get reserved word "return"
in Line: 15: get NUM, value = 1
in Line: 15: get ;
in Line: 16: get reserved word "else"
```

```
in Line: 17: get {
in Line: 18: get reserved word "while"
in Line: 18: get (
in Line: 18: get ID, name = "cnt"
in Line: 18: get <
in Line: 18: get ID, name = "n"
in Line: 18: get )
in Line: 19: get {
in Line: 20: get ID, name = "fib"
in Line: 20: get =
in Line: 20: get ID, name = "firstFib"
in Line: 20: get +
in Line: 20: get ID, name = "secondFib"
in Line: 20: get ;
in Line: 21: get ID, name = "firstFib"
in Line: 21: get =
in Line: 21: get ID, name = "secondFib"
in Line: 21: get ;
in Line: 22: get ID, name = "secondFib"
in Line: 22: get =
in Line: 22: get ID, name = "fib"
in Line: 22: get ;
in Line: 23: get ID, name = "cnt"
in Line: 23: get =
in Line: 23: get ID, name = "cnt"
in Line: 23: get +
in Line: 23: get NUM, value = 1
in Line: 23: get ;
in Line: 24: get }
in Line: 25: get }
in Line: 26: get reserved word "return"
in Line: 26: get ID, name = "fib"
in Line: 26: get ;
in Line: 27: get }
in Line: 29: get reserved word "void"
in Line: 29: get ID, name = "main"
in Line: 29: get (
in Line: 29: get reserved word "void"
in Line: 29: get )
in Line: 30: get {
in Line: 31: get reserved word "int"
in Line: 31: get ID, name = "n"
in Line: 31: get ;
in Line: 32: get ID, name = "n"
in Line: 32: get =
in Line: 32: get ID, name = "input"
in Line: 32: get (
in Line: 32: get )
in Line: 32: get ;
in Line: 33: get ID, name = "output"
in Line: 33: get (
in Line: 33: get ID, name = "fibonacci"
in Line: 33: get (
```

```
in Line: 33: get ID, name = "n"
in Line: 33: get )
in Line: 33: get )
in Line: 33: get ;
in Line: 34: get }
in Line: 35: get EOF
```

5. BNF 语法描述

```
1 . program → declaration-list

2 . declaration-list → declaration-list declaration | declaration

3 . declaration → var-declaration | fun-declaration

4 . var-declaration → type-specifier ID;

5 . type-specifier → int | void

6 . fun-declaration → type-specifierID(params) | compound-stmt

7 . params → params-list | void

8 . param-list → param-list , param | param

9 . param → type-specifierID

10. compound-stmt → {local-declarations statement-list}

11. local-declarations → local-declarations var-declaration | empty

12. statement-list → statement-list statement | empty

13. statement → expression-stmt
              | compound-stmt
              | selection-stmt
              | iteration-stmt
              | return-stmt

14. expression-stmt → expression; | ;

15. selection-stmt → if(expression)statement
                   | if (expression)statement else statement

16. iteration-stmt → while(expression)statement

17. return-stmt → return; | return expression;

18. expression → var = expression | simple-expression

19. var → ID
```

```
20. simple-expression → additive-expression relop additive-expression
                    | additive-expression

21. relop → <= | < | > | >= | == | !=

22. additive-expression → additive-expression addop term | term

23. addop → + | -

24. term → term mulop factor | factor

25. mulop → * | /

26. factor → (expression) | var | call | NUM

27. call → ID(args)

28. args → arg-list | empty

29. arg-list → arg-list, expression | expression
```

6. 语法分析程序

6.1 主要设计和实现思路

- 定义结点类型：结点分为三类： statement 结点, expression 结点, declaration 结点，其中 statement 结点包括 if 结点, while 结点, return 结点, 调用结点, 复合语句结点；expression 结点包括 oprator（操作符） 结点, identifier（标识符） 结点, const(num)（数字） 结点, assign（赋值） 结点；declaration 结点包括 scalar declaration（变量声明） 结点, function declaration（函数声明） 结点。

```c
typedef enum { StmtK, ExpK, DecK } NodeKind;
/* statement kind, expression kind, declaration kind */
typedef enum { IfK, WhileK, ReturnK, CallK, CompoundK } StmtKind;
/* if kind, while kind, return kind, call kind, compound kind */
typedef enum { OpK, IdK, ConstK, AssignK } ExpKind;
/* oprator kind, identifier kind, const(num) kind, assign kind */
typedef enum { ScalarDecK, FuncDecK } DecKind;
/* scalar declaration kind, function declaration kind */
typedef enum { Void, Integer, Function } ExpType;
```

- 定义数据结构 TreeNode：首先定义该结点的孩子结点（struct treeNode * child[MAXCHILDREN]）与兄弟结点（struct treeNode * sibling）；使用 int lineno 变量存储当前的代码行数；使用 union 将结点的三种可能的类型联合，方便后期使用；一个名为 op 的 TokenType 类型的变量用来存储操作符；一个名为 val 的整数用来存储当前值；一个名为 name 的字符串指针用了存储变量名字；三个名为 functionReturnType、variableDataType、expressionType 的变量存储可能的类型； isParameter 存储其是否为参数。

```
typedef struct treeNode
{
    struct treeNode * child[MAXCHILDREN];
    struct treeNode * sibling;
    int lineno;
    NodeKind nodekind;
    union
    {
        StmtKind stmt;
        ExpKind  exp;
        DecKind  dec;
    } kind;

    TokenType op;
    int val;
    char *name;

    ExpType functionReturnType;
    ExpType variableDataType;
    ExpType expressionType;

    int isParameter;

    struct treeNode *declaration;

} TreeNode;
```

- 采用递归下降进行语法分析，参照 C--语言的 BNF 语法，每个非终结符定义为一个函数，返回类型为 TreeNode

- 下降过程中，如果根据 BNF 范式，如果后一语句不为空，则将前一语句作为后一语句的树根，将他们全部连接起来，然后依次向下进行递归调用，根据相应的函数进行分析。

分析结束后，调用 `util.c` 中的 `printTree()`函数进行语法树的输出，函数如下：

```c
#define INDENT   indentno += 4
#define UNINDENT indentno -= 4


static void printSpaces(void)
{
    for (int i=0; i<indentno; ++i)
        fprintf(listing, " ");
}

char *typeName(ExpType type)
{
    static char i[] = "integer";
    static char v[] = "void";
    static char invalid[] = "<<invalid type>>";

    switch (type)
    {
        case Integer: return i; break;
        case Void:    return v; break;
        default:      return invalid;
    }
}

void printTree(TreeNode *tree)
{
    int i;

    INDENT;

    while (tree != NULL)
    {
        printSpaces();

        if (tree->nodekind == DecK)
        {
            switch(tree->kind.dec)
            {
            case ScalarDecK:
                fprintf(listing,"[Scalar declaration \"%s\" of type \"%s\"]\n"
                      , tree->name, typeName(tree->variableDataType));
                break;
            case ArrayDecK:
                fprintf(listing, "[Array declaration \"%s\" of size %d"
                        " and type \"%s\"]\n",
                      tree->name, tree->val, typeName(tree->variableDataType));
                break;
            case FuncDecK:
                fprintf(listing, "[Function declaration \"%s()\""
                        " of return type \"%s\"]\n",
                        tree->name, typeName(tree->functionReturnType));
                break;
            default:
```

```
        fprintf(listing, "<<<unknown declaration type>>>\n");
      break;
    }
}
else if (tree->nodekind == ExpK)
{
    switch(tree->kind.exp)
    {
    case OpK:
        fprintf(listing, "[Operator \"");
        printToken(tree->op, "");
        fprintf(listing, "\"]\n");
        break;
    case IdK:
        fprintf(listing, "[Identifier \"%s", tree->name);
        if (tree->val != 0) /* array indexing */
            fprintf(listing, "[%d]", tree->val);
        fprintf(listing, "\"]\n");
        break;
    case ConstK:
        fprintf(listing, "[Literal constant \"%d\"]\n", tree->val);
        break;
    case AssignK:
        fprintf(listing, "[Assignment]\n");
        break;
    default:
        fprintf(listing, "<<<unknown expression type>>>\n");
        break;
    }
}
else if (tree->nodekind == StmtK)
{
    switch(tree->kind.stmt)
    {
    case CompoundK:
        fprintf(listing, "[Compound statement]\n");
        break;
            case IfK:
                fprintf(listing, "[IF statement]\n");
                break;
            case WhileK:
                fprintf(listing, "[WHILE statement]\n");
                break;
            case ReturnK:
                fprintf(listing, "[RETURN statement]\n");
                break;
            case CallK:
                fprintf(listing, "[Call to function \"%s()\"]\n",
                         tree->name);
                break;
    default:
        fprintf(listing, "<<<unknown statement type>>>\n");
        break;
```

```
            }
        }
        else
            fprintf(listing, "<<<unknown node kind>>>\n");

        for (i=0; i<MAXCHILDREN; ++i)
            printTree(tree->child[i]);

        tree = tree->sibling;
    }

    UNINDENT;
}
```

6.2 递归下降语法分析程序代码

```
#include "globals.h"
#include "util.h"
#include "scan.h"
#include "parse.h"

static TokenType token; // current token

/* function prototypes for recursive calls */
static TreeNode *declaration_list(void);
static TreeNode *declaration(void);
static TreeNode *var_declaration(void);
static TreeNode *param(void);
static TreeNode *param_list(void);
static TreeNode *compound_statement(void);
static TreeNode *local_declarations(void);
static TreeNode *statement_list(void);
static TreeNode *statement(void);
static TreeNode *expression_statement(void);
static TreeNode *if_statement(void);
static TreeNode *while_statement(void);
static TreeNode *return_statement(void);
static TreeNode *expression(void);
static TreeNode *simple_expression(TreeNode *passdown);
static TreeNode *additive_expression(TreeNode *passdown);
static TreeNode *term(TreeNode *passdown);
static TreeNode *factor(TreeNode *passdown);
static TreeNode *args(void);
static TreeNode *arg_list(void);
static TreeNode *identifier_statement(void);

static void syntaxError(char *message)
{
    fprintf(listing, ">>> Syntax error at line %d: %s", lineno, message);
}
```

```
static void match(TokenType expected)
{
    if (token == expected)
        token = getToken();
    else
    {
        syntaxError("unexpected token ");
        printToken(token, tokenString);
        fprintf(listing, "\n");
    }
}

static ExpType matchType()
{
    ExpType t_type = Void;

    switch (token)
    {
        case INT:
            t_type = Integer;
            token = getToken();
            break;
        case VOID:
            t_type = Void;
            token = getToken();
            break;
        default:
        {
            syntaxError("expected a type identifier but got a ");
            printToken(token, tokenString);
            fprintf(listing, "\n");
            break;
        }
    }

    return t_type;
}

static int isAType(TokenType tok)
{
    if ((tok == INT) || (tok == VOID))
        return TRUE;
    else
        return FALSE;
}

static TreeNode * declaration_list(void)
{
    TreeNode * tree;
    TreeNode * ptr;

    tree = declaration();
    ptr = tree;
```

```
    while (token != ENDOFFILE)
    {
        TreeNode *tmp;

        tmp = declaration();
        if ((tmp != NULL) && (ptr != NULL))
        {
            ptr->sibling = tmp;
            ptr = tmp;
        }
    }

    return tree;
}

static TreeNode *declaration(void)
{
    TreeNode *tree = NULL;
    ExpType declaration_type;
    char *identifier;

    declaration_type = matchType();
    identifier = copyString(tokenString);
    match(ID);

    switch (token)
    {
        case SEMI: /* variable declaration */
            tree = newDecNode(ScalarDecK);
            if (tree != NULL)
            {
                tree->variableDataType = declaration_type;
                tree->name = identifier;
            }
            match(SEMI);
            break;

        case LPAREN: /* function declaration */
            tree = newDecNode(FuncDecK);
            if (tree != NULL)
            {
                tree->functionReturnType = declaration_type;
                tree->name = identifier;
            }
            match(LPAREN);
            if (tree != NULL)
                tree->child[0] = param_list();
            match(RPAREN);
            if (tree != NULL)
                tree->child[1] = compound_statement();
            break;
```

```
        default:
            syntaxError("unexpected token ");
            printToken(token, tokenString);
            fprintf(listing, "\n");
            token = getToken();
            break;
    }

    return tree;
}

static TreeNode *var_declaration(void)
{
    TreeNode *tree = NULL;
    ExpType declaration_type;
    char *identifier;

    declaration_type = matchType();
    identifier = copyString(tokenString);
    match(ID);

    if(token == SEMI)
    {
        tree = newDecNode(ScalarDecK); /* variable declaration */
        if (tree != NULL)
        {
            tree->variableDataType = declaration_type;
            tree->name = identifier;
        }
        match(SEMI);
    }
    else
    {
        syntaxError("unexpected token ");
        printToken(token, tokenString);
        fprintf(listing, "\n");
        token = getToken();
    }
    return tree;
}

static TreeNode *param(void)
{
    TreeNode *tree;
    ExpType parmType;
    char *identifier;

    parmType = matchType(); /* get type of formal parameter */
    identifier = copyString(tokenString);
    match(ID);

    tree = newDecNode(ScalarDecK);
```

```
    if (tree != NULL)
    {
        tree->name = identifier;
        tree->val = 0;
        tree->variableDataType = parmType;
        tree->isParameter = TRUE;
    }

    return tree;
}

static TreeNode *param_list(void)
{
    TreeNode *tree;
    TreeNode *ptr;
    TreeNode *newNode;

    if (token == VOID) /* void param */
    {
        match(VOID);
        return NULL;
    }

    tree = param();
    ptr = tree;

    while ((tree != NULL) && (token == COMMA)) /* mutiple params */
    {
        match(COMMA);
        newNode = param();
        if (newNode != NULL)
        {
            ptr->sibling = newNode;
            ptr = newNode;
        }
    }

    return tree;
}

static TreeNode *compound_statement(void)
{
    TreeNode *tree = NULL;

    match(LBRACE);

    if ((token != RBRACE) && (tree = newStmtNode(CompoundK)))
    {
        if (isAType(token))
            tree->child[0] = local_declarations();
        if (token != RBRACE)
            tree->child[1] = statement_list();
    }
```

```
    match(RBRACE);

    return tree;
}

static TreeNode *local_declarations(void)
{
    TreeNode *tree;
    TreeNode *ptr;
    TreeNode *newNode;

    /* find first variable declaration, if it exists */
    if (isAType(token))
        tree = var_declaration();

    /* subsetmpuent variable declarations */
    if (tree != NULL)
    {
        ptr = tree;

        while (isAType(token))
        {
            newNode = var_declaration();
            if (newNode != NULL)
            {
                ptr->sibling = newNode;
                ptr = newNode;
            }
        }
    }

    return tree;
}

static TreeNode *statement_list(void)
{
    TreeNode *tree = NULL;
    TreeNode *ptr;
    TreeNode *newNode;

    if (token != RBRACE)
    {
        tree = statement();
        ptr = tree;

        while (token != RBRACE)
        {
            newNode = statement();
            if ((ptr != NULL) && (newNode != NULL))
            {
                ptr->sibling = newNode;
                ptr = newNode;
            }
```

```
        }
    }

    return tree;
}

static TreeNode *statement(void)
{
    TreeNode *tree = NULL;

    switch (token)
    {
        case IF:
            tree = if_statement();
            break;
        case WHILE:
            tree = while_statement();
            break;
        case RETURN:
            tree = return_statement();
            break;
        case LBRACE:
            tree = compound_statement();
            break;
        case ID:
        case SEMI:
        case LPAREN:
        case NUM:
            tree = expression_statement();
            break;
        default:
            syntaxError("unexpected token ");
            printToken(token, tokenString);
            fprintf(listing, "\n");
            token = getToken();
            break;
    }

    return tree;
}

static TreeNode *expression_statement(void)
{
    TreeNode *tree = NULL;

    if (token == SEMI)
        match(SEMI);
    else if (token != RBRACE)
    {
        tree = expression();
        match(SEMI);
    }
```

```
    return tree;
}

static TreeNode *if_statement(void)
{
    TreeNode *tree;
    TreeNode *expr;
    TreeNode *ifStmt;
    TreeNode *elseStmt = NULL;


    match(IF);
    match(LPAREN);
    expr = expression();
    match(RPAREN);
    ifStmt = statement();

    if (token == ELSE)
    {
        match(ELSE);
        elseStmt = statement();
    }

    tree = newStmtNode(IfK);
    if (tree != NULL)
    {
        tree->child[0] = expr;
        tree->child[1] = ifStmt;
        tree->child[2] = elseStmt;
    }

    return tree;
}

static TreeNode *while_statement(void)
{
    TreeNode *tree;
    TreeNode *expr;
    TreeNode *stmt;

    match(WHILE);
    match(LPAREN);
    expr = expression();
    match(RPAREN);
    stmt = statement();

    tree = newStmtNode(WhileK);
    if (tree != NULL)
    {
        tree->child[0] = expr;
        tree->child[1] = stmt;
    }
```

```
    return tree;
}

static TreeNode *return_statement(void)
{
    TreeNode *tree;
    TreeNode *expr = NULL;

    match(RETURN);

    tree = newStmtNode(ReturnK);
    if (token != SEMI)
        expr = expression();

    if (tree != NULL)
        tree->child[0] = expr;

    match(SEMI);

    return tree;
}

static TreeNode *expression(void)
{
    TreeNode *tree = NULL;
    TreeNode *lvalue = NULL;
    TreeNode *rvalue = NULL;
    int gotLvalue = FALSE;


    if (token == ID)
    {
        lvalue = identifier_statement();
        gotLvalue = TRUE;
    }

    /* assign */
    if ((gotLvalue == TRUE) && (token == ASSIGN))
    {
        if ((lvalue != NULL) && (lvalue->nodekind == ExpK) &&
            (lvalue->kind.exp == IdK))
        {
            match(ASSIGN);
            rvalue = expression();
            tree = newExpNode(AssignK);
            if (tree != NULL)
            {
                tree->child[0] = lvalue; /* left  value */
                tree->child[1] = rvalue; /* right value*/
            }
        }
        else
        {
```

```
            syntaxError("attempt to assign to something not an lvalue\n");
            token = getToken();
        }
    }
    else
        tree = simple_expression(lvalue);

    return tree;
}

static TreeNode *simple_expression(TreeNode *passdown)
{
    TreeNode *tree;
    TreeNode *lExpr = NULL;
    TreeNode *rExpr = NULL;
    TokenType operator;

    lExpr = additive_expression(passdown);

    if ((token == LTE) || (token == GTE) || (token == GT) ||
        (token == LT) || (token == EQ) || (token == NE))
    {
        operator = token;
        match(token);
        rExpr = additive_expression(NULL);

        tree = newExpNode(OpK);
        if (tree != NULL)
        {
            tree->child[0] = lExpr;
            tree->child[1] = rExpr;
            tree->op = operator;
        }
    }
    else
        tree = lExpr;

    return tree;
}

static TreeNode *additive_expression(TreeNode *passdown)
{
    TreeNode *tree;
    TreeNode *newNode;

    tree = term(passdown);

    while ((token == PLUS) || (token == MINUS))
    {
        newNode = newExpNode(OpK);
        if (newNode != NULL)
        {
            newNode->child[0] = tree;
```

```
            newNode->op = token;
            tree = newNode;
            match(token);
            tree->child[1] = term(NULL);
        }
    }

    return tree;
}

static TreeNode *term(TreeNode *passdown)
{
    TreeNode *tree;
    TreeNode *newNode;

    tree = factor(passdown);

    while ((token == TIMES) || (token == DIVIDE))
    {
        newNode = newExpNode(OpK);

        if (newNode != NULL)
        {
            newNode->child[0] = tree;
            newNode->op = token;
            tree = newNode;
            match(token);
            newNode->child[1] = factor(NULL);
        }
    }

    return tree;
}

static TreeNode *factor(TreeNode *passdown)
{
    TreeNode *tree = NULL;

    /* If the subtree in "passdown" is a Factor, pass it back. */
    if (passdown != NULL) return passdown;

    if (token == ID)
    {
        tree = identifier_statement();
    }
    else if (token == LPAREN)
    {
        match(LPAREN);
        tree = expression();
        match(RPAREN);
    }
    else if (token == NUM)
    {
```

```
        tree = newExpNode(ConstK);
        if (tree != NULL)
        {
            tree->val = atoi(tokenString);
            tree->variableDataType = Integer;
        }
        match(NUM);
    }
    else
    {
        syntaxError("unexpected token ");
        printToken(token, tokenString);
        fprintf(listing, "\n");
        token = getToken();
    }

    return tree;
}

static TreeNode *identifier_statement(void)
{
    TreeNode *tree;
    TreeNode *expr = NULL;
    TreeNode *arguments = NULL;
    char *identifier;

    if (token == ID)
        identifier = copyString(tokenString);
    match(ID);

    if (token == LPAREN)
    {
        match(LPAREN);
        arguments = args();
        match(RPAREN);

        tree = newStmtNode(CallK);
        if (tree != NULL)
        {
            tree->child[0] = arguments;
            tree->name = identifier;
        }
    }
    else
    {
        tree = newExpNode(IdK);
        if (tree != NULL)
        {
            tree->child[0] = expr;
            tree->name = identifier;
        }
    }
```

```
    return tree;
}

static TreeNode *args(void)
{
    TreeNode *tree = NULL;

    if (token != RPAREN)
        tree = arg_list();

    return tree;
}

static TreeNode *arg_list(void)
{
    TreeNode *tree;
    TreeNode *ptr;
    TreeNode *newNode;

    tree = expression();
    ptr = tree;

    while (token == COMMA)
    {
        match(COMMA);
        newNode = expression();

        if ((ptr != NULL) && (tree != NULL))
        {
            ptr->sibling = newNode;
            ptr = newNode;
        }
    }

    return tree;
}

TreeNode *Parse(void)
{
    TreeNode *t;

    token = getToken();
    t = declaration_list();
    if (token != ENDOFFILE)
        syntaxError("Unexpected symbol at end of file\n");

    /* t points to the fully-constructed syntax tree */
    return t;
}
```

6.3 实验演示

1. 求三个整数中的最大值

```
[Function declaration "max()" of return type "integer"]
    [Scalar declaration "x" of type "integer"]
    [Scalar declaration "y" of type "integer"]
    [Scalar declaration "z" of type "integer"]
    [Compound statement]
        [Scalar declaration "biggest" of type "integer"]
        [Assignment]
            [Identifier "biggest"]
            [Identifier "x"]
        [IF statement]
            [Operator ">"]
                [Identifier "y"]
                [Identifier "biggest"]
            [Assignment]
                [Identifier "biggest"]
                [Identifier "y"]
        [IF statement]
            [Operator ">"]
                [Identifier "z]"]
                [Identifier "biggest"]
            [Assignment]
                [Identifier "biggest"]
                [Identifier "z"]
        [RETURN statement]
            [Identifier "biggest"]
[Function declaration "main()" of return type "void"]
    [Compound statement]
        [Scalar declaration "x" of type "integer"]
        [Scalar declaration "y" of type "integer"]
        [Scalar declaration "z" of type "integer"]
        [Scalar declaration "biggest" of type "integer"]
        [Assignment]
            [Identifier "x"]
            [Call to function "input()"]
        [Assignment]
            [Identifier "y"]
            [Call to function "input()"]
        [Assignment]
            [Identifier "z"]
            [Call to function "input()"]
        [Assignment]
            [Identifier "biggest"]
            [Call to function "max()"]
                [Identifier "x"]
                [Identifier "y"]
                [Identifier "z"]
```

```
        [Call to function "output()"]
            [Identifier "biggest"]
```

2. 给定 N，求 1 到 N 之和

```
[Function declaration "sum()" of return type "integer"]
    [Scalar declaration "n" of type "integer"]
    [Compound statement]
        [Scalar declaration "result" of type "integer"]
        [Scalar declaration "i" of type "integer"]
        [Assignment]
            [Identifier "i"]
            [Literal constant "1"]
        [Assignment]
            [Identifier "result"]
            [Literal constant "0"]
        [WHILE statement]
            [Operator "<="]
                [Identifier "i"]
                [Identifier "n"]
            [Compound statement]
                [Assignment]
                    [Identifier "result"]
                    [Operator "+"]
                        [Identifier "result"]
                        [Identifier "i"]
                [Assignment]
                    [Identifier "i"]
                    [Operator "+"]
                        [Identifier "i"]
                        [Literal constant "1"]
        [RETURN statement]
            [Identifier "result"]
[Function declaration "main()" of return type "void"]
    [Compound statement]
        [Scalar declaration "n" of type "integer"]
        [Scalar declaration "s" of type "integer"]
        [Assignment]
            [Identifier "n"]
            [Call to function "intput()"]
        [Assignment]
            [Identifier "s"]
            [Call to function "sum()"]
                [Identifier "n"]
        [Call to function "output()"]
            [Identifier "s"]
```

3. 计算第 n 个斐波那契数

```
[Function declaration "fibonacci()" of return type "integer"]
    [Scalar declaration "n" of type "integer"]
    [Compound statement]
        [Scalar declaration "cnt" of type "integer"]
```

```
        [Scalar declaration "firstFib" of type "integer"]
        [Scalar declaration "secondFib" of type "integer"]
        [Scalar declaration "fib" of type "integer"]
        [Assignment]
            [Identifier "firstFib"]
            [Literal constant "1"]
        [Assignment]
            [Identifier "secondFib"]
            [Literal constant "1"]
        [Assignment]
            [Identifier "cnt"]
            [Literal constant "2"]
        [IF statement]
            [Operator "=="]
                [Identifier "n"]
                [Literal constant "1"]
            [RETURN statement]
                [Literal constant "1"]
            [IF statement]
                [Operator "=="]
                    [Identifier "n"]
                    [Literal constant "2"]
                [RETURN statement]
                    [Literal constant "1"]
                [Compound statement]
                    [WHILE statement]
                        [Operator "<"]
                            [Identifier "cnt"]
                            [Identifier "n"]
                        [Compound statement]
                            [Assignment]
                                [Identifier "fib"]
                                [Operator "+"]
                                    [Identifier "firstFib"]
                                    [Identifier "secondFib"]
                            [Assignment]
                                [Identifier "firstFib"]
                                [Identifier "secondFib"]
                            [Assignment]
                                [Identifier "secondFib"]
                                [Identifier "fib"]
                            [Assignment]
                                [Identifier "cnt"]
                                [Operator "+"]
                                    [Identifier "cnt"]
                                    [Literal constant "1"]
        [RETURN statement]
            [Identifier "fib"]
[Function declaration "main()" of return type "void"]
    [Compound statement]
        [Scalar declaration "n" of type "integer"]
        [Assignment]
            [Identifier "n"]
```

```
            [Call to function "input()"]
        [Call to function "output()"]
            [Call to function "fibonacci()"]
                [Identifier "n"]
```

7. 语义分析程序

7.1 主要设计和实现思路

7.1.1 符号表构造

首先通过 void buildSymbolTable(TreeNode *syntaxTree)函数进行符号表构造

- static void drawRuler(FILE *output, char *string)函数，用于确定符号表的格式,打印分割线
- static void declarePredefines(void)函数，用来将 C--语言内置的 input()和 output()函数添加进符号表
- static void startBuildSymbolTable(TreeNode *syntaxTree)函数开始构造符号表：
  - ➢ 该函数的参数为语法树根结点，通过遍历整个语法树，来寻找结点类型为 DECK（声明结点）的结点，并将他们插入到符号表中，插入的过程中将进行检测。
  - ➢ 声明也分为两种：变量的声明与函数的声明：变量的声明可直接插入到符号表；而函数的声明中可能有变量声明，此时，将会调用之前提到过的 drawruler()函数，增加一张符号表，用于处理该函数内部的变量声明。
  - ➢ 对于非声明类结点，只需进行错误检测，即在既有的符号表中查询其是否进行过声明。

7.1.2 类型检查

首先通过 traverse(syntaxTree, nullProc, checkNode)函数，检查相邻语法树结点的词法属性来判断是否出错。

- static void nullProc(TreeNode *syntaxTree)函数直接返回（即遇到叶子结点），不进行其他操作
- static void checkNode(TreeNode *syntaxTree)函数通过遍历语法树，遍历到当前的一个结点之后，检查该结点的相邻结点是否符合语法规则，如果不符合就报错，如果符合就可以继续遍历。

7.2 类型检查语义分析程序代码

7.2.1 symtab.c

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

#include "Globals.h"
#include "SymTab.h"
#include "Util.h"

#define MAXTABLESIZE 233
#define HIGHWATERMARK "__invalid__"

/* The hash table itself */
static HashNodePtr hashtable[MAXTABLESIZE];

/* The "temporary list", used to track scopes. */
static HashNodePtr tempList;

extern int TraceAnalyse;
int scopeDepth;

static HashNodePtr allocateSymbolNode(char *name,
                                      TreeNode *declaration,
                                      int lineDefined);

/* hashfunction(): takes a string and generates a hash value. */
```

```c
static int hashFunction(char *key);

/* error reporting */
static void flagError(char *message);

/* used in symbol table scope dump */
static char *formatSymbolType(TreeNode *node);

/* the guts of dumpCurrentScope() */
static void startDumpCurrentScope(HashNodePtr cursor);

void initSymbolTable(void)
{
    memset(hashtable, 0, sizeof(HashNodePtr) * MAXTABLESIZE);
    tempList = NULL;
}

void insertSymbol(char *name, TreeNode *symbolDefNode, int lineDefined)
{
    char errorString[80];

    HashNodePtr newHashNode, temp;
    int hashBucket;

    /* If the symbol already exists, flag an error */
    if (symbolAlreadyDeclared(name))
    {
        sprintf(errorString, "duplicate identifier \"%s\"\n", name);
        flagError(errorString);
    }
    else
    {
        /* Locate bucket we're using */
        hashBucket = hashFunction(name);
        /* Allocate and insert record on front of bucket */
        newHashNode = allocateSymbolNode(name, symbolDefNode, lineDefined);
        if (newHashNode != NULL)
        {
            temp = hashtable[hashBucket];
            hashtable[hashBucket] = newHashNode;
            newHashNode->next = temp;
        }

        /* Stick node on front of "tempList" */
        newHashNode = allocateSymbolNode(name, symbolDefNode, lineDefined);
        if (newHashNode != NULL)
        {
            temp = tempList;
            tempList = newHashNode;
            tempList->next = temp;
        }
    }
}
```

```c
/* Check to see if the symbol given by "name" is already declared in thecurrent scope.
*/

int symbolAlreadyDeclared(char *name)
{
    int symbolFound = FALSE;
    HashNodePtr cursor;

    /* Scan "tempList" within _current_ scope for duplicate definition */
    cursor = tempList;

    while ((cursor != NULL) && (!symbolFound) && ((strcmp(cursor->name,
HIGHWATERMARK) != 0)))
    {
        if (strcmp(name, cursor->name) == 0)
            symbolFound = TRUE;
        else
            cursor = cursor->next;
    }

    return (symbolFound);
}

HashNodePtr lookupSymbol(char *name)
{
    HashNodePtr cursor;
    int hashBucket;    /* hash bucket on which to conduct our search */
    int found = FALSE;

    hashBucket = hashFunction(name);
    cursor = hashtable[hashBucket];

    while (cursor != NULL)
    {
        if (strcmp(name, cursor->name) == 0)
        {
            found = TRUE;
            break;
        }

        cursor = cursor->next;
    }

    if (found == TRUE)
        return cursor;
    else
        return NULL;
}

void dumpCurrentScope()
{
    HashNodePtr cursor;
```

```
    cursor = tempList;

    /* if the current scope isn't empty,  dump it out */
    if ((cursor != NULL) && (strcmp(HIGHWATERMARK, cursor->name)))
        startDumpCurrentScope(cursor);
}

#define IDENT_LEN 12

static void startDumpCurrentScope(HashNodePtr cursor)
{
    char paddedIdentifier[IDENT_LEN + 1];
    char *typeInformation; /* used to catch result of formatSymbolType */

    if ((cursor->next != NULL) && (strcmp(cursor->next->name, HIGHWATERMARK) != 0))
        startDumpCurrentScope(cursor->next);

    /* pad identifier name */
    memset(paddedIdentifier, ' ', IDENT_LEN);
    memmove(paddedIdentifier, cursor->name, strlen(cursor->name));
    paddedIdentifier[IDENT_LEN] = '\0';

    /* output symbol table entry */
    typeInformation = formatSymbolType(cursor->declaration);

    fprintf(listing, "%3d   %s   %7d     %c    %s\n",
            scopeDepth,
            paddedIdentifier,
            cursor->lineFirstReferenced,
            cursor->declaration->isParameter ? 'Y' : 'N',
            typeInformation);

    free(typeInformation);
}

void newScope()
{
    HashNodePtr newNode, temp;
    newNode = allocateSymbolNode(HIGHWATERMARK, NULL, 0);
    if (newNode != NULL)
    {
        temp = tempList;
        tempList = newNode;
        tempList->next = temp;
    }
}

void endScope()
{
    HashNodePtr hashPtr;
    HashNodePtr temp; /* used in freeing HashNodes */
    int hashBucket;
```

```
    while ((tempList != NULL) && (strcmp(HIGHWATERMARK, tempList->name)) != 0)
    {
        /* locate this node in the hash table, delete it */
        hashBucket = hashFunction(tempList->name);
        hashPtr = hashtable[hashBucket];
        assert((tempList != NULL) && (hashtable[hashBucket] != NULL));
        assert(strcmp(tempList->name, hashPtr->name) == 0);

        /* delete from hash table */
        temp = hashtable[hashBucket]->next;
        free(hashtable[hashBucket]);
        hashtable[hashBucket] = temp;

        /* ... and from second list */
        temp = tempList->next;
        free(tempList);
        tempList = temp;
    }

    /* delete high water mark */
    assert(strcmp(tempList->name, HIGHWATERMARK) == 0);
    temp = tempList->next;
    free(tempList);
    tempList = temp;
}

static HashNodePtr allocateSymbolNode(char *name,
                                      TreeNode *declaration,
                                      int lineDefined)
{
    HashNode *temp;

    temp = (HashNode *)malloc(sizeof(HashNode));
    if (temp == NULL)
    {
        Error = TRUE;
        fprintf(listing,
                "*** Out of memory allocating memory for symbol table\n");
    }
    else
    {
        temp->name = copyString(name);
        temp->declaration = declaration;
        temp->lineFirstReferenced = lineDefined;
        temp->next = NULL;
    }

    return temp;
}

/* Power-of-two multiplier in hash function */
#define SHIFT 4
```

```
/* Code borrowed from Louden p.522 */
static int hashFunction(char *key)
{
    int temp = 0;
    int i = 0;

    while (key[i] != '\0')
    {
        temp = ((temp << SHIFT) + key[i]) % MAXTABLESIZE;
        ++i;
    }

    return temp;
}

static void flagError(char *message)
{
    fprintf(listing, ">>> Semantic error (symbol table): %s", message);
    Error = TRUE; /* global variable to inhibit subseq. passes on error */
}

static char *formatSymbolType(TreeNode *node)
{
    char stringBuffer[100];

    if ((node == NULL) || (node->nodekind != DecK))
        strcpy(stringBuffer, "<<ERROR>>");
    else
    {
        /* node is a declaration */
        switch (node->kind.dec)
        {
        case ScalarDecK:
            sprintf(stringBuffer, "Scalar of type %s",
                    typeName(node->variableDataType));
            break;
        case ArrayDecK:
            sprintf(stringBuffer, "Array of type %s with %d elements",
                    typeName(node->variableDataType), node->val);
            break;
        case FuncDecK:
            sprintf(stringBuffer, "Function with return type %s",
                    typeName(node->functionReturnType));
            break;
        default:
            strcpy(stringBuffer, "<<UNKNOWN>>");
            break;
        }
    }

    return copyString(stringBuffer);
}
```

7.2.2 analyse.c

```
#include "Analyse.h"
#include "Globals.h"
#include "SymTab.h"
#include "Util.h"

/* draw a ruler on the screen */
static void drawRuler(FILE *output, char *string);

/* the guts of buildSymbolTable() */
static void startBuildSymbolTable(TreeNode *syntaxTree);

/* flag an error from the type checker */
static void flagSemanticError(char *str);

/* generic tree traversal routine */
static void traverse(TreeNode *syntaxTree,
                     void (*preProc)(TreeNode *),
                     void (*postProc)(TreeNode *));

/* routine to perform the actual type check on a node */
static void checkNode(TreeNode *syntaxTree);

/* dummy do-nothing procedure used to keep traversal() happy */
static void nullProc(TreeNode *syntaxTree);

/* traverse the syntax tree, marking global variables as such */
void markGlobals(TreeNode *tree);

/* declare the C-minus "built-in" input() and output() routines */
static void declarePredefines(void);

/* type-check functions' formal parameters against actual parameters */
static int checkFormalAgainstActualParms(TreeNode *formal, TreeNode *actual);

void buildSymbolTable(TreeNode *syntaxTree)
{
    /* Format headings */
    if (TraceAnalyse)
    {
        drawRuler(listing, "");
        fprintf(listing,
                "Scope Identifier        Line   Is a   Symbol type\n");
        fprintf(listing,
                "depth                   Decl.  parm?\n");
    }

    declarePredefines(); /* make input() and output() visible in globals */
    startBuildSymbolTable(syntaxTree);
}
```

```c
void typeCheck(TreeNode *syntaxTree)
{
    traverse(syntaxTree, nullProc, checkNode);
}

/* make input() and output() visible in globals */
static void declarePredefines(void)
{
    TreeNode *input;
    TreeNode *output;
    TreeNode *temp;

    /* define "int input(void)" */
    input = newDecNode(FuncDecK);
    input->name = copyString("input");
    input->functionReturnType = Integer;
    input->expressionType = Function;

    /* define "void output(int)" */
    temp = newDecNode(ScalarDecK);
    temp->name = copyString("arg");
    temp->variableDataType = Integer;
    temp->expressionType = Integer;

    output = newDecNode(FuncDecK);
    output->name = copyString("output");
    output->functionReturnType = Void;
    output->expressionType = Function;
    output->child[0] = temp;

    /* get input() and output() added to global scope */
    insertSymbol("input", input, 0);
    insertSymbol("output", output, 0);
}

static void startBuildSymbolTable(TreeNode *syntaxTree)
{
    int i;                     /* iterate over node children */
    HashNodePtr currentSymbol; /* symbol being looked up */
    char errorMessage[80];

    /* used to decorate RETURN nodes with enclosing procedure */
    static TreeNode *enclosingFunction = NULL;

    while (syntaxTree != NULL)
    {
        /* Examine current symbol: if it's a declaration, insert intosymbol table. */
        if (syntaxTree->nodekind == DecK)
            insertSymbol(syntaxTree->name, syntaxTree, syntaxTree->lineno);

        /* If entering a new function, tell the symbol table */
        if ((syntaxTree->nodekind == DecK) && (syntaxTree->kind.dec == FuncDecK))
        {
```

```
        /* record the enclosing procedure declaration */
        enclosingFunction = syntaxTree;

        if (TraceAnalyse)
            drawRuler(listing, syntaxTree->name);

        newScope();
        ++scopeDepth;
    }

    /* if entering a compound-statement, create a new scope as well */
    if ((syntaxTree->nodekind == StmtK) && (syntaxTree->kind.stmt == CompoundK))
    {
        newScope();
        ++scopeDepth;
    }

    /* if it's an identifier, it needs to be check symbol table*/
    if (((syntaxTree->nodekind == ExpK) && (syntaxTree->kind.exp == IdK))
     || ((syntaxTree->nodekind == StmtK) && (syntaxTree->kind.stmt == CallK)))
    {
        currentSymbol = lookupSymbol(syntaxTree->name);
        if (currentSymbol == NULL)
        {
            /* operation failed; say so to user */
            sprintf(errorMessage,
                    "identifier \"%s\" unknown or out of scope\n",
                    syntaxTree->name);
            flagSemanticError(errorMessage);
        }
        else
            syntaxTree->declaration = currentSymbol->declaration;
    }

    /* mark return type */
    if ((syntaxTree->nodekind == StmtK) &&
        (syntaxTree->kind.stmt == ReturnK))
    {
        syntaxTree->declaration = enclosingFunction;
    }

    for (i = 0; i < MAXCHILDREN; ++i)
        startBuildSymbolTable(syntaxTree->child[i]);

    /* If leaving a scope, tell the symbol table */
    if (((syntaxTree->nodekind == DecK) && (syntaxTree->kind.dec == FuncDecK))
     || ((syntaxTree->nodekind == StmtK) && (syntaxTree->kind.stmt == CompoundK)))
    {
        if (TraceAnalyse)
            dumpCurrentScope();
        --scopeDepth;
        endScope();
    }
```

```
        syntaxTree = syntaxTree->sibling;
    }
}

static void drawRuler(FILE *output, char *string)
{
    int length;
    int numTrailingDashes;
    int i;

    /* empty string */
    if (strcmp(string, "") == 0)
        length = 0;
    else
        length = strlen(string) + 2;

    fprintf(output, "---");
    if (length > 0)
        fprintf(output, " %s ", string);
    numTrailingDashes = 45 - length;

    for (i = 0; i < numTrailingDashes; ++i)
        fprintf(output, "-");
    fprintf(output, "\n");
}

static void flagSemanticError(char *str)
{
    fprintf(listing, ">>> Semantic error (type checker): %s", str);
    Error = TRUE;
}

/* generic tree traversal routine */
static void traverse(TreeNode *syntaxTree,
                     void (*preProc)(TreeNode *),
                     void (*postProc)(TreeNode *))
{
    while (syntaxTree != NULL)
    {
        preProc(syntaxTree);
        for (int i = 0; i < MAXCHILDREN; ++i)
            traverse(syntaxTree->child[i], preProc, postProc);
        postProc(syntaxTree);
        syntaxTree = syntaxTree->sibling;
    }
}

static int checkFormalAgainstActualParms(TreeNode *formal, TreeNode *actual)
{
    TreeNode *firstList;
    TreeNode *secondList;

    firstList = formal->child[0];
```

```
    secondList = actual->child[0];

    while ((firstList != NULL) && (secondList != NULL))
    {
        if (firstList->expressionType != secondList->expressionType)
            return FALSE;

        if (firstList)
            firstList = firstList->sibling;
        if (secondList)
            secondList = secondList->sibling;
    }

    if (((firstList == NULL) && (secondList != NULL))
     || ((firstList != NULL) && (secondList == NULL)))
        return FALSE;

    return TRUE;
}

static void checkNode(TreeNode *syntaxTree)
{
    char errorMessage[100];

    switch (syntaxTree->nodekind)
    {
    case DecK:

        switch (syntaxTree->kind.dec)
        {
        case ScalarDecK:
            syntaxTree->expressionType = syntaxTree->variableDataType;
            break;

        case ArrayDecK:
            syntaxTree->expressionType = Array;
            break;

        case FuncDecK:
            syntaxTree->expressionType = Function;
            break;
        }

        break; /* case DecK */

    case StmtK:

        switch (syntaxTree->kind.stmt)
        {
        case IfK:

            if (syntaxTree->child[0]->expressionType != Integer)
            {
```

```
            sprintf(errorMessage,
                    "IF-expression must be integer (line %d)\n",
                    syntaxTree->lineno);
            flagSemanticError(errorMessage);
        }
        break;

    case WhileK:

        if (syntaxTree->child[0]->expressionType != Integer)
        {
            sprintf(errorMessage,
                    "WHILE-expression must be integer (line %d)\n",
                    syntaxTree->lineno);
            flagSemanticError(errorMessage);
        }
        break;

    case CallK:

        /*  Check types and numbers of formal against actual parameters */
        if (!checkFormalAgainstActualParms(syntaxTree->declaration,
                                           syntaxTree))
        {
            sprintf(errorMessage, "formal and actual parameters to "
                                  "function don\'t match (line %d)\n",
                    syntaxTree->lineno);
            flagSemanticError(errorMessage);
        }
        syntaxTree->expressionType = syntaxTree->declaration->functionReturnType;
        break;

    case ReturnK:

        /* match return type */
        if (syntaxTree->declaration->functionReturnType == Integer)
        {
            if ((syntaxTree->child[0] == NULL) ||
                (syntaxTree->child[0]->expressionType != Integer))
            {
                sprintf(errorMessage, "RETURN-expression is either "
                                      "missing or not integer (line %d)\n",
                        syntaxTree->lineno);
                flagSemanticError(errorMessage);
            }
        }
        else if (syntaxTree->declaration->functionReturnType == Void)
        {
            /* does a return-expression exist? complain */
            if (syntaxTree->child[0] != NULL)
            {
                sprintf(errorMessage, "RETURN-expression must be"
                                      "void (line %d)\n",
```

```
                        syntaxTree->lineno);
            }
        }

        break;

    case CompoundK:

        syntaxTree->expressionType = Void;
        break;
    }

    break; /* case StmtK */

case ExpK:

    switch (syntaxTree->kind.exp)
    {
    case OpK:
        /* Arithmetic operators */
        if ((syntaxTree->op == PLUS) || (syntaxTree->op == MINUS) ||
            (syntaxTree->op == TIMES) || (syntaxTree->op == DIVIDE))
        {
            if ((syntaxTree->child[0]->expressionType == Integer) &&
                (syntaxTree->child[1]->expressionType == Integer))
                syntaxTree->expressionType = Integer;
            else
            {
                sprintf(errorMessage, "arithmetic operators must have "
                                      "integer operands (line %d)\n",
                        syntaxTree->lineno);
                flagSemanticError(errorMessage);
            }
        }
        /* Relational operators */
        else if ((syntaxTree->op == GT) || (syntaxTree->op == LT) ||
                 (syntaxTree->op == LTE) || (syntaxTree->op == GTE) ||
                 (syntaxTree->op == EQ) || (syntaxTree->op == NE))
        {
            if ((syntaxTree->child[0]->expressionType == Integer) &&
                (syntaxTree->child[1]->expressionType == Integer))
                syntaxTree->expressionType = Integer;
            else
            {
                sprintf(errorMessage, "relational operators must have "
                                      "integer operands (line %d)\n",
                        syntaxTree->lineno);
                flagSemanticError(errorMessage);
            }
        }
        else
        {
            sprintf(errorMessage, "error in type checker: unknown operator"
```

```
                                " (line %d)\n",
                        syntaxTree->lineno);
                flagSemanticError(errorMessage);
            }

            break;

        case IdK:

            if (syntaxTree->declaration->expressionType == Integer)
            {
                if (syntaxTree->child[0] == NULL)
                    syntaxTree->expressionType = Integer;
                else
                {
                    sprintf(errorMessage, "identifier is an illegal type "
                                    "(line %d)\n",
                            syntaxTree->lineno);
                    flagSemanticError(errorMessage);
                }
            }
            break;

        case ConstK:

            syntaxTree->expressionType = Integer;
            break;

        case AssignK:

            /* Variable assignment */
            if ((syntaxTree->child[0]->expressionType == Integer) &&
                (syntaxTree->child[1]->expressionType == Integer))
                syntaxTree->expressionType = Integer;
            else
            {
                sprintf(errorMessage, "both assigning and assigned expression"
                                " must be integer (line %d)\n",
                        syntaxTree->lineno);
                flagSemanticError(errorMessage);
            }

            break;
        }

        break; /* case ExpK */

    } /* switch (syntaxTree->nodekind) */
}

static void nullProc(TreeNode *syntaxTree)
{
```

```
    return;
}
```

7.3 实验演示

```
Scope Identifier         Line   Is a   Symbol type
depth                    Decl.  parm?
--- fibonacci ----------------------------------
  2   cnt                   3     N    Scalar of type integer
  2   firstFib              4     N    Scalar of type integer
  2   secondFib             5     N    Scalar of type integer
  2   fib                   6     N    Scalar of type integer
  1   n                     1     Y    Scalar of type integer
--- main ---------------------------------------
  2   n                    31     N    Scalar of type integer
--- GLOBALS ------------------------------------
  0   input                 0     N    Function with return type integer
  0   output                0     N    Function with return type void
  0   fibonacci             1     N    Function with return type integer
  0   main                 29     N    Function with return type void
------------------------------------------------
```