语义分析器实验报告

一、问题描述

【任务名称】语义分析器的设计与实现

【任务性质】主线任务，应在规定时间内完成。

【输入】一个名字类似 source.txt 的纯文本文件，内容是字符串形式的源程序。

【输出】一个名字类似 semantic_out.txt 的纯文本文件，内容要么是对源程序中存在错误的提示，要么是四元式形式的中间代码，它与源程序语义等价。

【题目说明】设计一个程序，基于选定的源语言 LittleC，从输入文件中得到源程序，然后对其依次做词法分析、语法分析、类型检查和代码翻译。若源程序中存在语法错误或类型错误，则报错，结束分析。若源程序不含错误，则将其翻译成四元式形式的中间代码并输出。程序应将所有结果信息写入输出文件。

二、源语言描述

本次实验选择的源语言为伪造的“实验语言”。该语言即老师测试用例提供的 Little C，具有明确的构词规则和单词分类。具体的文法如下所示：

```
PROG        →    {  DECLS  STMTS  }

DECLS       →    DECLS  DECL    |   empty

DECL        →    int  NAMES  ;  |  bool  NAMES  ;

NAMES       →    NAMES ,  NAME  |  NAME

NAME        →    id


STMTS    →    STMTS  STMT  |   STMT

STMT       →    id  =  EXPR ;    |   id := BOOL ;

STMT       →    if  id   then  STMT
```

```
STMT      →    if  id   then  STMT  else STMT

STMT      →    while   id  do  STMT

STMT      →    {  STMTS   STMT  }

STMT      →    read  id  ;

STMT      →    write  id  ;


EXPR    →    EXPR  ADD  TERM  |  TERM

ADD     →    + | -

TERM    →    TERM  MUL NEGA  |  NEGA

MUL     →    * | /


NEGA   →    FACTOR  |  - FACTOR

FACTOR→    (  EXPR ) |  id  |  number


BOOL    →    BOOL  ||  JOIN    |    JOIN

JOIN     →    JOIN   &&   NOT  |   NOT

NOT      →    REL   |  ! REL   | true | false

REL       →    EXPR   ROP  EXPR

ROP      →     >  |  >=  |  <  |  <=  |  ==  |   !=
```

这个文法是一个用于描述一种简单程序语言的上下文无关文法。它定义了这种语言的语法结构，包括程序结构、声明、语句、表达式和布尔表达式的规则。这种语言支持整数（int）和布尔（bool）类型的变量声明，赋值语句，条件语句（if-then 和 if-then-else）、循环语句（while）、读入（read）和写出（write）操作，以及基本的算术和布尔运算。下面是各部分的简要解释：

（1）PROG 表示一个程序，由声明（DECLS）和语句（STMTS）组成，整个程序被大括号包围。

（2）DECLS 和 DECL 定义了如何声明变量。可以声明整数（int）和布尔（bool）类型的变量。变量名通过 NAMES 和 NAME 规则定义，支持单个或多个变量的声明。

（3）STMTS 和 STMT 描述了语句的结构。语句可以是赋值语句（包括算术表达式赋值和布尔表达式赋值）、条件语句（包括带有和不带有 else 的 if 语句）、循环语句（while）、复合语句（用大括号括起来的语句序列）、读入和写出语句。

（4）EXPR, TERM, NEGA, 和 FACTOR 规则定义了算术表达式的结构，支持加减乘除和负号操作，以及括号用于改变运算顺序。

（5）BOOL, JOIN, NOT, 和 REL 规则定义了布尔表达式的结构，支持逻辑运算（与、或、非）和关系运算（比较大小、等于、不等于），且定义了 true 和 false。

这个文法通过组合这些基本构建块，可以描述一个相当丰富的编程语言特性集，包括变量声明、算术和布尔运算、条件判断、循环控制、以及输入输出操作。

三、程序算法设计

整体思路

实际编码过程中并未按照学习通的参考思路去做，原因在于实现的复杂程度和效率的提升。

先从局部的角度分析：每一种语句我们从根节点出发重新遍历，然后判断分支节点语句类型做语义检查，这样做和建树的递归下降大同小异，只不过附带了其他的一些语义动作，其次相当于第二次遍历树，复杂度也提高了。

再从整体的角度分析：由于我们的整个程序是由各种语句混合而成，在建树的时候可以很清晰的根据词法分析器的结果得知是哪种语句并建树，但是如果是抽象语法树的根节点，我们对于各种语句就需要进行判断。而针对执行语句还要考虑是 while/if 内部的，还是 Program 中的，我们一开始构建语法树是很清晰的，但是自顶向下遍历需要传递额外的语义信息。

于是我们先是在建树的过程中，添加了一些静态检查的部分，然后 copy 了一份 task2 的头文件到 task3。在 task3 中写了对应的翻译代码命名为 xxxGen 函数，边建树边翻译。当语法树建完了，报错信息在建树的过程中输出到文件，反之，通过一个全局变量决定是否输出翻译代码，实现建树、静态检查、翻译一趟完成。

模块设计

驱动模块

（1）输入模块：

```
void inputLex(int start, int end) {
    for (int index = start;index <= end;index++) {
        string inname = "../source/sourceProgram" + to_string(index) + ".txt";
        string outname = "input/lex_out" + to_string(index) + ".txt";
        // 如果存在 outname 文件，说明已经进行过词法分析
        if (filesystem::exists(outname)) {
            cout << outname << " exists" << endl;
            continue;
        }
        single_process(index, "../source/sourceProgram", "input/lex_out"); // 词法分析
    }
}
```

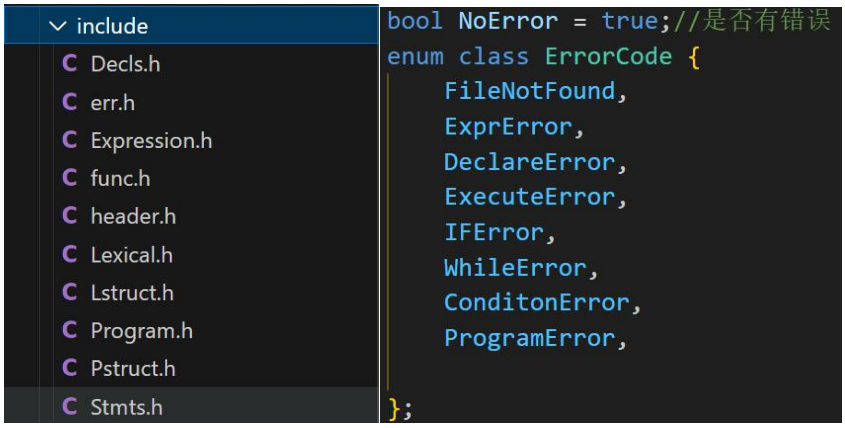
这里的 inputLex 函数用于词法分析，调用 single_process 将源程序中的代码转变成二元组形式，并以文件的形式保存，如果已经存在重名文件不予处理。

（2）输出模块：

```
void outputCode(int start, int end, string inPath, string outPath, int flag = 0) {
    if (flag == 0)
        tran2Terminal(start, end, inPath, outPath);
    else
        tran2File(start, end, inPath, outPath);
}
```

这里 outputCode 函数用于输出代码翻译的结果，将输入模块保存的二元组作为输入，然后同步构建语法树、静态语义检查和代码翻译（这里涉及到诸多封装的关系，具体请参考源码中的 program.h、func.h 等头文件），同时为了方便测试和运行分别封装了输出流在终端或者文件的函数 tran2Terminal 和 tran2File。

词法分析模块



词法分析采用的依然是头文件的形式，将源文件 lexical.cpp 的代码重构成 Lexical.h 的算法头文件以及 LStruct.h 的数据结构头文件。

错误分析模块

在老师的建议下，我把对于静态语义检查和语法检查的可能发生的错误类型放在了 err.h 文件夹下。该头文件先定义了一个异常的基础类型 BaseException，然后根据错误类型分为 FileException、DeclareException 等继承该基础异常。当程序发现了错误，要么在当前程序中 catch，要么在调用它的父亲程序中 catch 到，不会出现被 terminated 的情况。这样的设计可以将正确代码和错误分开管理，大大提高了代码的可维护性，很感谢老师的优质建议。

综合处理模块

由于我们将构建语法树、静态语义检查和代码翻译放在了一起，所以语法分析、语义检查和代码翻译模块没有独立出来。这里简单说下他们的构成和关系。

构建语法树我统一采用的是 build_xxx_tree 的函数命名，将算术表达式和布尔表达式的相关代码放在 Expression.h 中，将声明语句的代码放在 Decls.h 中，将执行语句（包含赋值语句和 I/O 语句）、if 语句、while 语句等放在了 Stmts.h 头文件下。

在构建抽象语法树的同时进行静态检查，包括检查 ID 是否声明，类型是否匹配、是否已有确定值等，然后利用 try...catch...机制捕获出现的异常，并重定向 cerr 输出流，输出相关错误信息到文件中。检查完之后，我们拿到主函数中 Stmts 函数构建好的各种语句根节点，通过调用执行、if、while 语句对应的翻译函数递归下降去翻译该语句，添加相应的四元组到结果数组。

这样我们就在建树的同时完成了代码翻译的工作，没有对整个语法树再次进行遍历。当然，这样做也有缺陷：即使静态语义检查失败，代码翻译工作也会进行，只不过对于最后的输出结果，我们使用了全局变量 NoError，用它来确定之前的静态检查状态，为 true 才输出四元组。

输出四元组的格式形如：（+，3，i，t1），这里+表示算术加，第 1 个操作数是整常量 3，第 2 个操作数是整形变量 i，运算结果放入临时变量 t1 中。其中，条件判断中我们引入了 JT、JF、JN，JT 表示有条件跳转，JF 紧跟其后是无条件跳转，JN 是 if 和 else 中间执行完 if 跳过 else 的无条件跳转。这里的 read/write 语句，我们没有做特殊的处理，只是将其形式化，形如 (read_, _, a)。

四、程序运行和测试

本次实验使用老师学习通中提供的 10 个测试，并在成功通过所有的用例添加了两个测试用例 11 和 12。下面我来一一介绍程序测试的结果和一些细节，测试中间生成的词法分析结果将不再展示。

1. sourceProgram1:

```
//program 1: add two numbers.
{
        int a, b, c  ;
        a = 1;
        b = 2;
        c = a + b ;
}
```

输出:

```
1: (=,1,_,a)
2: (=,2,_,b)
3: (+,a,b,t1)
4: (=,t1,_,c)
```

分析:

我们刨除声明语句，对于三条执行语句进行翻译。‘=’右部如果是常数则直接赋值，如果是表达式，将结果先复制到临时寄存器 t1，然后再从 t1 复制到 c。

2. sourceProgram2:

```
//program 2: do some calculation.
{
        int a, b, c  ;
        a = 5; //positive number
        b = -3; //negative number
```

```
        c = (a+b)*(a-b);//calculation
        write c  ; //output:16.
}
```

输出：

```
1: (=,5,_,a)
2: (=,-3,_,b)
3: (+,a,b,t1)
4: (-,a,b,t2)
5: (*,t1,t2,t3)
6: (=,t3,_,c)
7: (write,_,_,c)
```

分析：

对于 c 的赋值语句，首先需要计算右部表达式的值然后才能进行赋值，按照顺序，那就是先把 a+b 的结果存在 t1 中，把 a-b 的结果存在 t2 中，然后把 t1 和 t2 相乘的结果放到 t3 中，最后把 t3 复制到 c 中。

3. sourceProgram3：

```
/*program 3: add numbers from 1 to 100
 *and print the result.
 */
{
        int a , sum ;
        bool  b ;
        a = 1 ;
        sum = 0 ;
        b := a <= 100 ;
        while b do
        {
        sum = sum + a ;
        a = a + 1 ;
        b := a <= 100 ;
        }
        write  sum  ;
}
```

输出：

```
1: (=,1,_,a)
2: (=,0,_,sum)
```

3: (<=,a,100,t1)
4: (:=,t1,_,b)
5: (JT,b,_,7)
6: (JF,_,_,14)
7: (+,sum,a,t1)
8: (=,t1,_,sum)
9: (+,a,1,t1)
10: (=,t1,_,a)
11: (<=,a,100,t1)
12: (:=,t1,_,b)
13: (JT,b,_,5)
14: (write,_,_,sum)

分析：

这里主要针对 while 语句进行了翻译。

先判断 b 是否为真(JT,b,_,7)，如果为真跳转到第七行(JT,b,_,7)，反之跳出循环到第十四行(JF,_,_,14)，中间的_,_代表无条件跳转。然后是循环体内部的各种执行语句，循环体结束时判断(JT,b,_,7)，如果 b 为真就跳回循环开始处第五行，反之退出循环执行 I/O 语句。

4. sourceProgram4:

```
{
        int a,b,c;
        int lg;
        bool cond;

        read  a; read  b; read  c;
        cond := a > b  ;
        if  cond  then  lg = a ;
        else  lg = b ;
        cond := lg < c ;
        if  cond  then  lg = c ;

        write lg  ;
}
```

输出：

1: (read,_,_,a)
2: (read,_,_,b)

3: (read,_,_,c)

4: (>,a,b,t1)

5: (:=,t1,_,cond)

6: (JT,cond,_,8)

7: (JF,_,_,10)

8: (=,a,_,lg)

9: (JN,_,_,11)

10: (=,b,_,lg)

11: (<,lg,c,t1)

12: (:=,t1,_,cond)

13: (JT,cond,_,15)

14: (JF,_,_,16)

15: (=,c,_,lg)

16: (write,_,_,lg)14: (JF,_,_,16)

分析：

这里主要是对于 if 条件判断的测试。

第六到第十一行，(JT,cond,_,8)先判断 cond 是否为真，真则跳转到第八行，执行完第八行无条件跳转到第十一行（跳过 else 部分），反之(JF,_,_,10)跳转到第九行执行 else 语句，然后顺序执行。

5. sourceProgram5:

```
/* program 5: find all numbers which is
 * divisible by 3   between 1 and 12 .
 */
{
        int   number, res ;
        bool   cond1, cond2   ;

        number = 1 ;
        cond1 := number <= 12 ;
        while cond1 do
        {
        res = number - ( number / 3 ) * 3 ;
        cond2 := res == 0   ;
        if cond2 then write number ;
        number = number + 1 ;
        cond1 := number <= 12 ;
        }
}
```

输出：

1: (=,1,_,number)
2: (<=,number,12,t1)
3: (:=,t1,_,cond1)
4: (JT,cond1,_,6)
5: (JF,_,_,20)
6: (/,number,3,t1)
7: (*,t1,3,t2)
8: (-,number,t2,t3)
9: (=,t3,_,res)
10: (==,res,0,t1)
11: (:=,t1,_,cond2)
12: (JT,cond2,_,14)
13: (JF,_,_,15)
14: (write,_,_,number)
15: (+,number,1,t1)
16: (=,t1,_,number)
17: (<=,number,12,t1)
18: (:=,t1,_,cond1)
19: (JT,cond1,_,4)

分析：

这里主要是对于 while 里面嵌套 if 语句的测试。

首先是 while 循环条件 cond1 的判断(JT,cond1,_,6)，判断条件不满足(JF,_,_,20)，即跳出整个循环。满足当前条件就进入循环体，执行到第 12 行对 cond2 的判断，如果为 true 则跳转 14 行执行真分支，否则 (JF,_,_,15)跳出真分支，由于没有假分支，正常往下执行。最后到循环体末尾，再次判断 cond1 是否为真，为真跳回第 4 行循环入口。

6. sourceProgram6：

```
//program 6: no decls in program .
//NO errors in Syntax Checking.
//Undeclared id errors should be reported in Symantic Checking.
{
    //int a; bool b;
    a = -1 ;
    b := a <= 0;
    write a;
}
```

输出：

执行语句左部 a 未声明

执行语句左部 b 未声明

赋值表达式中 a 未声明

write 对象 a 未声明

静态检查失败，编译不通过！

分析：

设计 err 的时候考虑比较细致，对于执行语句左部，赋值表达式，I/O 对象都有检查变量是否声明，同时报错信息注明是在哪一块地方出现的问题，便于用户去查找和修改错误。

7. sourceProgram7：

```
//program 7: read    in two numbers and do some calculations.
{
        int a, b, c    ;
        read a ;
        read b ;
        //Test Expr().
        a = a + 1 + 2 * 3 * 4 ;
        b = b * 8 / 2 / 4 ;
        c = ( a + b ) * ( a - b ) ;

        write c    ;
}
```

输出：

1: (read,_,_,a)

2: (read,_,_,b)

3: (+,a,1,t1)

4: (*,2,3,t2)

5: (*,t2,4,t3)

6: (+,t1,t3,t4)

7: (=,t4,_,a)

8: (*,b,8,t1)

9: (/,t1,2,t2)

10: (/,t2,4,t3)

11: (=,t3,_,b)

12: (+,a,b,t1)

13: (-,a,b,t2)

14: (*,t1,t2,t3)

15: (=,t3,_,c)

16: (write,_,_,c)

分析：

这段代码的主要功能是从用户那里读取两个整数 a 和 b，执行一系列计算，并最终输出一个结果。计算涉及到基本的算术操作，包括加法、乘法和除法。与之前的用例并无不同，这里不再详细分析。

8. sourceProgram8：

```
/* program 8: input three numbers ,output the largest one.
 * Test Ambiguous Statement : Nested If_then and If_Then_Else .
 */
{
        int a,b,c ;
        bool cond1,cond2,cond3;
        read a; read b; read c;
        cond1 := a >= b ;
        cond2 := a >= c ;
        cond3 := b >= c ;

        if cond1 then
        if cond2 then write a ;
        else write c ;
        cond1 := a < b ;
        if cond1 then
        if cond3 then write b ;
        else write c ;
}
```

输出：

1: (read,_,_,a)

2: (read,_,_,b)

3: (read,_,_,c)

4: (>=,a,b,t1)

5: (:=,t1,_,cond1)

6: (>=,a,c,t1)

7: (:=,t1,_,cond2)

8: (>=,b,c,t1)

9: (:=,t1,_,cond3)
10: (JT,cond1,_,12)
11: (JF,_,_,17)
12: (JT,cond2,_,14)
13: (JF,_,_,16)
14: (write,_,_,a)
15: (JN,_,_,17)
16: (write,_,_,c)
17: (<,a,b,t1)
18: (:=,t1,_,cond1)
19: (JT,cond1,_,21)
20: (JF,_,_,26)
21: (JT,cond3,_,23)
22: (JF,_,_,25)
23: (write,_,_,b)
24: (JN,_,_,26)
25: (write,_,_,c)

分析：

这里的涉及到 if if else 嵌套结构的测试。

首先是对于 else 的归属问题，采用就近原则，else 匹配最近的 if，所以对于第一个 if 的 else 为空，则没有 JN，而第二个 if 有 else 就有 JN。

我们挑第 19 到 25 行来说，首先是第一个 JT 判断 cond1，为真就跳转 21 行执行真分支，反之执行假分支，假分支为空，就跳到了第 26 行结束。而真分支又是一个 if 语句，判断 cond3，为真跳转 23 行执行子真分支，执行完跳到第 26 行(JN,_,_,26)，反之执行第 25 行假分支，顺序执行到 26 行结束。

9. sourceProgram9：

```
//program 9: empty program.
{
}
```

输出：

程序不合法：缺少执行语句
静态检查失败，编译不通过！

10. sourceProgram10：

```
//program10: test prime numbers.
```

```
{
    int number,i,j;
    bool cond,flag;
    read number;

    i = 2;
    flag = true;
    cond := i<number && flag;

    while cond do
    {
            j=number-(number/i)*i;
            flag:= j!=0;
            i=i+1;
            cond := i<number && flag;
    }
    if flag then write number;
}
```

输出：

```
flag:类型不匹配=运算符
静态检查失败，编译不通过！
```

分析：

由于代码静态检查有报错，所以没有输出代码翻译的内容。

while 循环和赋值语句之前有出现过，本质与之前的用例无不同。

11. sourceProgram11:

```
//program 11: read in two numbers and do some calculations.
{
    int a, b, c   ;
    read a ;
    read b ;
    //Test Expr().
    a = a + 1 + 2 * 3 * 4 ;
    b = b * 8 / 2 / 4 ;
    c = ( a + b ) * ( a - b   ;  //missing ')' here to test error handling
    write c   ;
}
```

输出：

```
括号不能参与运算
(运算符不合法
静态检查失败，编译不通过!
```

分析：

把之前的用例拿来并去除了一个右括号，建立表达式子树的时候遇到分号进行规约的时候发现没有匹配的(，

又不能把它当做运算符，于是提示报错。

12. sourceProgram12:

```
// Program to print prime numbers between 1 and 100
{
    int number, i, j;
    bool flag, cond, cond1;
```

```
    number = 2;
    cond := number <= 100;
    while cond do
    {
        i = 2;
        flag := true;
        cond1 := i < number && flag;
        while cond1 do
        {
            j = number-(number/i)*i;
            flag:= j!=0;
            i = i + 1;
            cond1 := i < number && flag;
        }
        if flag then write number;
        number = number + 1;
        cond := number <= 100;
    }
}
```

输出：

```
1: (=,2,_,number)
2: (<=,number,100,t1)
```

3: (:=,t1,_,cond)

4: (JT,cond,_,6)

5: (JF,_,_,33)

6: (=,2,_,i)

7: (:=,true,_,flag)

8: (<,i,number,t1)

9: (&&,t1,flag,t2)

10: (:=,t2,_,cond1)

11: (JT,cond1,_,13)

12: (JF,_,_,25)

13: (/,number,i,t1)

14: (*,t1,i,t2)

15: (-,number,t2,t3)

16: (=,t3,_,j)

17: (!=,j,0,t1)

18: (:=,t1,_,flag)

19: (+,i,1,t1)

20: (=,t1,_,i)

21: (<,i,number,t1)

22: (&&,t1,flag,t2)

23: (:=,t2,_,cond1)

24: (JT,cond1,_,11)

25: (JT,flag,_,27)

26: (JF,_,_,28)

27: (write,_,_,number)

28: (+,number,1,t1)

29: (=,t1,_,number)

30: (<=,number,100,t1)

31: (:=,t1,_,cond)

32: (JT,cond,_,4)

分析：

这个用例是一个嵌套循环结构的测试。

其中外部循环控制变量为 number，内部循环控制变量为 i。

1. 外部循环

指令 4-32 描述了外部循环的逻辑：

指令 4: 如果条件 cond 为真（外部初始化或上一次循环迭代设置），则跳转到指令 6，否则直接进入指令 6

指令 32: 检查 number 是否小于等于 100。如果是，设置 cond 为真，并跳回指令 4，继续外循环。

2. 内部循环

指令 6-24 描述了内部循环的逻辑：

指令 6-20: 初始化和更新循环变量 i。将 i 赋值为 2（指令 6），然后 flag 设置为 true（指令 7）。循环每次迭代 i 都增加 1（指令 19-20）。

指令 8-10: 检查 i 是否小于 number，并且 flag 为 true。如果这两个条件都满足，将 cond1 设置为 true（指令 10）。

指令 11-24: 如果 cond1 为真，继续执行到指令 13-18，这部分代码可能检查 number 是否可以被 i 整除（一个简单的因数检查）。如果 j（指令 16 的结果）不等于 0（即 number 不能被 i 整除），则设置 flag 为 true（指令 18）。最后，如果 cond1 仍为真，跳转回指令 11 继续内循环。

3. 终止条件和输出

指令 25-28: 如果 flag 仍为真（意味着 number 没有其他因数，可能是一个素数），则打印 number（指令 27）。之后 number 增加 1（指令 28-29），并再次检查是否继续外部循环（指令 30-32）。

五、实验总结与心得

实验结束之后，我通过 VSCode 中的代码统计插件统计代码总数，记录本次实验的工程量。

Languages

language	files	code	comment	blank	total
C++	16	1,614	269	134	2,017

language	files	code	comment	blank	total
C++	13	1,375	251	112	1,738

从一开始将老师的 10 个测试用例通过的 2000+行左右代码，到后来经过老师点播修改并逐渐整合优化完善的 1700+行代码，记录了实验过程一路走来的点点滴滴。诚如老师在任务 3.1 中所说，整个编译器前端的开发过程是一个大瀑布模型，具体到每一个阶段，其分析、设计、编码、测试、发布等也是一个小瀑布模型。在本科阶段，与以往的实验课程不同，编译器前端设计的工程性质更强，也更加连贯，让学生在开发的过程中不断发现问题并修改问题，最后迭代到一个相对较好的版本，给与了学生极大的成就感。

经过这次实验，我对静态语义检查和代码翻译有了更加深入的理解。我意识到，这不仅仅是计算机科学的编译原理的重要理论，更是两个相互关联、相辅相成的过程。静态语义检查为代码翻译提供了前提和基础，而代码翻译则是静态语义检查的一种应用和扩展。

同时，我也意识到自己在某些方面还存在不足。例如，对于某些复杂的编程语言特性，我还需要进一步的学习和研究。对于项目的整体设计还需进一步思考，尽可能采用面向对象的设计思路进行改进。但我相信，只要我保持好奇心和学习的热情，就一定能够克服这些困难，取得更大的进步。

展望未来，我希望能够参与到更多类似的实验中，通过实践来不断提升自己的能力和水平。

总的来说，这次实验是一次非常有价值的经历。它不仅让我更加深入地理解了静态语义检查和代码翻译的原理和实现，更让我体会到了实践和探索的乐趣。我相信，这次实验将成为我学习和成长过程中的一笔宝贵财富。