同济大学

TONGJI UNIVERSITY

《编译原理》
设计说明

作业名称	词法和语法分析工具
小组成员	2153603 王政尧
	2153683 郭嘉
	2154071 张博文
学院（系）	电子与信息工程学院
专　　业	计算机科学与技术
任课教师	丁志军
日　　期	2023 年 11 月 26 日

一、 整体设计

本次作业完成的词法、语法分析器的工作过程为：先由词法分析器对源程序文件进行读取，解析单词符号为二元组形式，并将其送入语法分析器；语法分析器按照给定文法生成 LR(1)规范项目集族，再生成 ACTION 表和 GOTO 表，利用两个表对给定输入串进行移进规约分析；由 UI 界面显示各项内容与移进规约的进栈出栈过程。

二、 词法分析实现

2.1 原理概述

缺乏系统整体设计图，展示各模块间关系

词法分析的功能是读入源程序文件，对程序中的每个单词进行判别和分类，并以二元组的形式进行输出。

单词可被分为五大种类：关键字、标识符、常数、运算符和界符。除此以外的单词符号被认为是未定义符号。

关键字是程序预留的有固定意义的标识符，其它标识符不允许与其重名。在本次作业中，关键字共有六个，分别是：”int”，”void”，”if”，”else”，”while”，”return”。

标识符用于对变量、函数等命名，参考 C 语言的标识符命名要求，本次作业的命名要求为：标识符需以字母（大小写均可）开头，并且只包含字母和数字。

常数即数字，本次作业未考虑小数，因此常数仅包含整数。

运算符为各种运算符号以及关系符号，运算符号包含 +、-、*、/ 与赋值符号 =，关系符号包含 >、<、>=、<=、==、!=。

界符起到分隔作用，包含 ;、，，此外还包含三种括号 ()、[]、{} 以及表示结束的符号 #。

2.2 总体设计

词法分析的过程可以表示成如下三个流程：读文件存入缓存区，以空格、换行符等为分界将单词一一分隔出，判断单词类别并以二元组形式输出。

2.3 详细设计

首先需要预先存储关键字，并为各单词安排种别码以示区分。关键字可用一 string 数组进行存储；种别码可用 enum 类型存储，这里为每个关键字、符号安排了不同的种别码，为所有标识符安排了一个种别码，为数字安排了一个种别码，为未定义符号安排了一个种别码。共 32 种。

由于个人编程习惯的原因，并未严格按照以上三个过程编写程序，而是采用一边读、一边判断并输出的形式完成。

用 while 循环逐个读取字符，每次开始新的循环表示在读取一个新的单词。利用

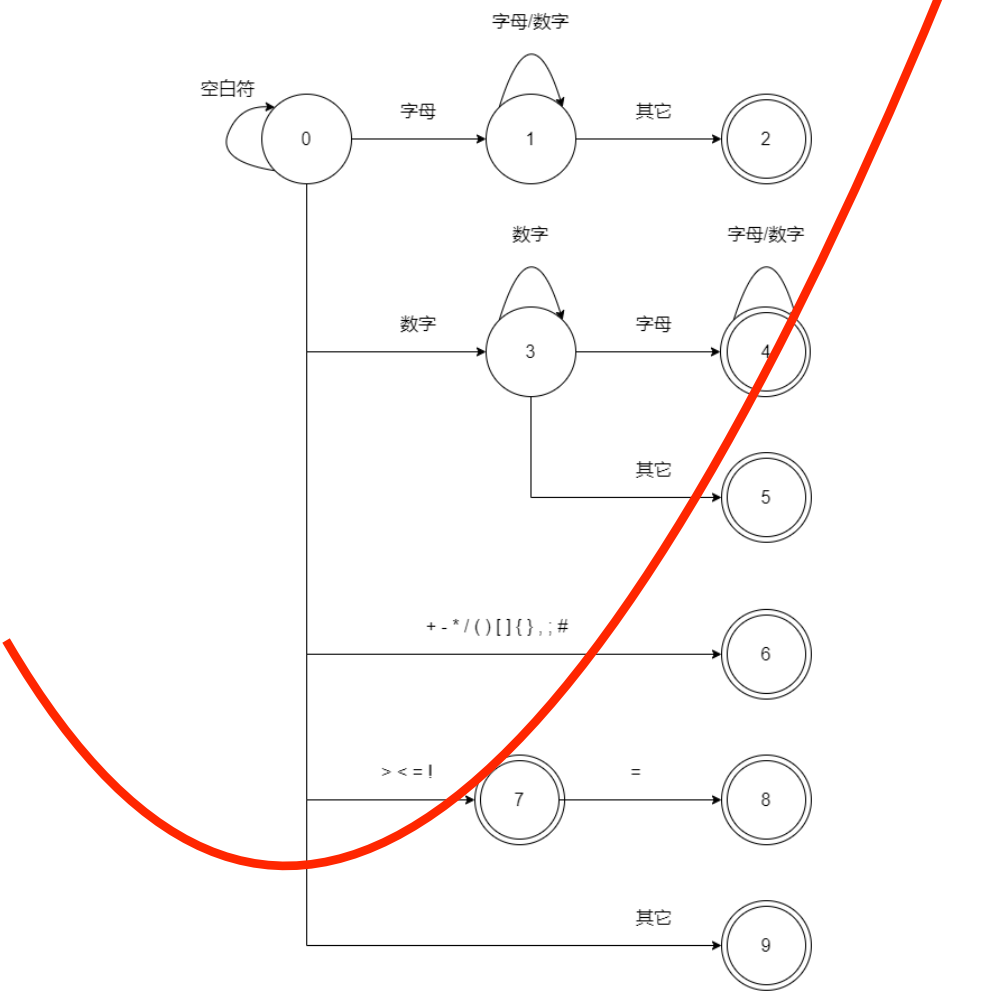
每个单词的第一个读入字符来判断单词的种类：如果读入了字母，则该单词可能为标识符或关键字；如果读入了数字，则该单词可能为常量；如果读入了符号，则进一步判断符号种类。下面按这三种情况进行分类讨论。

如果读入了字母，则不断地进行超前读取，如果是字母或数字则读取，并将其加入已经读到的字符串，如果不是则停止。之后对读到的字符串进行判断，如果是关键字，则置种别码为对应关键字，否则置种别码为标识符。超前搜索可利用 C++的 fin.peek()实现，它会读取文件指针的下个字符，并且不会移动文件指针。

如果读入了数字，则不断地进行超前读取，如果是数字则读取，并将其加入已经读到的字符串，如果读到了字母，则表示这是一个非法的单词符号。读取直到遇到数字、字母以外的字符才停止，根据读到的字符串，置种别码为数字或未定义类型。

如果读入了符号，利用 switch 功能对符号进行判断，需要注意 >、<、=、！符号可能是双字符的符号，需要进行一次超前读取。之后置种别码为对应符号。

以上三种情况可表示为如下状态转换图。



装 订 线

在每种情况下，一旦读取到了非本种情况可能遇到的字符，就停止读取，完成后续处理，将得到的二元组加到存储结果的数组种，之后开始下一次循环。

词法分析全部完成后，得到一个二元组的数组，将其送给语法分析器，完成之后的分析工作。

三、 语法分析实现

缺乏原理概述

3.1 获取全部终结符与非终结符

3.1.1 函数接口

```
// 计算终结符和非终结符
getTerminalAndNonTerminalSymbols(productions);
```

3.1.2 实现思路

我们将全部的文法产生式通过 vector 集合的形式预先保存起来，如下图形式：

```
// 定义文法产生式集合
vector<Production> productions = {
    Production("Start", {"Program"}),

    // Program产生式
    Production("Program", {"Type", "idt", "(", "FormalParameters", ")", "StatementBlock"}),

    // Type产生式
    Production("Type", {"int"}),
    Production("Type", {"void"}),
```

其中的 Production 使用 struct 结构体来分别保存他的左部和右部：

```
// 定义文法产生式结构体
struct Production {
    string left; // 左部
    vector<string> right; // 右部

    Production() {

    }
    Production(string l, vector<string> r) : left(l), right(r) {}

    bool operator==(const Production& other) const {
        return (
            left == other.left &&
            right == other.right
            );
    }
};
```

这样一来，访问一个文法产生式便可以以 production.left -> production.right 形式快捷表示。

我们两次遍历文法产生式的集合。第一次遍历将所有产生式左部的非终结符号加入到 NonTerminal 集合中，第二次遍历我们将所有产生式右部出现的终结符和未出现在左部的终结符加入到 Terminal 集合中。这样就实现了获取终结符和非终结符的功能。

3.1.3 主要功能代码

```
// 获取文法中的所有符号（终结符和非终结符）
void getTerminalAndNonTerminalSymbols(vector<Production> productions) {
    // 两次遍历产生式集合，将所有符号分别加入到对应集合中
    for (const auto& production : productions) {
        // 将产生式左部（非终结符号）加入到符号集合中
        nonTerminals.insert(production.left);
    }
    for (const auto& production : productions) {
        // 将产生式右部的符号加入到符号集合中
        for (const auto& symbol : production.right) {
            if (symbol != "ε" && nonTerminals.find(symbol) == nonTerminals.end()) {
                terminals.insert(symbol);
            }
        }
    }

    return;
}
```

3.2 根据终结符和非终结符构建符号表

3.2.1 函数接口

```
// 获取全部符号表
G = getAllSymbols();
```

3.2.2 实现思路

首先定义一个 unordered_set 类的函数，这个函数的返回值即为全部符号表；实现过程非常简单，只需要合并终结符集合和非终结符集合即可。

装 订 线

```
// 遍历非终结符集，将所有符号加入到集合中
for (const auto& symbol : nonTerminals) {
    symbols.insert(symbol);
}

// 遍历终结符集，将所有符号加入到集合中
for (const auto& symbol : terminals) {
    symbols.insert(symbol);
}
```

3.2.3 主要功能代码

```
    // 获取文法中的所有符号（终结符和非终结符）
unordered_set<string> getAllSymbols() {
    unordered_set<string> symbols;

    // 遍历非终结符集，将所有符号加入到集合中
    for (const auto& symbol : nonTerminals) {
        symbols.insert(symbol);
    }

    // 遍历终结符集，将所有符号加入到集合中
    for (const auto& symbol : terminals) {
        symbols.insert(symbol);
    }

    return symbols;
}
```

3.3 初始化全部项目

3.3.1 函数接口

```
// 项目初始化
getAllItem(productions);
```

3.3.2 实现思路

首先，定义一个 LR(1)项目：

```
// LR(1) 项目结构体
struct LR1Item {
    Production production; // 产生式
    int dotIndex; // 点的位置
    string lookahead; // 向前查看符号
```


装 订 线

我们在这里规定，一个项目由三部分组成：产生式、点的位置和向前查看符号。进而定义一个 LR(1)项目集：

```
// LR(1) 项目集
vector<LR1Item> LR1ItemSet;
```

在此基础上，遍历所有的产生式，默认初始所有产生式的 lookahead 为“#”。

```
ct.dotIndex = count;
ct.lookahead = "#";
LR1ItemSet.push_back(ct);
```

输出监视产生的所有项目情况：

```
Start -> . Program
Start -> Program .
Program -> . Type idt ( FormalParameters ) StatementBlock
Program -> Type . idt ( FormalParameters ) StatementBlock
Program -> Type idt . ( FormalParameters ) StatementBlock
Program -> Type idt ( . FormalParameters ) StatementBlock
Program -> Type idt ( FormalParameters . ) StatementBlock
Program -> Type idt ( FormalParameters ) . StatementBlock
Program -> Type idt ( FormalParameters ) StatementBlock .
```

产生成功！

3.3.3 主要功能代码

```
// 获取全部项目，构建项目集
void getAllItem(vector<Production> productions) {
    for (auto production = productions.begin(); production != productions.end(); production++) {
        LR1Item ct;
        ct.production = *production;
        for (int count = 0; count <= production->right.size(); count++) {
            ct.dotIndex = count;
            ct.lookahead = "#";
            LR1ItemSet.push_back(ct);
        }
    }
```


装 订 线

3.4 计算 First 关系集

3.4.1 函数接口

```
// 计算First关系
calculateFirst();
```

3.4.2 实现思路

While 一直循环遍历全部产生式，直到每个非终结符的 First 集不再增大为止。为了解决文法中存在左递归的问题，我会循环计算很多次；如果产生式右部第一个非终结符的 First 集不为空，则把它加入左部的 First 集；如果为空则直接跳过。如果是终结符，直接插入即可。如此循环。

```
// If the next symbol is a terminal, add it to the First set
if (!isNonTerminal(nextSymbol))
{
    if (!First[leftSymbol].count(nextSymbol)) {
        firstSet.insert(nextSymbol);
        flag = false;
    }

}
```

3.4.3 主要功能代码

```
// 计算First集
void calculateFirst()
{
    // Get the production rules associated with the symbol
    while (1) {
        bool flag = true;
        for (const auto& production : productions)
        {
            // Check the first symbol in the production rule
            string leftSymbol = production.left;
            string nextSymbol = production.right[0];
            set<string> firstSet;
            // If the next symbol is a terminal, add it to the Fi
rst set
            if (!isNonTerminal(nextSymbol))
            {
                if (!First[leftSymbol].count(nextSymbol)) {
                    firstSet.insert(nextSymbol);
                    flag = false;
```

```
                }

            }
            else
            {
                if (!First[nextSymbol].empty()) {
                    for (auto it = First[nextSymbol].begin(); it
!= First[nextSymbol].end(); it++) {
                        if (!First[leftSymbol].count(*it)) {
                            firstSet.insert(*it);
                            flag = false;
                        }
                    }
                }
            }
            if (!firstSet.empty()) {
                First[leftSymbol].insert(firstSet.begin(), firstS
et.end());
            }
        }
        if (flag) { break; }
    }
    return ;
}
```

3.5 计算项目集规范族与 Go 转移关系

3.5.1 函数接口

```
// 0#项目集算闭包
outclosure(0, *LR1ItemSet.begin());

// 计算Go关系
calGo();
```

3.5.2 实现思路

3.5.2.1 算闭包

我将闭包的计算分为了两种：一种是初始项目在当前闭包的，我使用 inclosure 计算当前项目集闭包；另一种是初始项目不在当前闭包的，由另一闭包或外部转移过来的，我使用 outclosure 计算闭包。

装 订 线

```
// 内部算闭包
int inclosure(int curNum, LR1Item curItem, LR1Item lastItem) {
```

```
// 从另一闭包转移到当前闭包
int outclosure(int curNum, LR1Item curItem) {
```

对于内部算闭包，当前项目的进栈过程会有上一个项目参与，即当前项目的 lookahead 取决于上一项目产生式的 dotindex 对应的符号串的 First 集合。而对于外部算闭包则无需考虑这个问题，直接将当前项目进栈即可。

计算的中间过程，我单独提取出了过程函数 enterprocess：

```
// 算闭包的中间过程
void enterprocess(int curNum, LR1Item tmp) {
```

如果当前项目不在当前项目规范集中，则插入当前。接下来，如果当前项目产生式的 dot 不在末尾，且 dot 后第一个符号为非终结符，则继续拓展计算，在项目集中检索产生式左部为对应非终结符，且 dot 在产生式开始的项目。如果检索到，则该项目作为新的当前项目，上一个当前项目作为 lastItem 进入 inclosure 进行递归计算，直至结束。

```
if (nextp->production.left == nextsymbol && nextp-
>dotIndex == 0) { //s->.AB
    inclosure(curNum, *nextp, tmp);
}
```

3.5.2.2 算 Go 转移关系

对于 Go 转移关系的计算，主要参考了课本 P.115 的构造算法：

关于文法 G'的 LR(1)项目集族 C 的构造算法是：

BEGIN

C: = {CLOSURE({[S'→·S, #]})};

REPEAT

FOR C 中的每个项目集 I 和 G'的每个符号 X DO

IF GO(I,X)非空且不属于 C,THEN 把 GO(I,X)加入 C 中

UNTIL C 不再增大

END

首先遍历所有已存在的项目规范集，随后遍历每个符号，查找是否存在 dot 后为当前符号的项目，找到后该项目进入 outclosure 外部计算闭包。注意到，这样产生的闭包可能会有重复，无法正确获得项目集的包含关系和转移关系，所以在每一个每一

族生成一族新的规范集后，都要进行是否属于已存在项目集的操作。

```
if (have) {
    if (checkBelongto(canonicalCollection[nextNumber]) != -1) {
        // 转移关系进栈
```

我们再定义 Go 转移关系：

```
// Go表
struct Go {
    int pre;
    int next;
    int exist = -1;
    string changeStr;
```

其中，pre 表示父项目集号，next 表示子项目集号，exist 表示存在该关系，changeStr 表示转移的符号。

```
// 判断是否属于先前项目集
int checkBelongto(vector<LR1Item> v) {
    for (int i = 0; i < canonicalCollection.size() - 1; i++) {
        if (isSubset(canonicalCollection[i], v)) {
            return i;
        }
    }
    return -1;
}
```

检查是否属于的过程中，如果确实存在，则将转移关系的 next 赋值为已存在的项目集号：

```
Go tmp;
tmp.pre = i;
tmp.next = checkBelongto(canonicalCollection[nextNumber]);
tmp.exist = 1;
tmp.changeStr = *go_symbol;
if (!findVec(tmp)) { go.push_back(tmp); }
```

同时，删除新计算得到的闭包和项目集：

装 订 线

```
                // 若属于先前，则删除当前项目集
                Exist[nextNumber] = 0;
                canonicalCollection[nextNumber].clear();
```

如果不属于已存在的项目集，则正常关系进栈即可，无需其他操作。一直循环计算到项目集规范族不再增大为止。

3.5.3 主要功能代码

```
// 内部算闭包
int inclosure(int curNum, LR1Item curItem, LR1Item lastItem) {
    Exist[curNum] = 1; //当前族置存在
    LR1Item tmp = curItem;
    string head;

    if (lastItem.dotIndex + 1 < lastItem.production.right.size())
 { // 有后续串+lookhead
        head = lastItem.production.right.at(lastItem.dotIndex + 1
);
        if (isNonTerminal(head)) //如果是非终结符
        {
            for (auto it = First[head].begin(); it != First[head]
.end(); it++) {
                tmp.lookahead = *it;
                //判断当前项目是否存在于当前项目族中
                enterprocess(curNum, tmp);
            }
        }
        else {
            tmp.lookahead = head;
            //判断当前项目是否存在于当前项目族中
            enterprocess(curNum, tmp);
        }
    }
    else { // 只有lookhead
        head = lastItem.lookahead;
        if (isNonTerminal(head)) //如果是非终结符
        {
            for (auto it = First[head].begin(); it != First[head]
.end(); it++) {
                tmp.lookahead = *it;
                //判断当前项目是否存在于当前项目族中
                enterprocess(curNum, tmp);
            }
```

```
        }
        else {
            tmp.lookahead = head;
            //判断当前项目是否存在于当前项目族中
            enterprocess(curNum, tmp);
        }
    }

    return 1;
}

// 从另一闭包转移到当前闭包
int outclosure(int curNum, LR1Item curItem) {
    Exist[curNum] = 1; //当前族置存在
    LR1Item tmp = curItem;
    //判断当前项目是否存在于当前项目族中
    enterprocess(curNum, tmp);
    return 1;
}

// 算闭包的中间过程
void enterprocess(int curNum, LR1Item tmp) {
    // 判断当前项目是否在当前项目规范集中
    if (!isItemcurExistcanonical(curNum, tmp)) {
        canonicalCollection[curNum].push_back(tmp);//插入当前
        if (tmp.dotIndex < tmp.production.right.size()) {  //dot
不在末尾
            if (isNonTerminal(tmp.production.right.at(tmp.dotInde
x))) { //为非终结符，继续拓展计算，且不等于左式，防止左递归
                string nextsymbol = tmp.production.right.at(tmp.d
otIndex); //下一个左式
                for (auto nextp = LR1ItemSet.begin(); nextp != LR
1ItemSet.end(); nextp++) { // 在项目集中检索对应左式的表达式
                    if (nextp->production.left == nextsymbol && n
extp->dotIndex == 0) { //s->.AB
                        inclosure(curNum, *nextp, tmp);
                    }
                }
            }
        }
    }
    return;
}
```

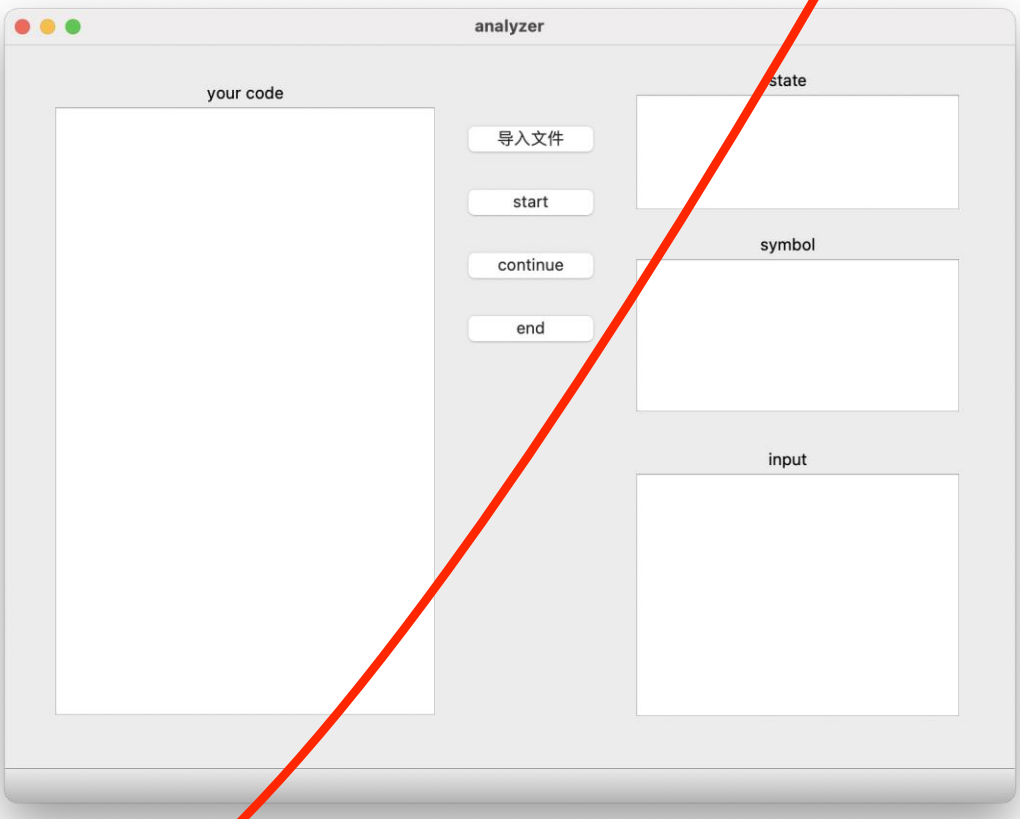

装 订 线

```
// 计算Go关系
void calGo() {
    while (1) {
        bool flag = true;
        for (int i = 0; i < canonicalCollection.size(); i++) {
            for (auto go_symbol = G.begin(); go_symbol != G.end()
; go_symbol++) {
                // 遍历符号表
                bool have = false;
                int nextNumber = nextZoneNumber();
                for (auto it = canonicalCollection[i].begin(); it
 != canonicalCollection[i].end(); it++) {
                    // 遍历当前族所有表达式
                    if (it->dotIndex < it->production.right.size(
) && it->production.right.at(it->dotIndex) == *go_symbol) {
                        have = true;
                        LR1Item nextp = *it;
                        nextp.dotIndex = it->dotIndex + 1;
                        // 进入闭包计算
                        outclosure(nextNumber, nextp);
                    }
                }
                if (have) {
                    if (checkBelongto(canonicalCollection[nextNum
ber]) != -1) {
                        // 转移关系进栈
                        Go tmp;
                        tmp.pre = i;
                        tmp.next = checkBelongto(canonicalCollect
ion[nextNumber]);
                        tmp.exist = 1;
                        tmp.changeStr = *go_symbol;
                        if (!findVec(tmp)) { go.push_back(tmp); }
                        // 若属于先前，则删除当前项目集
                        Exist[nextNumber] = 0;
                        canonicalCollection[nextNumber].clear();
                    }
                    else {
                        flag = false;
                        // 转移关系进栈
                        Go tmp;
                        tmp.pre = i;
                        tmp.next = nextNumber;
```


装 订 线

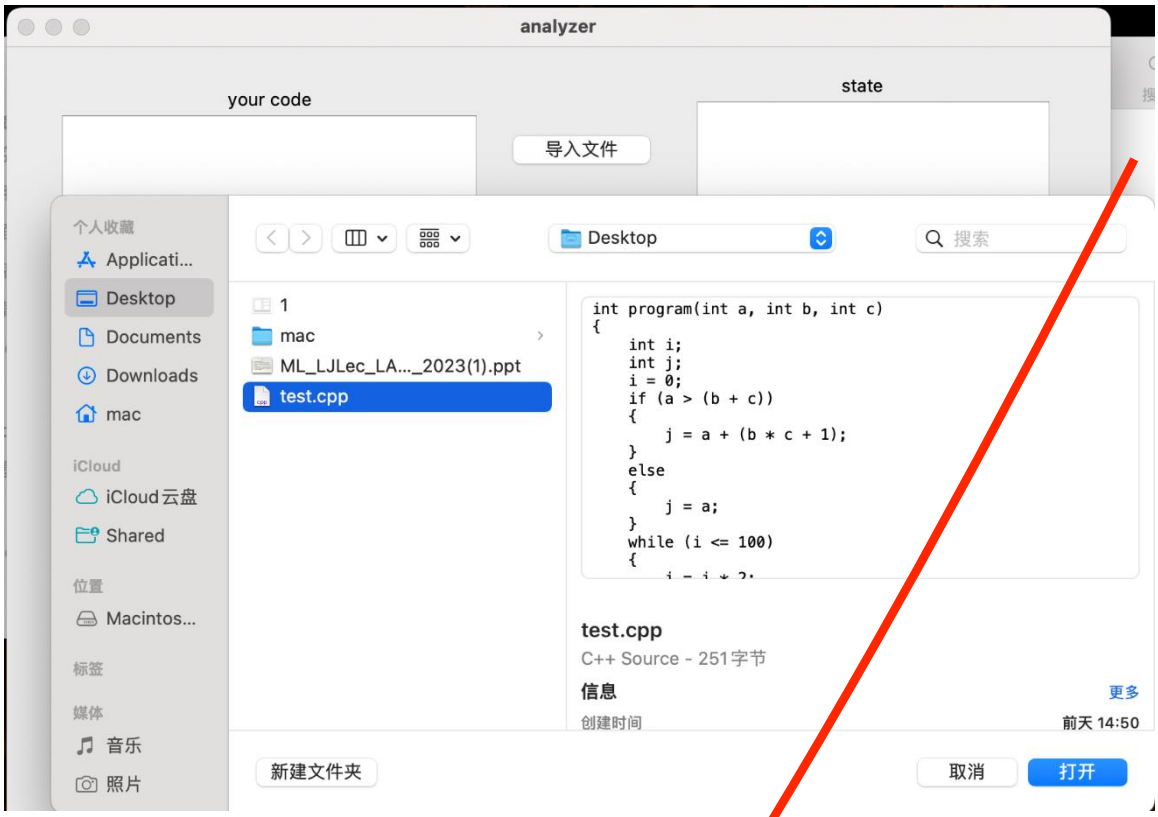
```
                            tmp.exist = 1;
                            tmp.changeStr = *go_symbol;
                            if (!findVec(tmp)) { go.push_back(tmp); }
                        }
                    }
                }
            }
            // 项目集规范族不再变化，计算结束
            if (flag) { break; }
        }

        return;
}
```

四、 界面设计与实验结果

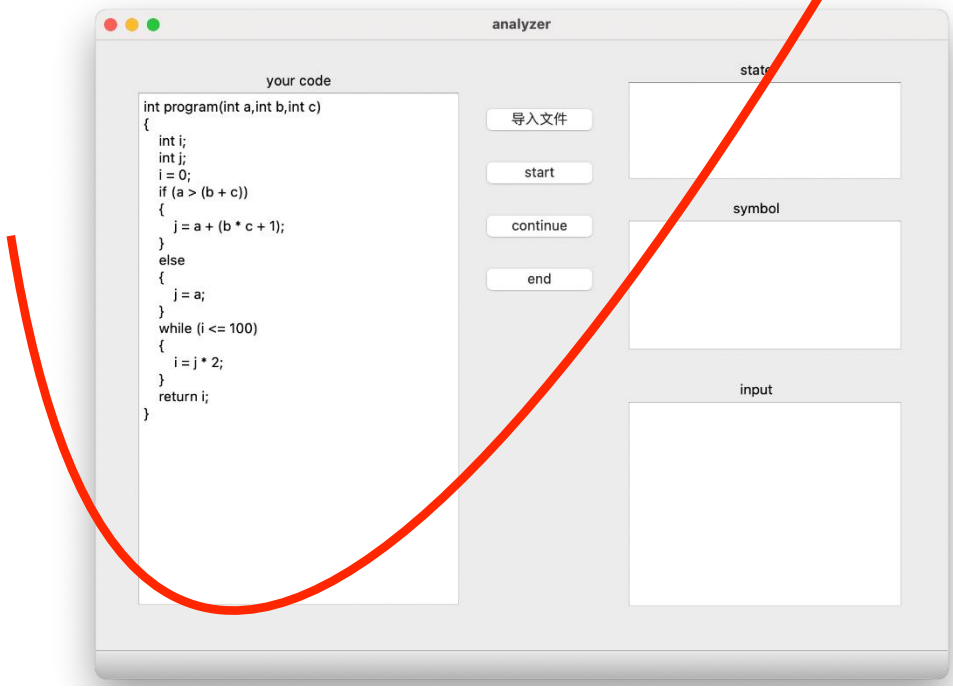


装 订 线

（1） 用户从主界面选择导入文件

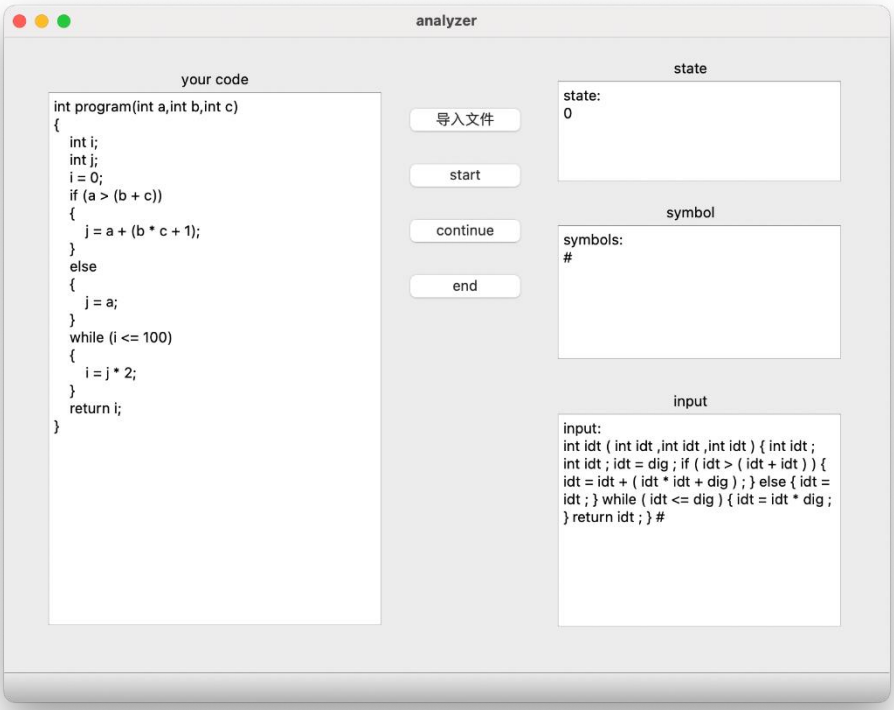


（2） 导入文件后，your code 界面自动显示文件内容

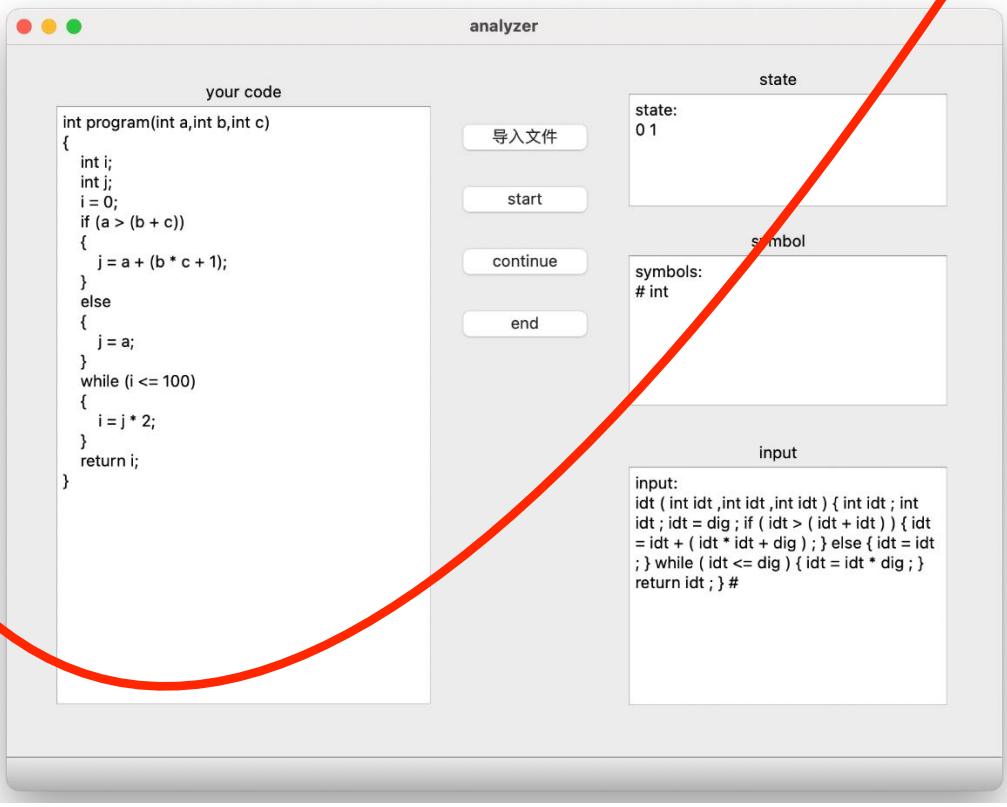


装 订 线

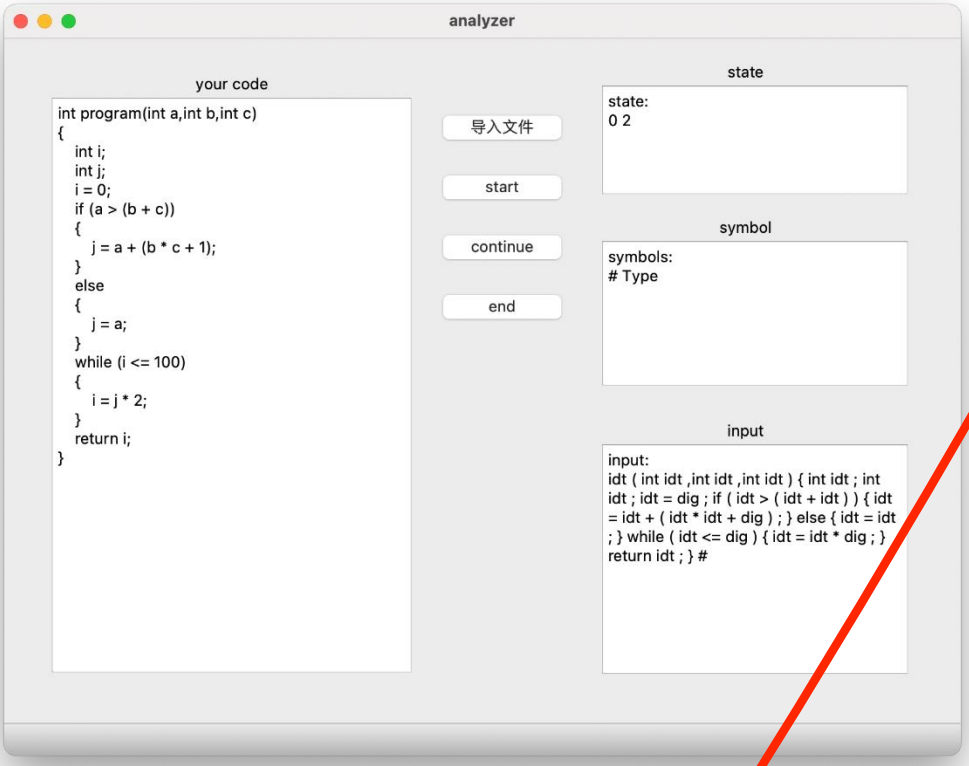
（3） 点击 start 按钮，进行词法与语法分析。state 窗口显示状态栈，symbol 窗口显示符号栈，input 窗口显示词法分析之后的输入串。

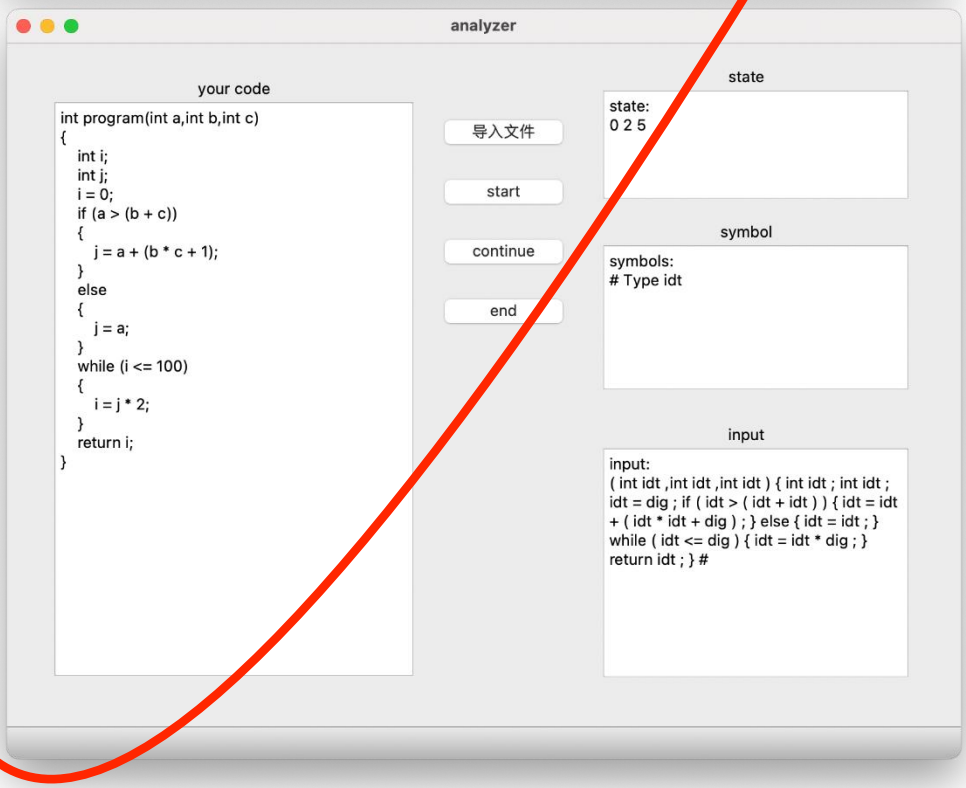


（4） 点击 continue 按钮，一步一步执行。窗口显示现在的状态。



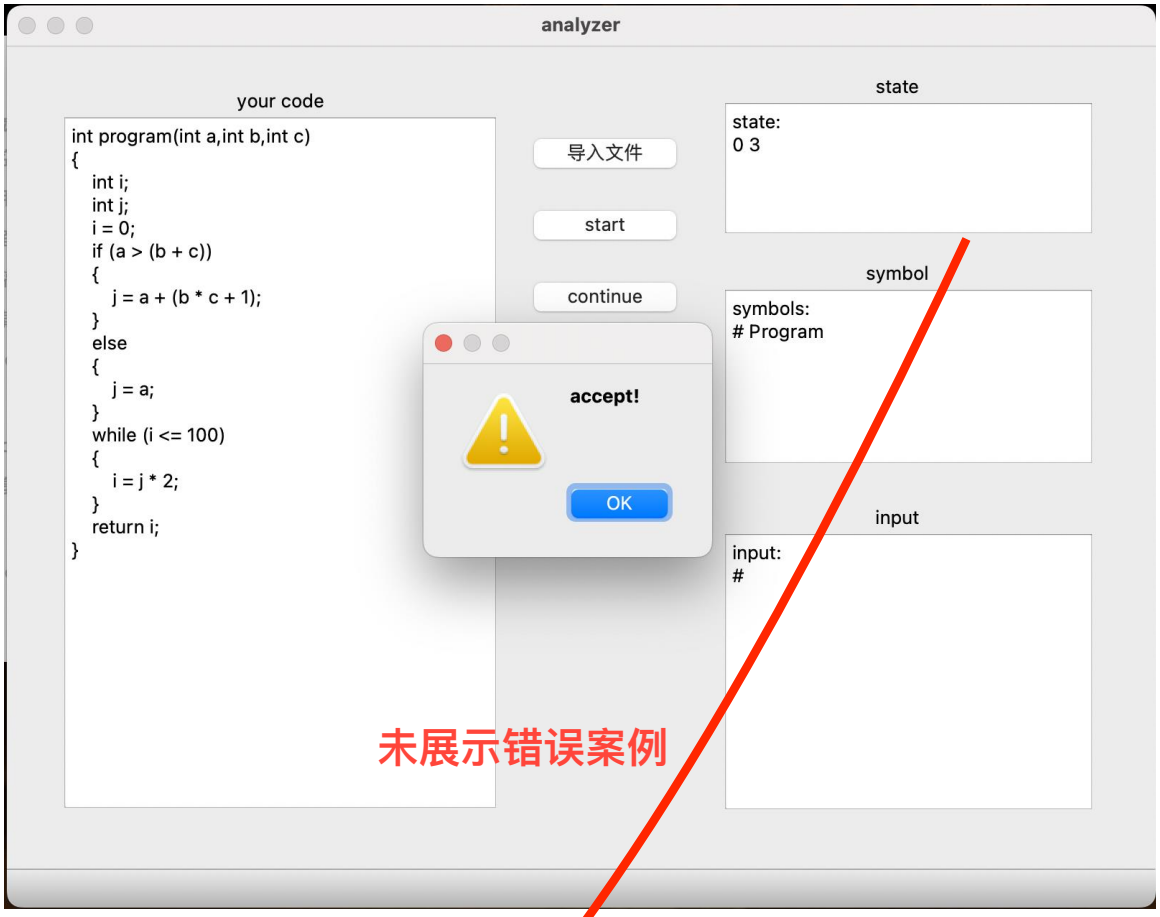
装 订 线


analyzer
your code
int program(int a,int b,int c)
{
int i;
int j;
i = 0;
if (a > (b + c))
{
j = a + (b * c + 1);
}
else
{
j = a;
}
while (i <= 100)
{
i = j * 2;
}
return i;
}
导入文件
start
continue
end
state
state:
0 2
symbol
symbols:
Type
input
input:
idt (int idt ,int idt ,int idt) { int idt ; int idt ; idt = dig ; if (idt > (idt + idt)) { idt = idt + (idt * idt + dig) ; } else { idt = idt ; } while (idt <= dig) { idt = idt * dig ; } return idt ; } #


analyzer
your code
int program(int a,int b,int c)
{
int i;
int j;
i = 0;
if (a > (b + c))
{
j = a + (b * c + 1);
}
else
{
j = a;
}
while (i <= 100)
{
i = j * 2;
}
return i;
}
导入文件
start
continue
end
state
state:
0 2 5
symbol
symbols:
Type idt
input
input:
(int idt ,int idt ,int idt) { int idt ; int idt ; idt = dig ; if (idt > (idt + idt)) { idt = idt + (idt * idt + dig) ; } else { idt = idt ; } while (idt <= dig) { idt = idt * dig ; } return idt ; } #

装 订 线

（5） 点击 end 按钮，直接显示最后的分析结果。如果语法正确则显示 accept。如果遇到错误则有 error 弹窗。



装 订 线

五、 小组成员分工

2153603 王政尧：语法分析器设计与实现、报告撰写。
2153683 郭嘉： UI 界面设计与实现、报告撰写。
2153603 张博文：词法、语法分析器设计与实现、报告撰写。

六、 参考资料

[1] 陈火旺 .程序设计语言编译原理：国防工业出版社，2020
[2] 网络资源. https://zhuanlan.zhihu.com/p/666554747
[3] 网络资源. https://zhuanlan.zhihu.com/p/666554747

装 订 线