

python实现中间代码生成器大作业实验报告

前言：本次中间代码生成器的编写和实现建立在语法分析大作业代码的前提下。本次报告选择简化语法分析的大部分内容，仅保留关键部分或者概述。

小组成员及贡献：

2151130 童海博：整体问题分析+框架代码和整体功能代码主要编写实现+测试调试+部分报告撰写（在文法分析的代码部分出力最大）

2152214 赵克祥：具体功能分析+具体功能代码主要编写实现+测试调试+部分报告撰写（在中间代码生成的代码部分出力最大）

2151140 王谦：程序代码分析+调试测试及结果呈现+部分代码修改调整+报告主要撰写（两次实验均承担相似任务）

虽然列举如此，但是实际上合作比较紧密，各个部分彼此都有互相参与，所以应该还是算共同完成了本次实验的问题分析、代码编写、功能分析与实现、调试测试、报告撰写等任务。

目录

python实现中间代码生成器大作业实验报告

目录

一、需求分析

程序任务输入及其范围

输入的简单描述

输出方式

输出内容

程序功能

测试数据

二、概要设计

任务分解

主要文件结构

数据类型定义

类之间的关系：

主程序流程与模块间的调用关系

三、详细设计

(1) `SpecialProductions`

(2) `AbstractSyntaxTree.py`

`ASTnode` 类

`__init__` 方法：

`__str__` 方法：

`AST` 类

`__init__` 方法：

`makeNodeMove` 方法：

`makeNodeReduce` 方法：

`dfsPrint` 方法：

(3) `IRGenerator.py`

`Quaternary` 类

`IRGenerator` 类

`Generate` 方法:

`dfsASTree` 方法:

`print` 方法:

对 `dfsASTree` 部分代码进行详细解释:

四、调试分析

测试数据一 (单个正确文件输入, if)

输入

输出

测试数据二 (单个正确文件, while)

输入

输出

测试数据三 (单个正确文件, for)

输入

输出

测试数据四 (单个正确文件, 综合)

输入

输出

测试数据五 (单个错误文件)

输入

输出

调试问题思考与解决

文法构造

中间代码生成四元式的处理

五、总结与收获

收获

对课程的认识与感受

六、参考文献

一、需求分析

程序任务输入及其范围

输入的简单描述

输入为c语言测试文件 `test.c` 和LR(1)文法文件 `grammar.txt`

其中 `test.c` 是相对简单的c语言代码文件, 相对灵活, 可以自行更改测试代码;

举例 `test.c`:

```
int main(){
//    a = a + 1;
    int a;
    int b;
    a = b + 1;
    for(a = 1; a < 2; a = a + 1)
    {
        a = 1;
        if(a < b){
            a = a + 1;
        }else{
            a = b * (a + 1);
        }
    }
}
```

```
}  
}
```

grammar.txt 为我们小组经过不断地分析和测试最终构造得到的能够满足测试c语言的LR(1)文法，相对固定。

最终完成的 grammar.txt 展示：

```
program -> declaration_list  
  
declaration_list -> declaration declaration_list | declaration  
  
declaration -> var_declaration | fun_declaration  
  
var_declaration -> type_specifier ID opt_init ; | type_specifier ID [ NUM ]  
opt_init ; | type_specifier ID [ ] opt_init ; | type_specifier ID [ NUM ] ; |  
type_specifier ID ;  
  
opt_init -> = expression  
  
type_specifier -> const simple_type | simple_type  
  
fun_declaration -> type_specifier ID ( params ) compound_stmt | type_specifier ID  
( ) compound_stmt  
  
params -> param_list | void  
  
param_list -> param | param_list , param  
  
param -> type_specifier ID | type_specifier ID [ ] | type_specifier ID [ NUM ]  
  
compound_stmt -> { block_items } | { }  
  
block_items -> statement_list  
  
statement_list -> statement statement_list | statement  
  
statement -> var_declaration | expression_stmt | compound_stmt | selection_stmt |  
iteration_stmt | return_stmt  
  
selection_stmt -> if ( expression ) statement | if ( expression ) statement else  
statement | else if ( expression ) statement  
  
iteration_stmt -> while ( expression ) statement | for ( expression_stmt  
expression_stmt expression ) statement | for ( expression_stmt expression_stmt  
expression ) ;  
  
return_stmt -> return ; | return expression ;  
  
expression_stmt -> expression ;  
  
expression -> simple_expression | var = expression | type_specifier ID opt_init
```

```

var -> ID | ID [ expression ]

simple_expression -> additive_expression relop additive_expression |
additive_expression | sglop additive_expression

additive_expression -> term | additive_expression addop term

term -> factor | term mulop factor

factor -> ( expression ) | var | call | NUM | STRING | CH

call -> ID ( args ) | ID ( )

args -> arg_list

arg_list -> expression | arg_list , expression

simple_type -> void | float | double | int | long | char

addop -> + | -

mulop -> * | /

relop -> < | > | == | >= | <=

sglop -> - | +

```

其中产生符号由右箭头 `->` 表示，不同的产生式由 `|` 分隔开，这种呈现格式和课程是保持一致的。算符之间都用空格分隔开，以作区分。

输出方式

既可以直接在调试窗口上直接打印出来，又可以将结果以文件的形式输出保存；语法树也可以以图片方式生成，不过图片形式效果不佳。

输出内容

在用文法构造的同时也将 `GOTO` 等内容保存并输出到文件 `table_new.cfg` 中。

`playground.py` 作为整体调用的程序，会逐步输出文法分析过程，然后逐节点输出抽象语法树，最后输出分析后的中间代码四元式。

详细输出内容说明请参考第四部分调试分析。

程序功能

语法分析部分（仅列举，不在本报告中做详细解释）：

1. 能够读取单个或者批量读取测试文件进行处理
2. 能够根据c语言测试文件输出词法分析的结果，并能进一步利用词法分析结果进行语法分析
3. 能够得到ACTION、GOTO表
4. 能够得到各个项目集的闭包CLOSURE
5. 能够得到FIRST集
6. 能够逐步输出移进规约的识别过程
7. 能够最终判断测试文件是否符合文法能被正确识别
8. 能够得到语法树

中间代码生成部分（本次报告的重点）：

1. 能够得到抽象语法树
2. 能够生成中间代码并输出四元式

测试数据

大致测试数据形式已经在上面给出，更多测试数据和测试过程将在第四模块调试分析进一步展示说明。

二、概要设计

任务分解

- 抽象语法树的获取和表示
- 中间代码的生成和四元式输出

主要文件结构

- 文件夹 `IRGenerator`（中间代码生成部分，本次报告的重点）
 - `AbstractSyntaxTree.py`
 - `IRGenerator.py`
 - `SpecialProductions.py`
- 文件夹 `LexicalAnalysis`（词法分析部分）
 - `Lexer.py`（词法分析）
 - `SpecialTokens.py`（词法分析中的特殊关键词）
- 文件夹 `Parser`（LR(1)文法分析部分）
 - `Grammer.py`（读取语法文件并储存）
 - `LR1Parser.py`（LR(1)文法分析）
 - `grammar.txt`（文法文件）
- 文件夹 `prog`（输入的测试数据文件）
 - 文件夹 `accepted`（能成功识别的正确测试用例）
 - 文件夹 `error`（无法识别的错误测试用例）
- 文件夹 `result`（测试输出结果）

- `accepted.txt` (能成功识别的正确测试用例的输出结果)
- `error.txt` (无法识别的错误测试用例的输出结果)
- `FileReader.py` (读取测试文件)
- `Compiler.py` (对特定文法和特定测试文件整合调用代码实现整体功能)
- `playground.py` (选择调用的语法文件和测试文件)

数据类型定义

此处仅给出在文法分析部分之后新增的部分

1. `SpecialProductions` 类:

- 用于表示特殊产生式的集合。
- 属性:
 - `productions`: 一个字典, 表示特殊产生式。键是产生式的标识符, 值是产生式的字符串形式。

2. `ASTnode` 类:

- 用于表示抽象语法树的节点。
- 属性:
 - `index`: 节点的索引。
 - `node`: 节点的类型或值。
 - `child`: 子节点的索引列表。
 - `isLeaf`: 表示节点是否为叶子节点。
 - `leafVal`: 叶子节点的值。
 - `production`: 节点对应的产生式。
 - `quaterAddress`: 节点对应的四元式地址。
- 方法:
 - `__str__(self)`: 返回节点的字符串表示形式。

3. `AST` 类:

- 用于表示抽象语法树。
- 属性:
 - `tree`: 一个列表, 存储抽象语法树的节点。
 - `root`: 树的根节点的索引。
 - `declared`: 一个字典, 记录已声明的标识符。
- 方法:
 - `makeNodeMove(self, node)`: 创建一个移动节点, 返回节点的索引。
 - `makeNodeReduce(self, production, index_stack)`: 创建一个规约节点, 返回节点的索引。
 - `dfsPrint(self, index)`: 深度优先搜索打印抽象语法树。

4. `IRGenerator` 类:

- 用于生成中间代码的类。
- 属性:
 - `IRCode`: 一个列表, 存储生成的中间代码 (Quaternary)。
 - `tmpVar`: 一个字典, 记录临时变量及其对应的中间代码地址。

- 方法：
 - `Generate(self, ASTree: AST)`: 中间代码生成的入口方法。
 - `dfsASTree(self, ASTree: AST, index)`: 深度优先搜索抽象语法树，生成中间代码。
 - `print(self)`: 打印生成的中间代码。

5. Quaternary 类：

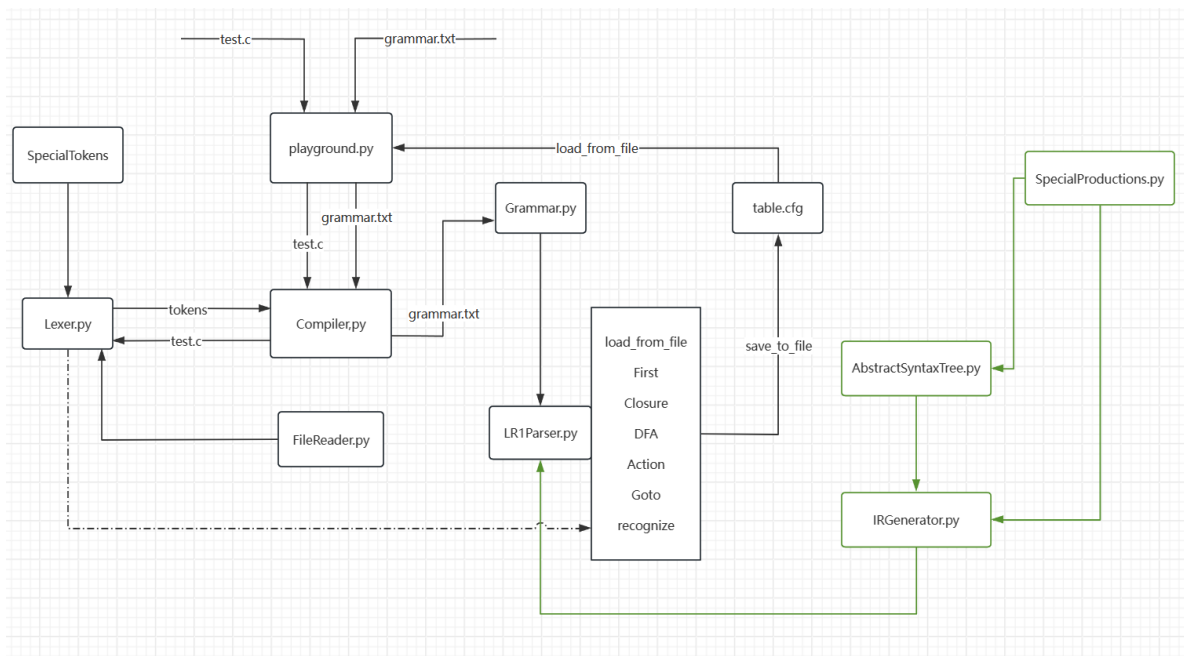
- 用于表示四元式（中间代码）的类。
- 属性：
 - `op`: 操作符。
 - `arg1`: 第一个操作数。
 - `arg2`: 第二个操作数。
 - `result`: 结果。

类之间的关系：

- `AST` 类包含了 `ASTnode` 类的实例，表示抽象语法树。
- `IRGenerator` 类使用了 `AST` 类的实例，生成中间代码。
- `IRGenerator` 类中的 `Quaternary` 类表示中间代码的四元式形式。

主程序流程与模块间的调用关系

简单的流程和调用关系如图所示，右侧加绿色框线部分为本次报告侧重点，详细请参考源代码和第三模块的详细设计



三、详细设计

(1) SpecialProductions

```
SpecialProductions = {
  "addop": "additive_expression -> additive_expression addop term",
  "mulop": "term -> term mulop factor",
  "assignop": "expression -> var = expression",
  "relop": "simple_expression -> additive_expression relop
additive_expression",
  "sglop": "simple_expression -> sglop additive_expression",
  "factor": "factor -> ( expression )",
  "if": "selection_stmt -> if ( expression ) statement",
  "if-else": "selection_stmt -> if ( expression ) statement else statement",
  "while": "iteration_stmt -> while ( expression ) statement",
  "for": "iteration_stmt -> for ( expression_stmt expression_stmt expression )
statement",
  "for-empty": "iteration_stmt -> for ( expression_stmt expression_stmt
expression ) ;",
  "declare-init": "var_declaration -> type_specifier ID opt_init ;",
  "declare-arrinit": "var_declaration -> type_specifier ID [ NUM ] opt_init
;",
  "declare-arr": "var_declaration -> type_specifier ID [ NUM ] ;",
  "declare-var": "var_declaration -> type_specifier ID ;",
  "fac-var": "factor -> var"
}
```

这段代码定义了一个上下文无关文法的一组特殊产生式规则。每个键在 `SpecialProductions` 字典中对应文法的一个非终结符号，而与之关联的值则提供了该非终结符号的产生式规则。

1. `"addop": "additive_expression -> additive_expression addop term"`
 - 规定了如何通过将另一个 `additive_expression` 与一个 `addop` 和一个 `term` 结合起来产生一个新的 `additive_expression`。
2. `"mulop": "term -> term mulop factor"`
 - 规定了如何通过将另一个 `term` 与一个 `mulop` 和一个 `factor` 结合起来产生一个新的 `term`。
3. `"assignop": "expression -> var = expression"`
 - 规定了如何通过将另一个 `expression` 赋值给变量 `var` 来产生一个新的 `expression`。
4. `"relop": "simple_expression -> additive_expression relop additive_expression"`
 - 规定了如何通过使用关系运算符 `relop` 比较两个 `additive_expression` 来产生一个新的 `simple_expression`。
5. `"sglop": "simple_expression -> sglop additive_expression"`
 - 规定了如何通过将一个符号运算符 `sglop` 应用于一个 `additive_expression` 来产生一个新的 `simple_expression`。
6. `"factor": "factor -> (expression)"`
 - 规定了如何通过将一个 `expression` 放在括号中来产生一个新的 `factor`。
7. `"if": "selection_stmt -> if (expression) statement"`
 - 规定了如何通过将一个 `expression` 与 `if` 关键字和一个 `statement` 结合起来产生一个新的 `selection_stmt` (if 语句)。
8. `"if-else": "selection_stmt -> if (expression) statement else statement"`

- 规定了如何通过添加 `else` 分支来产生具有 `else` 部分的 `if` 语句。
- 9. `"while": "iteration_stmt -> while (expression) statement"`
 - 规定了如何通过使用 `"while"` 关键字、一个 `expression` 和一个 `statement` 产生一个新的迭代语句 (`iteration_stmt`)。
- 10. `"for": "iteration_stmt -> for (expression_stmt expression_stmt expression) statement"`
 - 规定了如何通过使用 `"for"` 关键字、三个 `expression_stmt` 部分和一个 `statement` 产生一个新的迭代语句 (`iteration_stmt`)。
- 11. `"for-empty": "iteration_stmt -> for (expression_stmt expression_stmt expression) ;"`
 - 规定了一个变体的 `"for"` 循环，其中循环体为空（用分号表示）。
- 12. `"declare-init": "var_declaration -> type_specifier ID opt_init ;"`
 - 规定了如何通过使用类型说明符 `type_specifier`、标识符 `ID` 和可选的初始化部分 `opt_init` 产生一个新的变量声明 (`var_declaration`)。
- 13. `"declare-arrinit": "var_declaration -> type_specifier ID [NUM] opt_init ;"`
 - 规定了如何通过使用类型说明符 `type_specifier`、标识符 `ID`、带有数组长度的 `[NUM]` 部分和可选的初始化部分 `opt_init` 产生一个新的数组变量声明。
- 14. `"declare-arr": "var_declaration -> type_specifier ID [NUM] ;"`
 - 规定了如何通过使用类型说明符 `type_specifier`、标识符 `ID` 和带有数组长度的 `[NUM]` 部分产生一个新的数组变量声明，没有初始化。
- 15. `"declare-var": "var_declaration -> type_specifier ID ;"`
 - 规定了如何通过使用类型说明符 `type_specifier` 和标识符 `ID` 产生一个新的变量声明，没有数组或初始化。
- 16. `"fac-var": "factor -> var"`
 - 规定了如何通过使用变量 `var` 产生一个新的 `factor`。

(2) AbstractSyntaxTree.py

这段代码定义了一个抽象语法树 (Abstract Syntax Tree, AST) 的数据结构和相关操作。让我们逐步解释：

ASTnode 类

- `__init__`: 初始化 AST 节点对象，设置节点的索引 `index`、节点类型 `node`，子节点列表 `child`，是否为叶子节点 `isLeaf`，叶子节点的值 `leafVal`，产生式规则 `production` 和四元式地址 `quaterAddress`。
- `__str__`: 返回节点的字符串表示，包括节点的索引、类型、子节点列表、是否为叶子节点、叶子节点的值、产生式规则和四元式地址。

`ASTnode` 类用于表示抽象语法树中的每个节点。以下是对该类的详细解释：

`__init__` 方法：

```
def __init__(self, index, node):
    self.index = index # 节点的索引
    self.node = node # 节点类型
    self.child = [] # 子节点列表
    self.isLeaf = False # 是否为叶子节点
    self.leafVal = None # 叶子节点的值
    self.production = None # 产生式规则
    self.aterAddress = -1 # 四元式地址
```

这个方法用于初始化 `ASTnode` 对象的属性。每个节点包含了其在抽象语法树中的索引、节点类型、子节点列表、是否为叶子节点、叶子节点的值、产生式规则和四元式地址等信息。

`__str__` 方法:

```
def __str__(self):
    return f"index:{self.index}, node:{self.node}, child:{self.child}, isLeaf: {self.isLeaf}, leafVal:{self.leafVal}, " \
        f"pro:{self.production}, address:{self.aterAddress}"
```

这个方法用于返回节点的字符串表示，以便于调试和输出信息

AST 类

- `__init__`: 初始化 AST 对象，包括 AST 树的列表 `tree`、根节点的索引 `root` 和已声明变量的字典 `declared`。
- `makeNodeMove`: 创建一个移位 (shift) 节点，将该节点添加到 AST 中，并返回该节点的索引。
- `makeNodeReduce`: 创建一个规约 (reduce) 节点，将该节点添加到 AST 中，并返回该节点的索引。根据产生式的左侧和右侧构建节点，并根据特殊产生式规则进行相应的处理，例如变量声明和赋值操作。
- `dfsPrint`: 通过深度优先搜索遍历 AST，并打印每个节点的信息。

这个模块主要用于构建和操作抽象语法树，其中 `makeNodeMove` 用于处理移位操作，`makeNodeReduce` 用于处理规约操作，而 `dfsPrint` 用于打印整个抽象语法树。

`__init__` 方法:

```
def __init__(self):
    self.tree = [] # AST 树的列表
    self.root = -1 # 根节点的索引
    self.declared = {} # 已声明变量的字典
```

这个方法用于初始化 `AST` 对象的属性。包括 AST 树的列表、根节点的索引和已声明变量的字典。

`makeNodeMove` 方法:

```
def makeNodeMove(self, node):
    newNode = ASTnode(len(self.tree), node)
    newNode.isLeaf = True
    newNode.leafVal = node
    self.tree.append(newNode)
    return newNode.index
```

这个方法用于创建一个移进 (shift) 节点，将该节点添加到 AST 中，并返回该节点的索引。

`makeNodeReduce` 方法:

```
def makeNodeReduce(self, production, index_stack):
    faNode = ASTnode(len(self.tree), production["lhs"])
    faNode.production = production
    faIndex = len(self.tree)

    productionStr = production["lhs"] + " ->"
    for rhs in production["rhs"]:
        productionStr += " " + rhs

    # 处理不同的产生式规则
    if productionStr in [SpecialProductions["declare-init"],
        SpecialProductions["declare-arrinit"],
        SpecialProductions["declare-arr"],
        SpecialProductions["declare-var"]]:
        # 处理变量声明的情况
        # ...
    elif productionStr in [SpecialProductions["assignop"],
        SpecialProductions["fac-var"]]:
        # 处理赋值和变量引用的情况
        # ...
    else:
        # 处理其他情况
        # ...

    return faIndex
```

这个方法用于创建一个规约 (reduce) 节点, 将该节点添加到 AST 中, 并返回该节点的索引。根据产生式的左侧和右侧构建节点, 并根据特殊产生式规则进行相应的处理, 例如变量声明和赋值操作。

`dfsPrint` 方法:

```
def dfsPrint(self, index):
    print(self.tree[index])
    for son in self.tree[index].child:
        self.dfsPrint(son)
```

这个方法用于通过深度优先搜索遍历 AST, 并打印每个节点的信息。

(3) IRGenerator.py

`Quaternary` 类

```
class Quaternary:
    def __init__(self):
        self.op = None # 操作符
        self.arg1 = None # 第一个操作数
        self.arg2 = None # 第二个操作数
        self.result = None # 结果
```

这个类用于表示四元式 (Quaternary)。每个四元式包含了操作符 (op)、第一个操作数 (arg1)、第二个操作数 (arg2) 和结果 (result)。

IRGenerator 类

```
class IRGenerator:
    def __init__(self):
        self.IRCode = []    # 存储生成的中间代码的列表
        self.tmpVar = {}    # 临时变量的字典

    def Generate(self, ASTree: AST):
        pass
```

这个类包含了中间代码生成器的主要逻辑。

Generate 方法:

```
def Generate(self, ASTree: AST):
    pass
```

这个方法是中间代码生成的入口点，目前方法体为空。在实际情况下，它将遍历抽象语法树，调用 `dfsASTree` 方法来生成中间代码。

dfsASTree 方法:

```
def dfsASTree(self, ASTree: AST, index):
    # ...
```

这个方法使用深度优先搜索 (DFS) 遍历抽象语法树，并生成相应的中间代码。代码中包括了对不同的产生式规则的处理，例如加法操作、乘法操作、关系运算、赋值操作、条件语句等。每个四元式将被添加到 `IRCode` 列表中，并相应地更新抽象语法树节点的信息。部分产生式规则包括了对临时变量的管理，以及对条件语句的处理。

print 方法:

```
def print(self):
    for i in range(len(self.IRCode)):
        print(f"{i}: {self.IRCode[i]}")
```

这个方法用于打印生成的中间代码，以便于调试和查看生成的四元式。

对 dfsASTree 部分代码进行详细解释:

`dfsASTree` 方法是中间代码生成的核心部分，通过深度优先搜索 (DFS) 遍历抽象语法树 (AST)，并在遍历过程中生成中间代码。以

```
def dfsASTree(self, ASTree: AST, index):
    quater = Quaternary() # 创建一个 Quaternary 对象，用于存储生成的四元式
    faNode = ASTree.tree[index] # 获取当前 AST 节点
```

```

leafVals = [] # 子节点的叶子值列表
quaterAddress = [] # 子节点的四元式地址列表

# 遍历子节点
for son in faNode.child:
    # 递归调用 dfsASTree 处理子节点
    self.dfsASTree(ASTree, son)

    # 处理单一子节点情况
    if len(faNode.child) == 1:
        faNode.leafVal = ASTree.tree[son].leafVal
        faNode.quaterAddress = ASTree.tree[son].quaterAddress
        ASTree.tree[faNode.index] = faNode
        return

    leafVals.append(ASTree.tree[son].leafVal)
    quaterAddress.append(ASTree.tree[son].quaterAddress)

# 获取产生式规则字符串
if faNode.production:
    productionStr = faNode.production["lhs"] + " ->"
    for rhs in faNode.production["rhs"]:
        productionStr += " " + rhs

    # 处理不同的产生式规则
    if productionStr in [SpecialProductions["addop"],
SpecialProductions["mulop"], SpecialProductions["relop"]]:
        # 处理加法、乘法、关系运算
        quater.op = leafVals[1]
        quater.arg1 = leafVals[0]
        quater.arg2 = leafVals[2]
        newTmp = len(self.tmpVar)
        quater.result = "$" + str(newTmp)
        self.tmpVar[quater.result] = len(self.IRCode)
        self.IRCode.append(quater)
        faNode.leafVal = quater.result
        faNode.quaterAddress = len(self.IRCode) - 1
        ASTree.tree[faNode.index] = faNode
    elif productionStr == SpecialProductions["assignop"]:
        # 处理赋值操作
        quater.op = leafVals[1]
        quater.arg1 = leafVals[0]
        quater.arg2 = leafVals[2]
        faNode.leafVal = quater.arg1
        self.IRCode.append(quater)
        faNode.quaterAddress = len(self.IRCode) - 1
        ASTree.tree[faNode.index] = faNode
    elif productionStr == SpecialProductions["sglop"]:
        # 处理单目运算操作
        quater.op = leafVals[0]
        quater.arg1 = leafVals[1]
        newTmp = len(self.tmpVar)
        quater.result = "$" + str(newTmp)
        self.tmpVar[quater.result] = len(self.IRCode)

```

```

self.IRCode.append(quarter)
faNode.leafVal = quarter.result
faNode.quaterAddress = len(self.IRCode) - 1
ASTree.tree[faNode.index] = faNode
elif productionStr == SpecialProductions["factor"]:
    # 处理因子（子表达式）情况
    faNode.leafVal = ASTree.tree[faNode.child[1]].leafVal
    faNode.quaterAddress = len(self.IRCode) - 1
    ASTree.tree[faNode.index] = faNode
elif productionStr == SpecialProductions["if"]:
    # 处理 if 语句
    quarter.op = "jnz"
    quarter.arg1 = leafVals[2]
    quarter.result = quarterAddress[4] + 2
    quarter1 = Quaternary()
    quarter1.op = "j"
    quarter1.result = len(self.IRCode) + 2
    self.IRCode = self.IRCode[0:quarterAddress[4]] + [quarter, quarter1] +
self.IRCode[quarterAddress[4]:]
    faNode.quaterAddress = quarterAddress[2]
    ASTree.tree[faNode.index] = faNode
    ASTree.tree[faNode.child[4]].quaterAddress += 2
    for i in range(quarterAddress[4] + 2, len(self.IRCode)):
        code = self.IRCode[i]
        if code.op == "j" or code.op == "jnz":
            self.IRCode[i].result += 2
    # 处理其他产生式规则，略...

# 更新节点的四元式地址信息
index_tmp = 1e9
for son in faNode.child:
    if ASTree.tree[son].quaterAddress >= 0:
        index_tmp = min(index_tmp, ASTree.tree[son].quaterAddress)
if index_tmp == 1e9:
    index_tmp = faNode.quaterAddress
if faNode.quaterAddress < 0:
    faNode.quaterAddress = index_tmp
else:
    faNode.quaterAddress = min(index_tmp, faNode.quaterAddress)
ASTree.tree[faNode.index] = faNode

```

这个方法根据抽象语法树中的不同产生式规则生成相应的中间代码。代码中包含了对各种语法结构的处理，包括算术运算、赋值语句、条件语句等。每次处理完一个产生式规则，就会在 `IRCode` 中添加相应的四元式。在遍历的过程中，会更新抽象语法树节点的信息，包括叶子值和四元式地址。

四、调试分析

写在之前：

在此前的文法分析部分，我们小组对C语言的语法分析相对来说已经是相当完善的了，除了指针没做以外，其他几乎都有实现，包括但不限于for循环、while循环、空大括号、函数等等。

但到了本次中间代码生成部分，受限于所学知识的内容范围、对知识的熟悉程度以及时间紧张程度等场内外诸多因素，实现效果达到了if语句、while循环、for循环等功能，已经满足了本课程当前要求，但并没有实现更加复杂的函数功能。

测试数据一（单个正确文件输入，if）

输入

本测试代码虽然包含多个函数，但是都为空且并未被主程序调用，实际上能够通过我们的程序测试。

```
// test for function

void solve() {
}

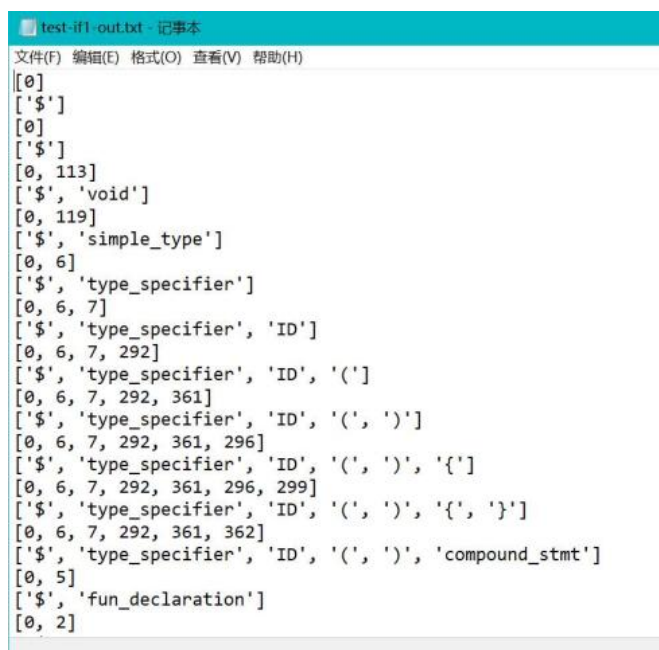
const char is_digit(char ch) {
}

void input(void) {
}

int main() {
    int a;
    if (a == 1) {
        a = 2;
    }
    else {
        a = 3;
    }
}
```

输出

首先输出的是文法分析过程



```
test-if1-out.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
[0]
['$']
[0]
['$']
[0, 113]
['$', 'void']
[0, 119]
['$', 'simple_type']
[0, 6]
['$', 'type_specifier']
[0, 6, 7]
['$', 'type_specifier', 'ID']
[0, 6, 7, 292]
['$', 'type_specifier', 'ID', '(']
[0, 6, 7, 292, 361]
['$', 'type_specifier', 'ID', '(', ')']
[0, 6, 7, 292, 361, 296]
['$', 'type_specifier', 'ID', '(', ')', '{']
[0, 6, 7, 292, 361, 296, 299]
['$', 'type_specifier', 'ID', '(', ')', '{', '}']
[0, 6, 7, 292, 361, 362]
['$', 'type_specifier', 'ID', '(', ')', 'compound_stmt']
[0, 5]
['$', 'fun_declaration']
[0, 2]
```

文法分析结束后开始输出抽象语法树，语法树是逐个节点输出的，节点具体内容在之前已经解释过，这里再重复一遍：


```
def __init__(self, index, node):
    self.index = index # 节点的索引
    self.node = node # 节点类型
    self.child = [] # 子节点列表
    self.isLeaf = False # 是否为叶子节点
    self.leafVal = None # 叶子节点的值
    self.production = None # 产生式规则
    self.quaterAddress = -1 # 四元式地址
```

```
test-if1-out.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
[0, 2, 2, 2, 6, 7, 292, 361, 296, 300]
['$', 'declaration', 'declaration', 'declaration', 'type_specifier', 'ID', '(', ')', '{'
[0, 2, 2, 2, 6, 7, 292, 361, 296, 297]
['$', 'declaration', 'declaration', 'declaration', 'type_specifier', 'ID', '(', ')', '{'
[0, 2, 2, 2, 6, 7, 292, 361, 296, 297, 298]
['$', 'declaration', 'declaration', 'declaration', 'type_specifier', 'ID', '(', ')', '{'
[0, 2, 2, 2, 6, 7, 292, 361, 362]
['$', 'declaration', 'declaration', 'declaration', 'type_specifier', 'ID', '(', ')', 'co
[0, 2, 2, 2, 5]
['$', 'declaration', 'declaration', 'declaration', 'fun_declaration']
[0, 2, 2, 2, 2]
['$', 'declaration', 'declaration', 'declaration', 'declaration']
[0, 2, 2, 2, 3]
['$', 'declaration', 'declaration', 'declaration', 'declaration_list']
[0, 2, 2, 3]
['$', 'declaration', 'declaration', 'declaration_list']
[0, 2, 3]
['$', 'declaration', 'declaration_list']
[0, 1]
['$', 'declaration_list']
index:124, node:declaration_list, child:[10, 123], isLeaf:False, leafVal:None, pro:{'lhs
'declaration_list'}}, address:0
index:10, node:declaration, child:[9], isLeaf:False, leafVal:None, pro:{'lhs': 'declarat
index:9, node:fun_declaration, child:[2, 3, 4, 5, 8], isLeaf:False, leafVal:None, pro:{'
'(', ')', 'compound_stmt'}}, address:-1
```

语法树按照节点逐个输出后，输出中间代码四元式，并在整个任务成功后输出True

```
test-if1-out.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
index:108, node:block_items, child:[107], isLeaf:False, leafVal:a, pro:{'lhs'
index:107, node:statement_list, child:[106], isLeaf:False, leafVal:a, pro:{'
index:106, node:statement, child:[105], isLeaf:False, leafVal:a, pro:{'lhs':
index:105, node:expression_stmt, child:[103, 104], isLeaf:False, leafVal:a,
index:103, node:expression, child:[95, 96, 102], isLeaf:False, leafVal:a, pr
index:95, node:var, child:[94], isLeaf:False, leafVal:a, pro:{'lhs': 'var',
index:94, node:a, child:[], isLeaf:True, leafVal:a, pro:None, address:-1
index:96, node:=", child:[], isLeaf:True, leafVal:=", pro:None, address:-1
index:102, node:expression, child:[101], isLeaf:False, leafVal:3, pro:{'lhs'
index:101, node:simple_expression, child:[100], isLeaf:False, leafVal:3, prc
address:-1
index:100, node:additive_expression, child:[99], isLeaf:False, leafVal:3, pr
index:99, node:term, child:[98], isLeaf:False, leafVal:3, pro:{'lhs': 'term'
index:98, node:factor, child:[97], isLeaf:False, leafVal:3, pro:{'lhs': 'fac
index:97, node:3, child:[], isLeaf:True, leafVal:3, pro:None, address:-1
index:104, node:;, child:[], isLeaf:True, leafVal:;, pro:None, address:-1
index:109, node:}, child:[], isLeaf:True, leafVal:}, pro:None, address:-1
index:117, node:}, child:[], isLeaf:True, leafVal:}, pro:None, address:-1
0: (==, a, 1, $0)
1: (jnz, $0, None, 3)
2: (j, None, None, 4)
3: (=: a, 2, None)
4: (=: a, 3, None)
True
```

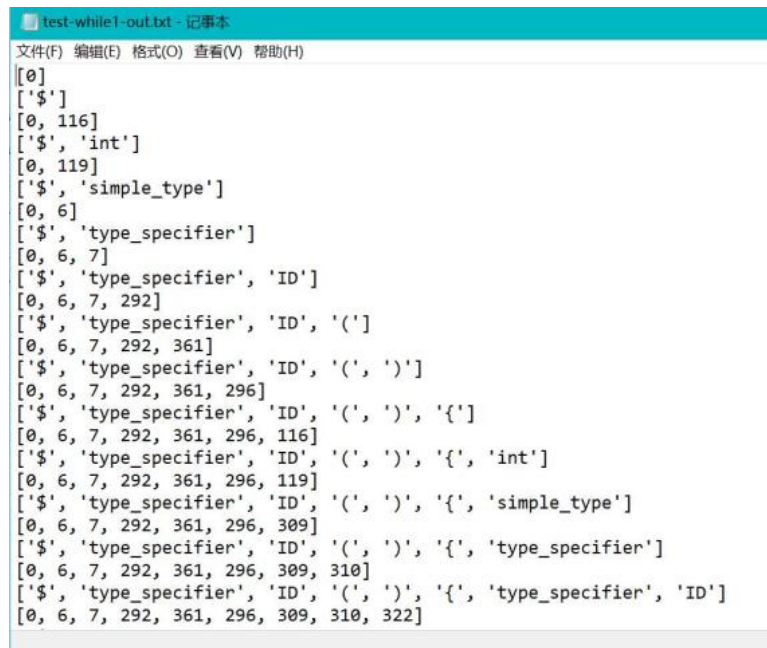

测试数据二（单个正确文件，while）

输入

```
int main() {
    int i;
    int n;
    while (i < n) {
        int j;
        j = 1;
        i = i + 1;
        for (j = 1; j < 5; j = j + 1) {
            i = i * n / n + i;
        }
    }
}
```

输出

首先输出的是文法分析过程



```
test-while1-out.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
[[0]
['$']
[0, 116]
['$', 'int']
[0, 119]
['$', 'simple_type']
[0, 6]
['$', 'type_specifier']
[0, 6, 7]
['$', 'type_specifier', 'ID']
[0, 6, 7, 292]
['$', 'type_specifier', 'ID', '(']
[0, 6, 7, 292, 361]
['$', 'type_specifier', 'ID', '(', ')']
[0, 6, 7, 292, 361, 296]
['$', 'type_specifier', 'ID', '(', ')', '{']
[0, 6, 7, 292, 361, 296, 116]
['$', 'type_specifier', 'ID', '(', ')', '{', 'int']
[0, 6, 7, 292, 361, 296, 119]
['$', 'type_specifier', 'ID', '(', ')', '{', 'simple_type']
[0, 6, 7, 292, 361, 296, 309]
['$', 'type_specifier', 'ID', '(', ')', '{', 'type_specifier']
[0, 6, 7, 292, 361, 296, 309, 310]
['$', 'type_specifier', 'ID', '(', ')', '{', 'type_specifier', 'ID']
[0, 6, 7, 292, 361, 296, 309, 310, 322]
```

逐节点输出抽象语法树

```

[0, 6, 7, 292, 361, 296, 297]
['$', 'type_specifier', 'ID', '(', ')', '{', 'block_items']
[0, 6, 7, 292, 361, 296, 297, 298]
['$', 'type_specifier', 'ID', '(', ')', '{', 'block_items', '}']
[0, 6, 7, 292, 361, 362]
['$', 'type_specifier', 'ID', '(', ')', 'compound_stmt']
[0, 5]
['$', 'fun_declaration']
[0, 2]
['$', 'declaration']
[0, 1]
['$', 'declaration_list']
index:185, node:declaration_list, child:[184], isLeaf:False, leafVal:None, pro
index:184, node:declaration, child:[183], isLeaf:False, leafVal:None, pro:{'lh
index:183, node:fun_declaration, child:[2, 3, 4, 5, 182], isLeaf:False, leafVa
'ID', '(', ')', 'compound_stmt']}, address:0
index:2, node:type_specifier, child:[1], isLeaf:False, leafVal:int, pro:{'lhs'
index:1, node:simple_type, child:[0], isLeaf:False, leafVal:int, pro:{'lhs': '
index:0, node:int, child:[], isLeaf:True, leafVal:int, pro:None, address:-1
index:3, node:main, child:[], isLeaf:True, leafVal:main, pro:None, address:-1
index:4, node:(, child:[], isLeaf:True, leafVal:(, pro:None, address:-1
index:5, node:), child:[], isLeaf:True, leafVal:), pro:None, address:-1
index:182, node:compound_stmt, child:[6, 180, 181], isLeaf:False, leafVal:None
address:0
index:6, node:{, child:[], isLeaf:True, leafVal:{, pro:None, address:-1

```

输出中间代码四元式，并在整个任务成功后输出True

```

index:152, node:term, child:[151], isLeaf:False, leafVal:i, pro:{'lhs': 'term', 'rhs': ['fa
index:151, node:factor, child:[150], isLeaf:False, leafVal:i, pro:{'lhs': 'factor', 'rhs':
index:150, node:var, child:[149], isLeaf:False, leafVal:i, pro:{'lhs': 'var', 'rhs': ['ID']
index:149, node:i, child:[], isLeaf:True, leafVal:i, pro:None, address:-1
index:157, node:;, child:[], isLeaf:True, leafVal:;, pro:None, address:-1
index:162, node:}, child:[], isLeaf:True, leafVal:}, pro:None, address:-1
index:172, node:}, child:[], isLeaf:True, leafVal:}, pro:None, address:-1
index:181, node:}, child:[], isLeaf:True, leafVal:}, pro:None, address:-1
0: (<, i, n, $0)
1: (jnz, $0, None, 3)
2: (j, None, None, 18)
3: (=, j, 1, None)
4: (+, i, 1, $1)
5: (=, i, $1, None)
6: (=, j, 1, None)
7: (<, j, 5, $2)
8: (jnz, $2, None, 10)
9: (j, None, None, 17)
10: (*, i, n, $4)
11: (/ , $4, n, $5)
12: (+, $5, i, $6)
13: (=, i, $6, None)
14: (+, j, 1, $3)
15: (=, j, $3, None)
16: (j, None, None, 7)
17: (j, None, None, 0)
True

```

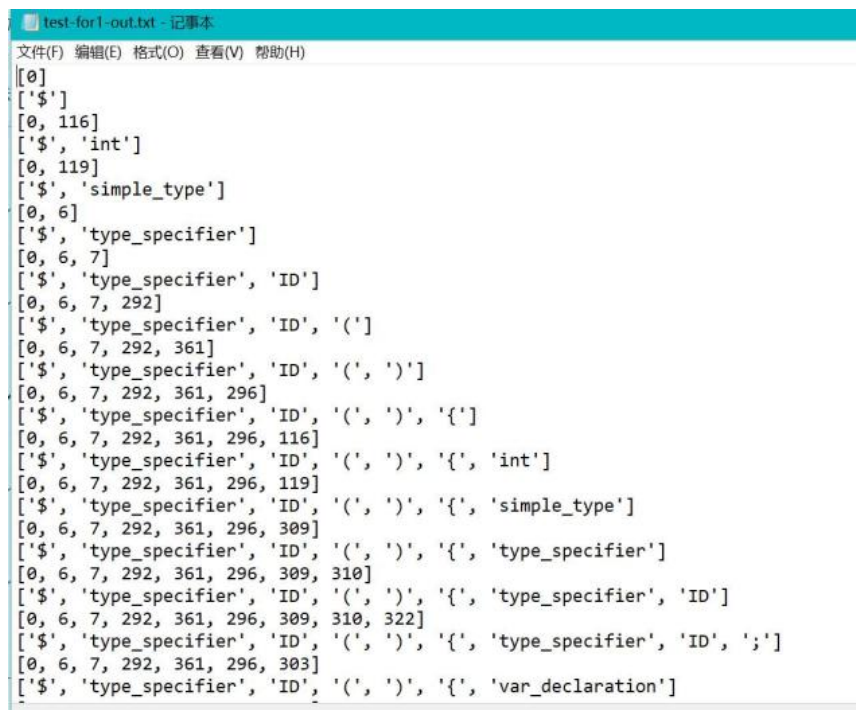
测试数据三（单个正确文件，for）

输入

```
int main() {  
    int i;  
    int n;  
    for (i = 0; i < n; i = i + 1) {  
        int j;  
        j = i;  
    }  
}
```

输出

首先输出的是文法分析过程



```
test-for1-out.txt - 记事本  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
[[0]  
['$']  
[0, 116]  
['$', 'int']  
[0, 119]  
['$', 'simple_type']  
[0, 6]  
['$', 'type_specifier']  
[0, 6, 7]  
['$', 'type_specifier', 'ID']  
[0, 6, 7, 292]  
['$', 'type_specifier', 'ID', '(']  
[0, 6, 7, 292, 361]  
['$', 'type_specifier', 'ID', '(', ')']  
[0, 6, 7, 292, 361, 296]  
['$', 'type_specifier', 'ID', '(', ')', '{']  
[0, 6, 7, 292, 361, 296, 116]  
['$', 'type_specifier', 'ID', '(', ')', '{', 'int']  
[0, 6, 7, 292, 361, 296, 119]  
['$', 'type_specifier', 'ID', '(', ')', '{', 'simple_type']  
[0, 6, 7, 292, 361, 296, 309]  
['$', 'type_specifier', 'ID', '(', ')', '{', 'type_specifier']  
[0, 6, 7, 292, 361, 296, 309, 310]  
['$', 'type_specifier', 'ID', '(', ')', '{', 'type_specifier', 'ID']  
[0, 6, 7, 292, 361, 296, 309, 310, 322]  
['$', 'type_specifier', 'ID', '(', ')', '{', 'type_specifier', 'ID', ';']  
[0, 6, 7, 292, 361, 296, 303]  
['$', 'type_specifier', 'ID', '(', ')', '{', 'var_declaration']
```

逐节点输出抽象语法树

```

test-for1-out.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
['$', 'type_specifier', 'ID', '(', ')', '{', 'block_items']
[0, 6, 7, 292, 361, 296, 297, 298]
['$', 'type_specifier', 'ID', '(', ')', '{', 'block_items', '}']
[0, 6, 7, 292, 361, 362]
['$', 'type_specifier', 'ID', '(', ')', 'compound_stmt']
[0, 5]
['$', 'fun_declaration']
[0, 2]
['$', 'declaration']
[0, 1]
['$', 'declaration_list']
index:107, node:declaration_list, child:[106], isLeaf:False, leafVal:
index:106, node:declaration, child:[105], isLeaf:False, leafVal:Non
index:105, node:fun_declaration, child:[2, 3, 4, 5, 104], isLeaf:Fa
['ID', '(', ')', 'compound_stmt']], address:0
index:2, node:type_specifier, child:[1], isLeaf:False, leafVal:int,
index:1, node:simple_type, child:[0], isLeaf:False, leafVal:int, pr
index:0, node:int, child:[], isLeaf:True, leafVal:int, pro:None, ad
index:3, node:main, child:[], isLeaf:True, leafVal:main, pro:None,
index:4, node:(, child:[], isLeaf:True, leafVal:(, pro:None, addres
index:5, node:), child:[], isLeaf:True, leafVal:), pro:None, addres
index:104, node:compound_stmt, child:[6, 102, 103], isLeaf:False, l
address:0
index:6, node:{, child:[], isLeaf:True, leafVal:{, pro:None, addres
index:102, node:block_items, child:[101], isLeaf:False, leafVal:Non
index:101, node:statement_list, child:[13, 100], isLeaf:False, leaf
'statement_list']], address:0
index:13, node:statement, child:[12], isLeaf:False, leafVal:None, p

```

输出中间代码四元式，并在整个任务成功后输出True

```

index:77, node:j, child:[], isLeaf:True, leafVal:j, pro:None, address:-1
index:79, node:=, child:[], isLeaf:True, leafVal:=, pro:None, address:-1
index:86, node:expression, child:[85], isLeaf:False, leafVal:i, pro:({'lhs': 'express
index:85, node:simple_expression, child:[84], isLeaf:False, leafVal:i, pro:({'lhs': '
address:-1
index:84, node:additive_expression, child:[83], isLeaf:False, leafVal:i, pro:({'lhs':
index:83, node:term, child:[82], isLeaf:False, leafVal:i, pro:({'lhs': 'term', 'rhs':
index:82, node:factor, child:[81], isLeaf:False, leafVal:i, pro:({'lhs': 'factor', 'r
index:81, node:var, child:[80], isLeaf:False, leafVal:i, pro:({'lhs': 'var', 'rhs': [
index:80, node:i, child:[], isLeaf:True, leafVal:i, pro:None, address:-1
index:88, node:;, child:[], isLeaf:True, leafVal:;, pro:None, address:-1
index:94, node:}, child:[], isLeaf:True, leafVal:}, pro:None, address:-1
index:103, node:}, child:[], isLeaf:True, leafVal:}, pro:None, address:-1
0: (=, i, 0, None)
1: (<, i, n, $0)
2: (jnz, $0, None, 4)
3: (j, None, None, 8)
4: (=, j, i, None)
5: (+, i, 1, $1)
6: (=, i, $1, None)
7: (j, None, None, 1)
True

```

测试数据四（单个正确文件，综合）

输入

```

int main(){
//    a = a + 1;
    int a;
    int b;
    a = b + 1;
    for(a = 1; a < 2; a = a + 1)
    {
        a = 1;
        if(a < b){

```



```

        a = a + 1;
    }else{
        a = b * (a + 1);
    }
}
}

```

输出

首先输出的是文法分析过程

```

test-out.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
[0]
['$']
[0, 116]
['$', 'int']
[0, 119]
['$', 'simple_type']
[0, 6]
['$', 'type_specifier']
[0, 6, 7]
['$', 'type_specifier', 'ID']
[0, 6, 7, 292]
['$', 'type_specifier', 'ID', '(']
[0, 6, 7, 292, 361]
['$', 'type_specifier', 'ID', '(', ')']
[0, 6, 7, 292, 361, 296]
['$', 'type_specifier', 'ID', '(', ')', '{']
[0, 6, 7, 292, 361, 296]
['$', 'type_specifier', 'ID', '(', ')', '{']
[0, 6, 7, 292, 361, 296, 116]
['$', 'type_specifier', 'ID', '(', ')', '{', 'int']
[0, 6, 7, 292, 361, 296, 119]
['$', 'type_specifier', 'ID', '(', ')', '{', 'simple_type']
[0, 6, 7, 292, 361, 296, 309]
['$', 'type_specifier', 'ID', '(', ')', '{', 'type_specifier']
[0, 6, 7, 292, 361, 296, 309, 310]
['$', 'type_specifier', 'ID', '(', ')', '{', 'type_specifier', 'ID']
[0, 6, 7, 292, 361, 296, 309, 310, 322]
['$', 'type_specifier', 'ID', '(', ')', '{', 'type_specifier', 'ID', ';']

```

逐节点输出抽象语法树

```

test-out.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
['$', 'type_specifier', 'ID', '(', ')', '{', 'statement', 'statement', 'statement_list']
[0, 6, 7, 292, 361, 296, 301, 302]
['$', 'type_specifier', 'ID', '(', ')', '{', 'statement', 'statement_list']
[0, 6, 7, 292, 361, 296, 300]
['$', 'type_specifier', 'ID', '(', ')', '{', 'statement_list']
[0, 6, 7, 292, 361, 296, 297]
['$', 'type_specifier', 'ID', '(', ')', '{', 'block_items']
[0, 6, 7, 292, 361, 296, 297, 298]
['$', 'type_specifier', 'ID', '(', ')', '{', 'block_items', '}']
[0, 6, 7, 292, 361, 362]
['$', 'type_specifier', 'ID', '(', ')', 'compound_stmt']
[0, 5]
['$', 'fun_declaration']
[0, 2]
['$', 'declaration']
[0, 1]
['$', 'declaration_list']
index:204, node:declaration_list, child:[203], isLeaf:False, leafVal:None, pro: {'lhs': 'dec
index:203, node:declaration, child:[202], isLeaf:False, leafVal:None, pro: {'lhs': 'declarat
index:202, node:fun_declaration, child:[2, 3, 4, 5, 201], isLeaf:False, leafVal:None, pro: {'
ID', '(', ')', 'compound_stmt'}}, address:0
index:2, node:type_specifier, child:[1], isLeaf:False, leafVal:int, pro: {'lhs': 'type_speci
index:1, node:simple_type, child:[0], isLeaf:False, leafVal:int, pro: {'lhs': 'simple_type',
index:0, node:int, child:[], isLeaf:True, leafVal:int, pro:None, address:-1
index:3, node:main, child:[], isLeaf:True, leafVal:main, pro:None, address:-1
index:4, node:(, child:[], isLeaf:True, leafVal:(, pro:None, address:-1
index:5, node:), child:[], isLeaf:True, leafVal:), pro:None, address:-1
index:201, node:compound_stmt, child:[6, 199, 200], isLeaf:False, leafVal:None, pro: {'lhs':

```

输出中间代码四元式，并在整个任务成功后输出True

```
test-out.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
index:157, node:a, child:[], isLeaf:True, leafVal:a, pro:None, address:-1
index:163, node:addop, child:[162], isLeaf:False, leafVal:+, pro: {'lhs': 'addop', 'rhs': ['+']},
index:162, node:+, child:[], isLeaf:True, leafVal:+, pro:None, address:-1
index:166, node:term, child:[165], isLeaf:False, leafVal:1, pro: {'lhs': 'term', 'rhs': ['factor']
index:165, node:factor, child:[164], isLeaf:False, leafVal:1, pro: {'lhs': 'factor', 'rhs': ['NUM
index:164, node:1, child:[], isLeaf:True, leafVal:1, pro:None, address:-1
index:170, node:), child:[], isLeaf:True, leafVal:), pro:None, address:-1
index:177, node:;, child:[], isLeaf:True, leafVal:;, pro:None, address:-1
index:182, node:}, child:[], isLeaf:True, leafVal:}, pro:None, address:-1
index:190, node:}, child:[], isLeaf:True, leafVal:}, pro:None, address:-1
index:200, node:}, child:[], isLeaf:True, leafVal:}, pro:None, address:-1
0: (+, b, 1, $0)
1: (=, a, $0, None)
2: (=, a, 1, None)
3: (<, a, 2, $1)
4: (jnz, $1, None, 6)
5: (j, None, None, 18)
6: (=, a, 1, None)
7: (<, a, b, $3)
8: (jnz, $3, None, 10)
9: (j, None, None, 12)
10: (+, a, 1, $4)
11: (=, a, $4, None)
12: (+, a, 1, $5)
13: (*, b, $5, $6)
14: (=, a, $6, None)
15: (+, a, 1, $2)
16: (=, a, $2, None)
17: (j, None, None, 3)
True
```

测试数据五（单个错误文件）

输入

```
int main(){
    if(a < b){
        if(a > b){
            a = a + 1;
        }else{
            a = a + 2;
        }
    }
}
```

输出

未能成功，最后输出False

```
test-err-if-out.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
[0]
['$']
[0, 116]
['$', 'int']
[0, 119]
['$', 'simple_type']
[0, 6]
['$', 'type_specifier']
[0, 6, 7]
['$', 'type_specifier', 'ID']
[0, 6, 7, 292]
['$', 'type_specifier', 'ID', '(']
[0, 6, 7, 292, 361]
['$', 'type_specifier', 'ID', '(', ')']
[0, 6, 7, 292, 361, 296]
['$', 'type_specifier', 'ID', '(', ')', '{']
[0, 6, 7, 292, 361, 296, 329]
['$', 'type_specifier', 'ID', '(', ')', '{', 'if']
[0, 6, 7, 292, 361, 296, 329, 330]
['$', 'type_specifier', 'ID', '(', ')', '{', 'if', '(']
[0, 6, 7, 292, 361, 296, 329, 330, 90]
['$', 'type_specifier', 'ID', '(', ')', '{', 'if', '(', 'ID']
[0, 6, 7, 292, 361, 296, 329, 330, 36]
['$', 'type_specifier', 'ID', '(', ')', '{', 'if', '(', 'var']
Error: No declared identifier a
False
```

调试问题思考与解决

文法构造

我们使用的文法是小组成员自己一点点修改完善出来的，不是网上直接搬来的，也不是老师提供的。
(当我们实现功能的时候老师还没有给我们提供文法，在中间代码生成部分也对此进行了相应的调整)

实际上，文法的构造是及其耗费心力的。某种程度上来说，这甚至比代码的实现还要麻烦。

参考我们的输入测试一，可以看到我们的识别已经是相对来说完成度相当高了，但相对的，能识别这样复杂的测试代码本身对文法的要求也是相当高，因此在构造中也经过了相当多的修改调整，这里举一个例子类比我们遇到的一个问题及其解决。

这里仅仅是举例，并非当时的具体内容：

我们早期的文法中对有这样的语句：

```
statement_list -> statement_list statement | empty
```

其中 `empty` 表示的是空，也就是课上通常讲的 ϵ 。

使用这样的形式的文法，我们在识别和如下两句类似的情况时出现了问题：

```
int a = b + c;
int a;
```

最后的解决方法是将空消除掉，消除方式是将产生式右侧出现的空直接放到前一个用来产生这个产生式左侧的产生式中。

例如：

```
A -> B | C
B -> D | ε
```

就改成：

```
A -> C | D
```

按照此种方式就可以解决类似的问题。

中间代码生成四元式的处理

这里以for语句为例，说明for语句的四元式是如何实现的。

在 `IRGenerator` 的 `dfsASTree` 中，深度优先遍历抽象语法树，当遇到for语句时，处理代码如下所示：

```
elif productionStr == SpecialProductions["for"]:
    # for(a;b;c) { d }
    # segA segB jnz j segD segC j
    segBefore = self.IRCode[0:quaterAddress[2]]
    segA = self.IRCode[quaterAddress[2]:quaterAddress[3]]
    segB = self.IRCode[quaterAddress[3]:quaterAddress[4]]
    segC = self.IRCode[quaterAddress[4]:quaterAddress[6]]
    segD = self.IRCode[quaterAddress[6]:]

    quater.op = "jnz"
    quater.arg1 = leafVals[3]
    quater.result = len(segBefore) + len(segA) + len(segB) + 2

    quater1 = Quaternary()
    quater1.op = "j"
    quater1.result = len(self.IRCode) + 3

    quater2 = Quaternary()
    quater2.op = "j"
    quater2.result = quaterAddress[3]

    for i in range(len(segC)):
        code = segC[i]
        if code.op == "j" or code.op == "jnz":
            segC[i].result += 2
    for i in range(len(segD)):
        code = segD[i]
        if code.op == "j" or code.op == "jnz":
            segD[i].result += 2 - len(segC)
    self.IRCode = segBefore + segA + segB + [quater, quater1] + segD
    + segC + [quater2]

    faNode.quaterAddress = quaterAddress[2]
    ASTree.tree[faNode.index] = faNode
    ASTree.tree[faNode.child[4]].quaterAddress = len(segBefore +
    segA + segB + [quater, quater1] + segD)
```



```
ASTree.tree[faNode.child[6]].quaterAddress = len(segBefore +
segA + segB + [quater, quater1])
```

对这部分代码处理了for循环语句，使之生成对应的中间代码，并且确保循环体内的语句的正确执行。

for语句的结构大致为：

```
<Before>;
for(<A>; <B>; <C>){
    <D>;//这里的D不一定是简单语句，也可能是复杂结构包括另一个for循环，所以需要
    对循环体内的语句进行处理
}
```

1. 分段划分：

- 将整个中间代码分为五个段：segBefore、segA、segB、segC、segD，每个段对应for循环不同的阶段。

2. 设置条件跳转：

- quater.op = "jnz"：将条件跳转的操作码设置为"jnz" (jump if not zero)。
- quater.arg1 = leafVals[3]：将条件跳转的条件设置为for循环的条件。
- quater.result = len(segBefore) + len(segA) + len(segB) + 2：设置跳转目标的位置，确保跳转到for循环体内。

3. 设置无条件跳转：

- quater1.op = "j" 和 quater1.result = len(self.IRCode) + 3：设置一个无条件跳转的操作码和目标，确保在循环体执行完后跳转到for循环体内的条件判断语句。
- quater2.op = "j" 和 quater2.result = quaterAddress[3]：设置另一个无条件跳转，确保在条件判断不满足时跳转到循环体外。

4. 更新跳转目标：

- 对segC中的跳转语句进行遍历，更新其目标地址。确保跳转到循环体外的语句保持正确的相对位置。

5. 更新循环体内语句执行后的跳转目标：

- 对segD中的跳转语句进行遍历，更新其目标地址。同样，确保跳转到循环体外的语句保持正确的相对位置。

6. 拼接新的中间代码：

- 将各个段按照顺序拼接成新的中间代码。这确保了for循环语句的各个部分按照正确的顺序组织在一起。

7. 更新语法树节点的中间代码生成地址：

- 将faNode.quaterAddress更新为quaterAddress[2]，这是for循环初始化语句的位置。

8. 更新for循环体内的第一个语句的中间代码生成地址：

- 将ASTree.tree[faNode.child[4]].quaterAddress更新为循环体内第一个语句的起始位置。

9. 更新for循环体内的第二个语句（条件判断语句）的中间代码生成地址：

- 将ASTree.tree[faNode.child[6]].quaterAddress更新为循环体内第二个语句（条件判断语句）的起始位置。

通过以上步骤，该段代码确保了对for循环语句的正确处理。具体来说，它生成了对应的条件跳转和无条件跳转的中间代码，同时保持了跳转目标的正确性。这样可以确保循环体内的语句能够按照预期的逻辑执行，实现了for循环语句的中间代码生成。

五、总结与收获

收获

1. 深入理解编译原理原理：

实现LR(1)文法的词法和语法分析，需要对编译原理的相关理论有深入的理解。这使我们更加熟悉自底向上的语法分析方法、First和Follow集合、LR(1)项、LR(1)自动机等概念。

2. 实际应用能力：

实现编译器的部分功能是对编译原理理论的实际应用，这有助于将抽象的理论知识转化为实际可运行的程序。

3. Python编程技能：

在编写编译器时，我们学习并熟悉和锻炼了许多Python编程技能，包括文件读写、数据结构的使用、递归、异常处理等。这有助于提升我们的编程能力。

4. 深入理解编译原理：

编写中间代码生成器需要对编译原理的基本概念有深入的理解，包括文法、语法树、语义分析等。这让我们更好地理解代码的组织结构和执行流程。

5. 学会使用工具：

使用了一些工具、数据结构和库函数，这展示了在实际项目中利用工具提高效率的重要性。

6. 理解语法分析和中间代码生成的关系：

中间代码生成是编译过程的一个重要阶段，它在语法分析后进行。通过实现中间代码生成器，我们深入理解了语法分析和中间代码生成之间的关联。

7. 处理语法规则和产生式：

编写编译器涉及处理文法规则和产生式，这加深了对语法结构和语言表达能力的理解。

8. 错误处理和调试：

编写编译器时，我们可能遇到了各种各样的错误。学会调试和处理错误是编程实践中的重要技能。

9. 团队协作：

作为一个团队项目，我们团队成员相互协助解决问题，提高团队协作和沟通的能力。

10. 对计算机科学理论的应用：

编译原理是计算机科学理论的实际应用之一。通过实现中间代码生成器，我们将理论知识应用到实际问题中，提升了对计算机科学的理解。

对课程的认识与感受

1. 实现了实践与理论结合：

编写编译器使我们更加深入理解编译原理的理论知识。这种实践与理论结合的方式使我们更好地理解课堂上学到的概念。

2. 拥有了对复杂性的认识：

编译器是软件工程中复杂的一部分。通过实现编译器，我们更加认识到软件工程中处理复杂问题的挑战。

3. 获得了深度学习机会：

通过编写编译器，我们有机会深入研究更高级的编译原理概念，例如优化、中间代码生成等。

编写编译器中间代码生成器的实验是一次充满挑战和收获的学习经历。在这个过程中，我们深刻体会到了编译原理在实际项目中的应用，同时也锻炼了我们的编程和问题解决能力。

首先，通过实现中间代码生成器，我们深入了解了编译原理的核心概念，包括语法规则、语法分析和中间代码生成。对于一个程序如何从源代码到中间代码的转化过程有了更清晰的认识，这对于理解编程语言的内部工作原理具有重要意义。

其次，实践中间代码生成让我们直面了语法规则和产生式的处理，学会了如何解析复杂的语法结构并将其转化为中间代码。这锻炼了我们对于语法和文法的理解能力，提高了对编程语言设计的敏感性。

在调试和处理错误的过程中，我们学到了如何有效地定位和解决问题。编写编译器时，错误可能涉及多个层面，包括语法分析、语义分析和中间代码生成，因此善于调试和处理错误是一个必备的技能。

与此同时，我们学到了如何有效利用工具，提高了代码可读性和理解性。这也是实际项目中常见的做法，通过工具提高工作效率。

总的来说，这次实验让我们不仅更加深入地理解了编译原理的理论知识，也在实践中掌握了一些关键的编程技能。这种将理论与实际结合的学习方式，使我们在计算机科学领域取得了更为全面和实际的进步。通过克服挑战，我们对编译原理及其在软件开发中的重要性有了更为深刻的认识。这次经历不仅为我们个人的技术积累提供了丰富的内容，也为未来的学习和工作奠定了坚实的基础。

六、参考文献

- [1] 陈火旺等. 程序设计语言编译原理（第3版）. 国防工业出版社, 2000
- [2] GQT. 词法分析器. 2021
- [3] Jutta Degener. ANSI C Yacc grammar. 2004
- [4] Jutta Degener. ANSI C grammar, Lex specification. 2017
- [5] ISO. ISO/IEC 9899:1999 Programming languages — C, 1999