电子科技大学

计算机专业类课程

实验报告

课程名称：编译原理

学　　院：计算机科学与工程学院

专　　业：计算机科学与技术

学生姓名：朱若愚

学　　号：2022150501027

指导教师：陈昆

日　　期：　　2025 年 5 月 10 日

电 子 科 技 大 学

实 验 报 告

实验一

一、实验名称：词法分析

二、实验学时：4

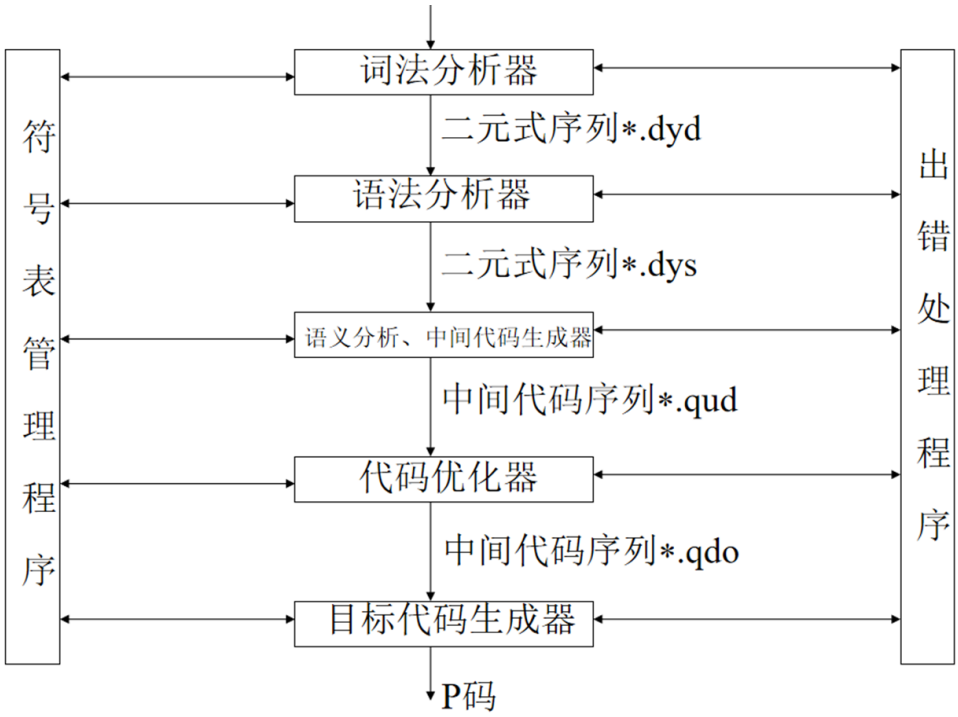
三、实验内容和目的：

1. 理解词法分析的原理和编程实现方式。
2. 熟悉状态转换图。
3. 对求 n!的极小语言进行词法分析，输出二元式文件（*.dyd）和错误信息文件（*.err）。

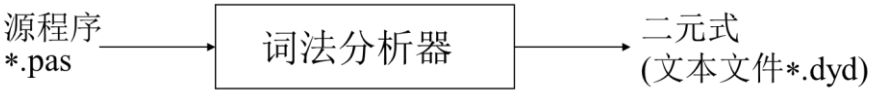
四、实验原理：

编译器将源程序看成一个很长的字符串，首先对它进行从左到右的扫描，并进行分析，识别出符合词法规则的单词，如基本字、标识符、常数、运算符和界符等。这些符号以固定的编码提供给后续的语法分析进一步处理。如果在词法分析过程中发现不符合词法规则的非法单词符号，则做出词法出错处理，给出相应的出错信息。

1. 编译程序的结构



2. 词法分析器

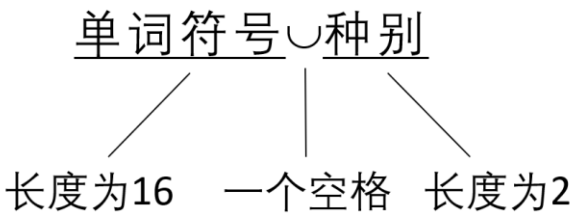


3. 单词符号与种类对照表

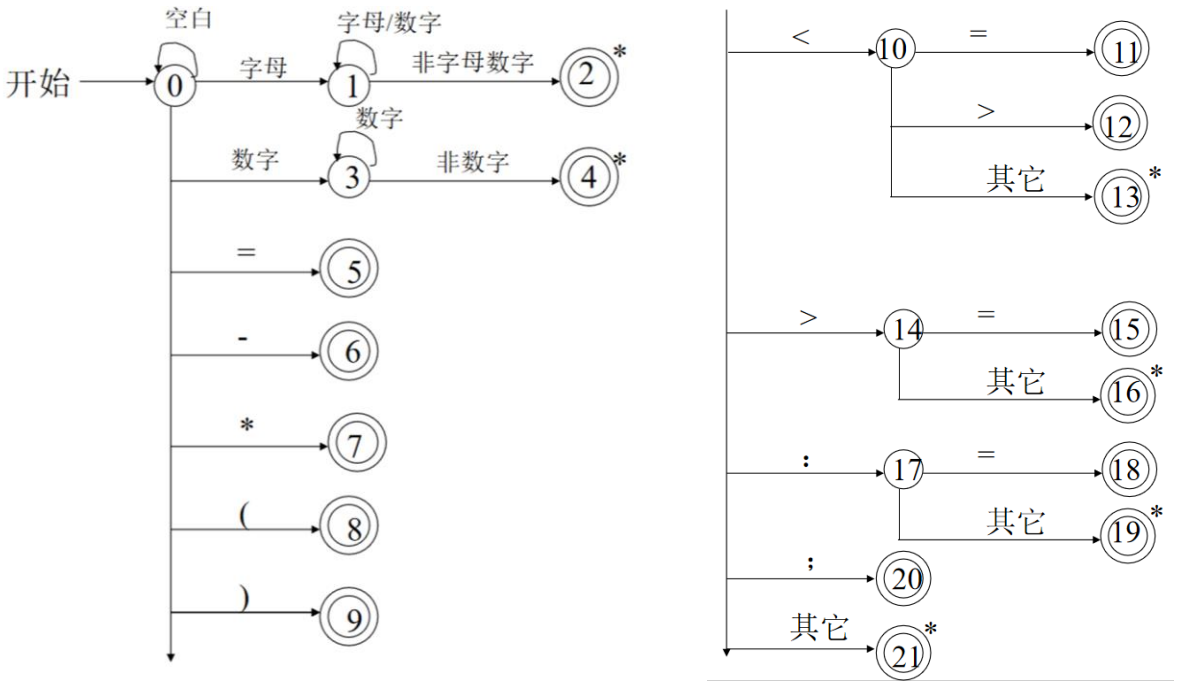
单词符号与种别对照表

单词符号	种别	单词符号	种别	单词符号	种别
begin	1	end	2	integer	3
if	4	then	5	else	6
function	7	read	8	write	9
标识符	10	常数	11	=	12
<>	13	<=	14	<	15
>=	16	>	17	-	18
*	19	:=	20	(	21
)	22	;	23		

4. 二原式格式标准



5. 状态转换图



五、实验目的：

1. 完成词法分析，语法分析。
2. 词法分析与语法分析必须有出错信息。

五、实验器材（设备、元器件）

Windows 11 操作系统
Python 3.12 环境

六、实验步骤：

1.读入源程序。
2.根据状态转换图，从左到右扫描源程序字符串，识别出所有单词。
3.将识别到的单词写入二元式文件。
4.将发现的错误写入错误信息文件。
5.检查二元式文件内容是否正确。
6.检查错误信息文件是否包含所有的错误。

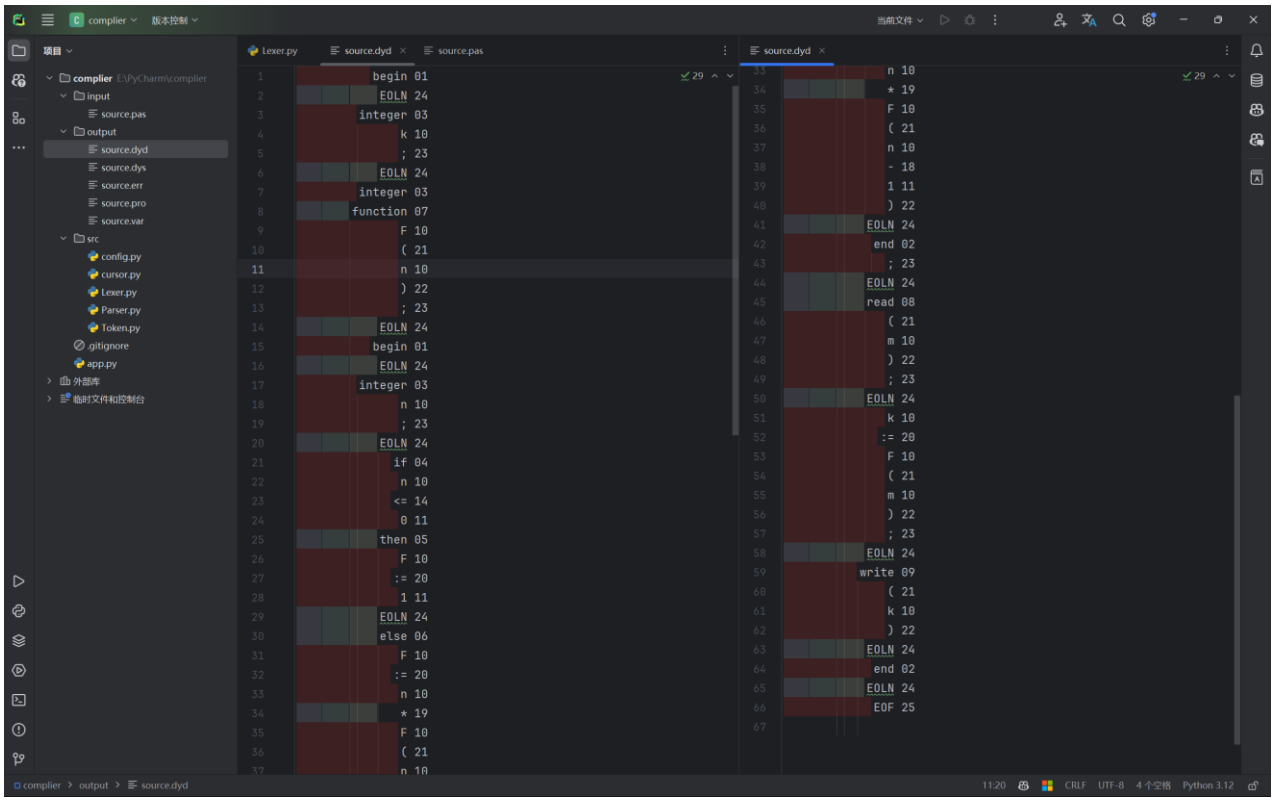
七、实验数据及结果分析：

1. 读入源程序，根据状态转换图，从左到右扫描源程序字符串，识别出所有单词。

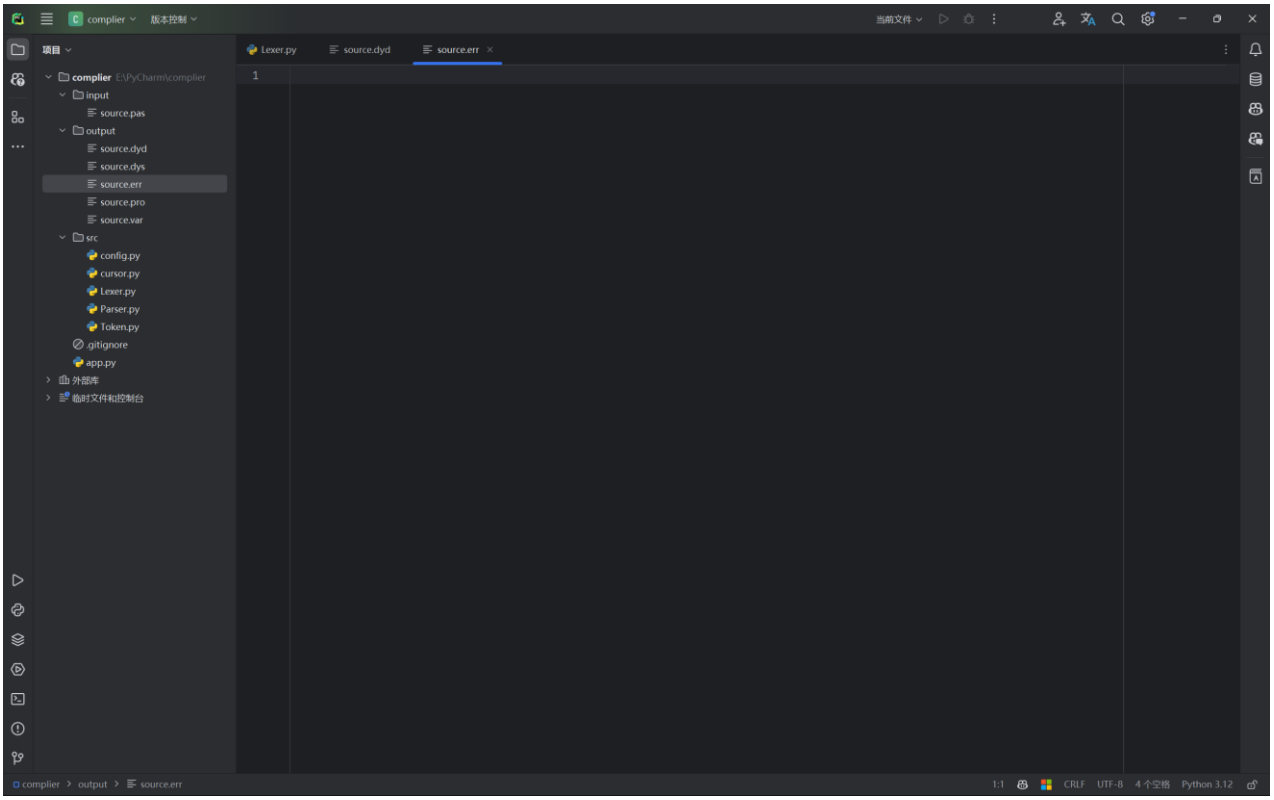
源程序 Source.pas

```
begin
  integer k;
  integer function F(n);
    begin
      integer n;
      if n<=0 then F:=1
      else F:=n*F(n-1)
    end;
  read(m);
  k:=F(m);
  write(k)
end
```

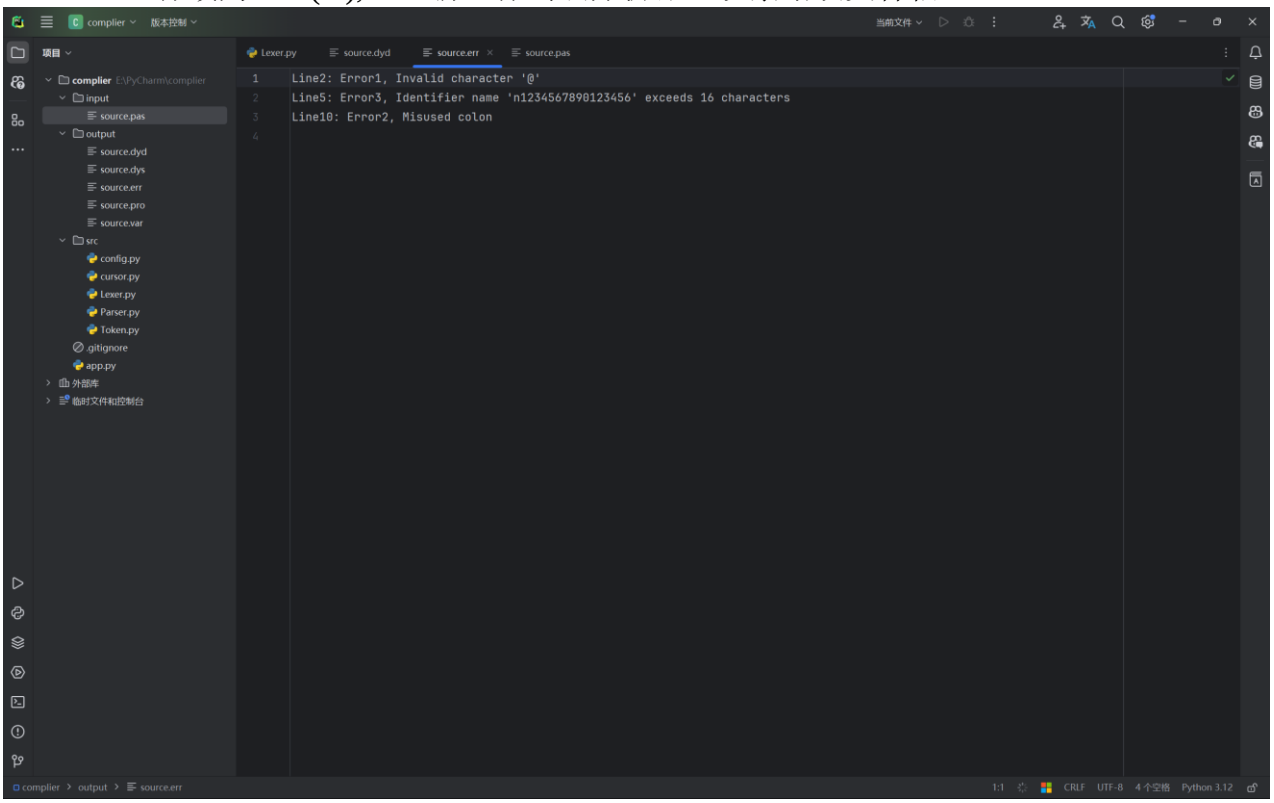
2. 将识别到的单词写入二元式文件。



3. 将发现的错误写入错误信息文件，此时无错误。



4. 将源程序第二行改为 integer @k;，第五行改为 integer n1234567890123456;，第十行改为 k: F(m);，重新运行词法分析器，观察错误文件信息。



正确识别到三种错误并指出错误所在行数。

八、实验结论、心得体会和改进建议：

在本次实验中，我设计并实现了一个词法分析器。该词法分析器能够精准识别源程序中的各类单词，对于出现的错误，它会提供具有高度描述性的错误信息，并明确指出错误所在的行数，全面满足了实验的所有既定要求。

通过此次实验实践，我对词法分析的底层原理有了更为深入且透彻的理解，也在亲身实践中熟练掌握了如何运用编程语言将抽象的状态转换图转化为具体的代码实现。同时，我深刻认识到，在编译器开发过程中，精准且友好的报错机制与词法分析本身同等重要。良好的报错机制能够帮助源程序开发者迅速定位并修正错误，从而使编译器更好地履行其协助开发者构建高质量程序的职责。

电 子 科 技 大 学

实 验 报 告

实验二

一、实验名称：语法分析

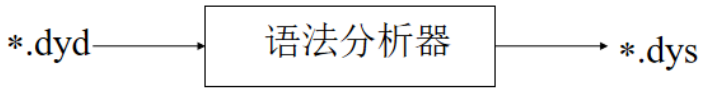
二、实验学时：4

三、实验内容和目的：

1. 理解语法分析的原理和编程实现方式。
2. 理解递归下降法。
3. 对词法分析生成的二元式文件进行语法分析，输出二元式文件（*.dys）、变量表（*.var）、过程表（*.pro）和错误信息文件（*.err）。

四、实验原理：

语法分析是对词法分析识别出来的符号流（也可看成符号串），按语法规则进行分析，识别出各类语法单位，如表达式、短语、子句、句子和程序等，以便后续步骤进行分析与合成。语法分析通常是一种结构分析，分析的结果形成一棵语法树（分析树）。如果在分析过程中发现不符合语法规则的符号串，将做出语法出错处理，给出相应的出错信息。



同时, 产生文本文件*.var、 *.pro

变量名表：

变量名 vname: char(16)
所属过程 vproc:char(16)
分类 vkind: 0..1(0—变量、1—形参)
变量类型 vtype: types
变量层次 vlev: int
变量在变量表中的位置 vadr: int(相对第一个变量而言)types=(ints)

过程名表：

过程名 pname: char(16)
过程类型 ptype: types
过程层次 plev: int
第一个变量在变量表中的位置 fadr: int

最后一个变量在变量表中的位置 ladr: int

错误信息：

(1)缺少符号错;

(2)符号匹配错;

(3)符号无定义或重复定义。

消除左递归后文法：

PG -> SUBPG eof

SUBPG -> begin DCLS EXES end

DCLS -> DCL DCLS'

DCLS' -> DCL DCLS' | e

DCL -> integer DCL';

DCL' -> VARDCL | PRODCL

VARDCL -> ID

VAR -> ID

PRODCL -> function PRONAMEDCL (PARAMDCL); PROBODY

PRONAMEDCL -> ID

PRONAME -> ID

PARAMDCL -> ID

PARAM -> ID

PROBODY -> begin DCLS EXES end

EXES -> EXE EXES'

EXES' -> ; EXE EXES' | e

EXE -> READ | WRITE | ASSIGN | CONDITION

READ -> read (VAR)

WRITE -> write (VAR)

ASSIGN -> VAR := ARITH | PRONAME := ARITH

ARITH -> TERM ARITH'

ARITH' -> - TERM ARITH' | e

TERM -> FACTOR TERM'

TERM' -> * FACTOR TERM' | e

FACTOR -> VAR | CONST | CALL

CALL -> PRONAME (ARITH)

CONDITION = if CDT then EXE else EXE

CDT -> ARITH OP ARITH

OP -> = | <> | < | <= | > | >=

递归下降分析法原理：

递归下降分析法是一种自顶向下的语法分析方法。所谓自顶向下，就是从文法的开始符号出发，尝试用替换规则，逐步向下推导，试图匹配输入的字符串。它通过为文法中的每个非终结符设置一个过程（或函数）来实现语法分析。当进行推导时，如果某个非终结符的可能产生式有多种选择，就按照一定的顺序（通常是固定的顺序，如从左到右）来尝试这些产生式。

五、实验器材（设备、元器件）

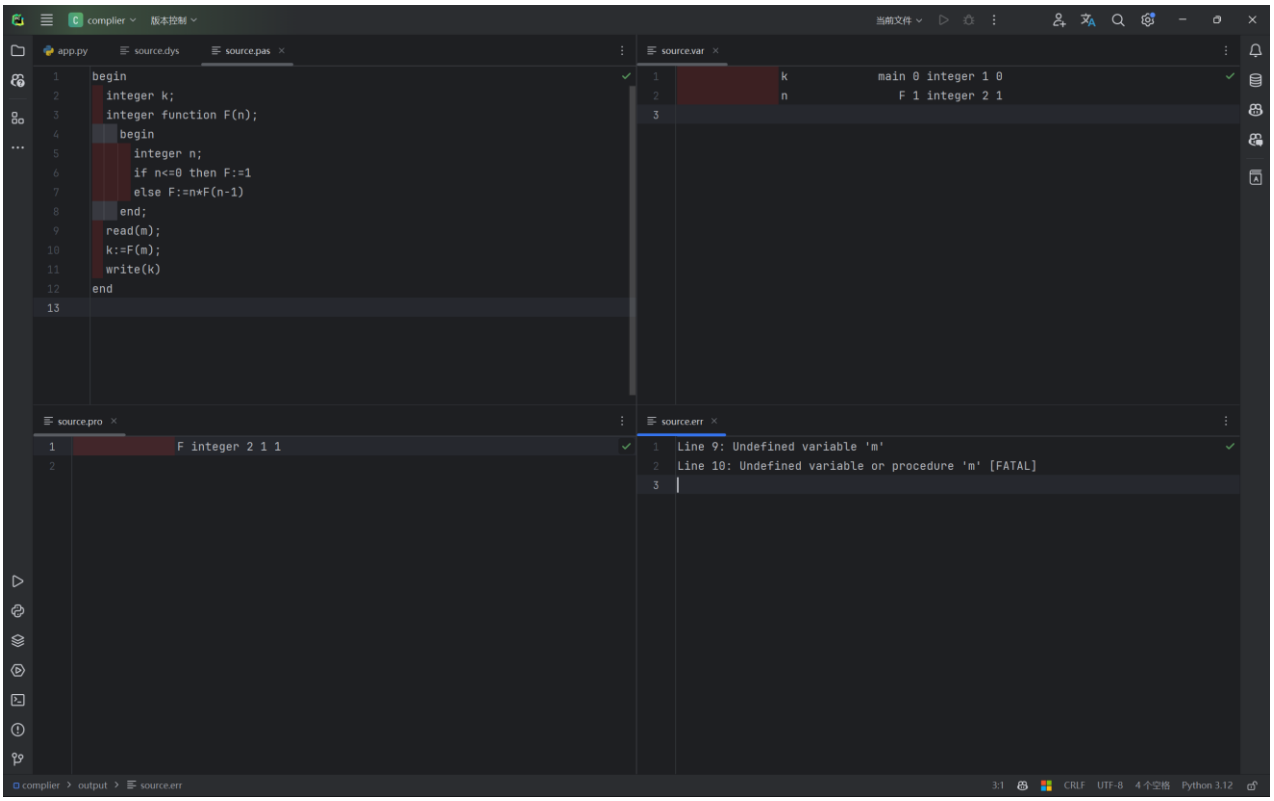
Windows 11 操作系统
Python 3.12 环境

六、实验步骤：

1.读入二元式文件。
2.根据递归下降文法，从上到下尝试匹配所有符号。
3.匹配失败则报错。如果不是重大错误，就尝试恢复。
4.写入二元式、变量表、过程表和错误信息文件

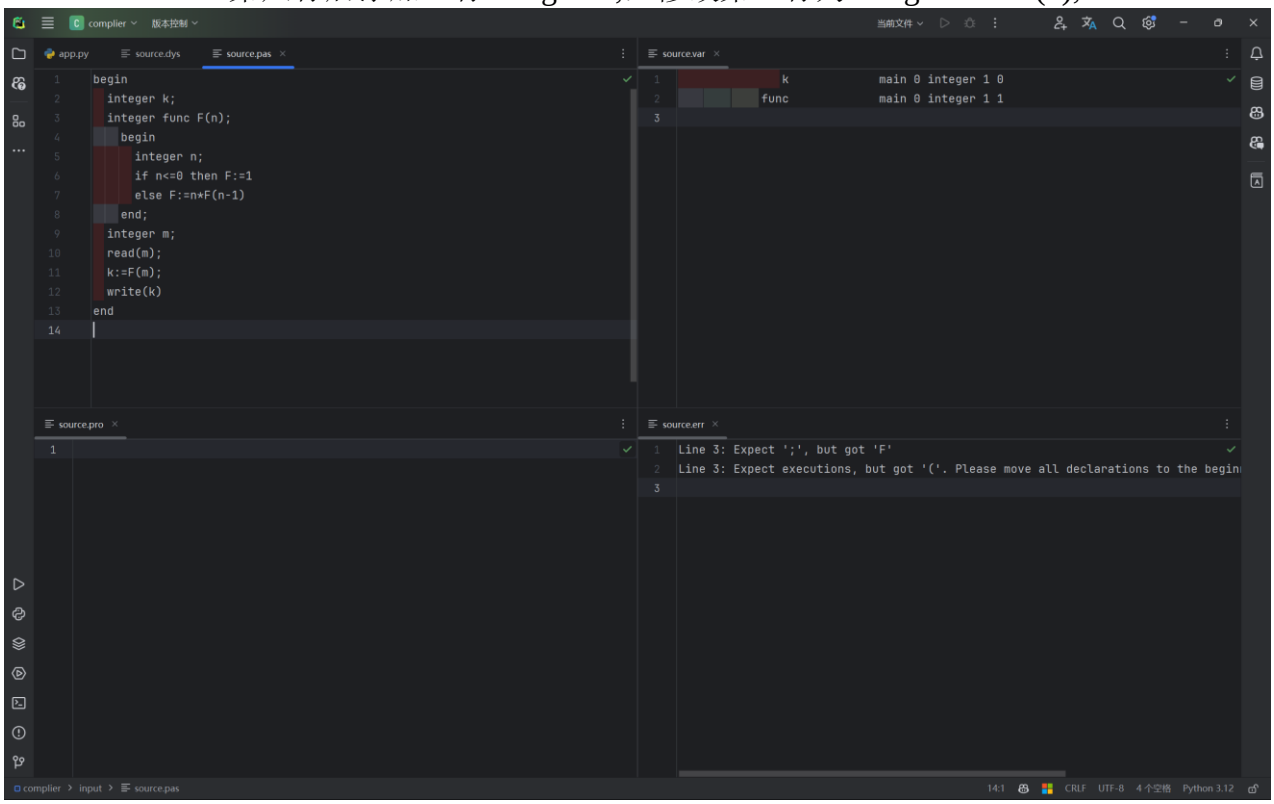
七、实验数据及结果分析：

1. 源程序测试



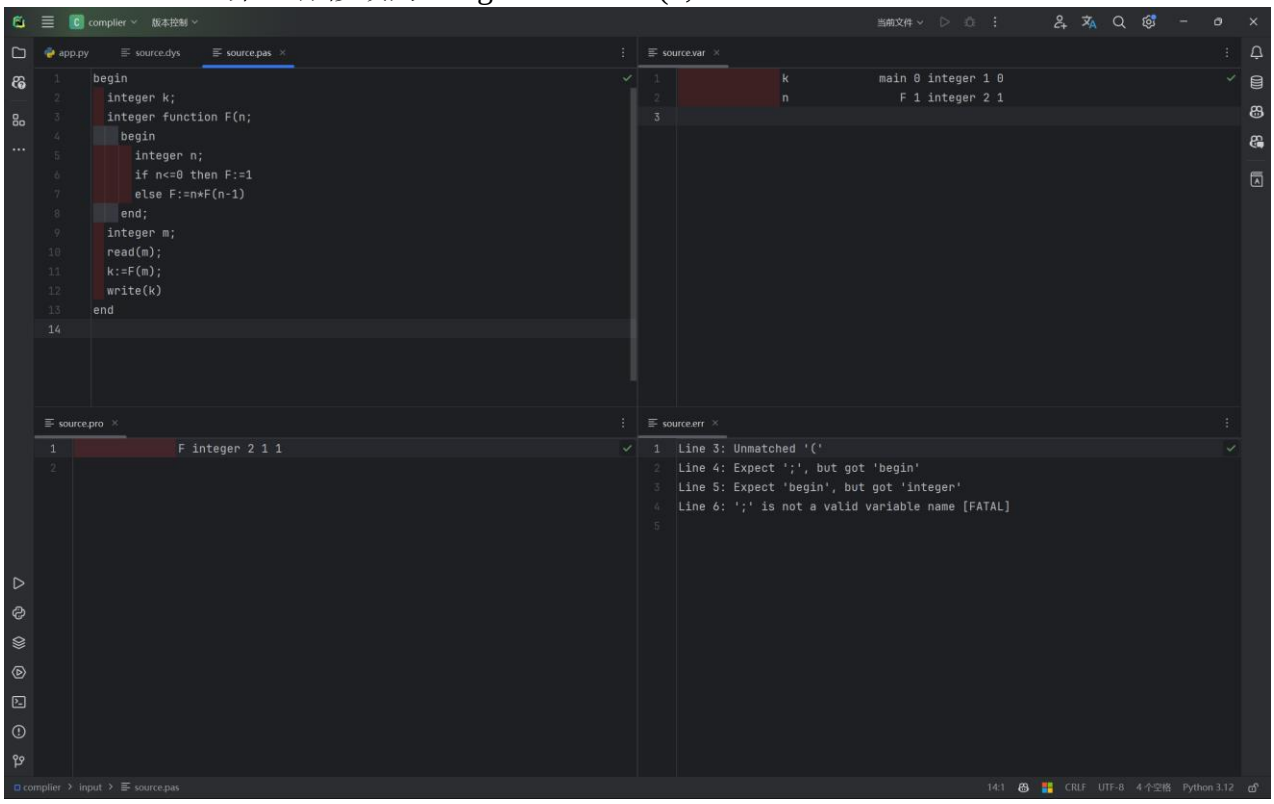
正确识别变量名，过程名。此时报错没有定义的变量 m。

2. 第八行后添加一行 integer m;，修改第三行为 integer func F(n);



修改后，分析器认为 func 是一个变量因而希望在变量名后出现;，同样跳过 F 后接收到(。

3. 第三行修改为 integer function F(n;



此时没有匹配(报错，同时跳过继续读入得到 begin 报错没有;。

八、实验结论、心得体会和改进建议：

在本次实验中，我设计并实现了一个功能完备的语法分析器。通过递归下降的方法对词法分析得到的结果逐个分析并指出所包含的三种错误类型。进一步深刻理解和掌握了语法分析的作用和递归下降分析法的原理和使用。通过输出四种类型的文件，合理地输出过程、变量以及精准地指出错误类型和位置。

建议在实验时进一步强调递归下降方法的原理和一个简单的代码示例。

Lexer.py

```
import sys;sys.path.append("..")
if __name__ != "__main__":
    sys.path.append("src")
from Token import TokenType, Token
from cursor import Cursor
from config import *

# 标志符的最大长度
MAX_IDENTIFIER_LENGTH = 16

class Lexer():
    # 成员变量：文件内容
    def __init__(self):
        self.line = 1 # 统计第几行的错误
        self.cursor = Cursor(Lexer.readSource())
        self.tokens = []
        self.errors = []
        self.tokenize()

    def tokenize(self):
        # 文件还没有读取完
        while self.cursor.isOpen():
            token = self.getNextToken()
            if token.mytype.name != 'ERROR':
                self.tokens.append(token)
            # 加入错误信息
            else: self.errors.append(token.value)

        # 放入文件结束符
        self.tokens.append(Token(TokenType.END_OF_FILE, 'EOF'))

        # 最后分别写入符号表和错误表
        self.writeTokens()
        self.writeErrors()

    def getNextToken(self):
        # 跳过空格
        while self.cursor.current() == ' ':
            self.cursor.consume()

        # initial 是符号的开始
        initial = self.cursor.consume()
```

```
        if Lexer.isLetter(initial):
            value = initial
            # 读取整个标识符或关键字
            while self.cursor.isOpen() and (Lexer.isLetter(self.cursor.current()) or Lexer.isDigit(self.cursor.current())):
                value += self.cursor.consume()

            keywordType = Lexer.getKeywordType(value)
            if keywordType is not None:
                return Token(keywordType, value)
            # 标识符的情况
            elif len(value) <= MAX_IDENTIFIER_LENGTH:
                return Token(TokenType.IDENTIFIER, value)
            else:
                errmsg = f"Line{self.line}: Error3, Identifier name '{value}' exceeds {MAX_IDENTIFIER_LENGTH} characters"
                return Token(TokenType.ERROR, errmsg)

        if Lexer.isDigit(initial):
            value = initial
            while Lexer.isDigit(self.cursor.current()):
                value += self.cursor.consume()
            return Token(TokenType.CONSTANT, value)

        if initial == '=': return Token(TokenType.EQUAL, '=')

        if initial == '-': return Token(TokenType.SUBTRACT, '-')

        if initial == '*': return Token(TokenType.MULTIPLY, '*')

        if initial == '(': return Token(TokenType.LEFT_PARENTHESES, '(')

        if initial == ')': return Token(TokenType.RIGHT_PARENTHESES, ')')

        if initial == ';': return Token(TokenType.SEMICOLON, ';')

        if initial == '<':
            if self.cursor.current() == '=':
                self.cursor.consume()
                return Token(TokenType.LESS_THAN_OR_EQUAL, '<=')
            elif self.cursor.current() == '>':
                self.cursor.consume()
                return Token(TokenType.NOT_EQUAL, '<>')
            return Token(TokenType.LESS_THAN, '<')

        if initial == '>':
            if self.cursor.current() == '=':
                self.cursor.consume()
```

```
                return Token(TokenType.GREATER_THAN_OR_EQUAL, '>=')
            return Token(TokenType.GREATER_THAN, '>')

        if initial == ':':
            if self.cursor.current() == '=':
                self.cursor.consume()
                return Token(TokenType.ASSIGN, ':=')
            errmsg = f'Line{self.line}: Error2, Misused colon'
            return Token(TokenType.ERROR, errmsg)

        if initial == '\n':
            self.line += 1
            return Token(TokenType.END_OF_LINE, 'EOLN')

        # 否则就是无法识别的字符返回
        errmsg = f"Line{self.line}: Error1, Invalid character '{initial}'"
        return Token(TokenType.ERROR, errmsg)

    def writeTokens(self):
        path = ''
        if __name__ == "__main__":
            path = '../'
        f = open(path+DYD_PATH, mode='w')
        for token in self.tokens:
            f.write(f'{token.value.rjust(16)} {str(token.mytype.value).rjust(2, "0")}\n')

    def writeErrors(self):
        path = ''
        if __name__ == "__main__":
            path = '../'
        f = open(path+ERR_PATH, mode='w')
        for err in self.errors:
            f.write(err+'\n')

    @staticmethod
    def isLetter(value: str):
        return value.isalpha()

    @staticmethod
    def isDigit(value: str):
        return value.isdigit()

    # 判断是关键字还是标志符。如果是标识符就返回 None
    @staticmethod
```

```
    def getKeywordType(value: str):
        v = value.lower()
        if v == 'begin': return TokenType.BEGIN
        elif v == 'end': return TokenType.END
        elif v == 'integer': return TokenType.INTEGER
        elif v == 'if': return TokenType.IF
        elif v == 'then': return TokenType.THEN
        elif v == 'else': return TokenType.ELSE
        elif v == 'function': return TokenType.FUNCTION
        elif v == 'read': return TokenType.READ
        elif v == 'write': return TokenType.WRITE
        return None

    @staticmethod
    def readSource():
        path = ''
        if __name__ == "__main__":
            path = '../'
        f = open(path+SOURCE_PATH)
        return f.read()

if __name__ == "__main__":
    lex = Lexer()
```

Paser.py

```
import sys;sys.path.append("..")
if __name__ != "__main__":
    sys.path.append("src")
from Token import TokenType, Token, Variable, Procedure
from cursor import Cursor
from config import *

class Parser():
    def __init__(self):
        self.line = 1 #  统计第几行的错误
        self.callStack = ["main"] #  从 main 函数开始调用
        self.cursor = Cursor(Parser.readTokens()) #  将 token 作为输入
        self.currentVariableAddress = -1 #  当前还没有读取变量
        self.shouldAddError = True
        self.correctTokens = [] #  二元式表
        self.variables = [] #  变量表
        self.procedures = [] #  过程表
        self.errors = [] #  错误信息文件
        self.parse()

    def parse(self):
        try:
```

```
            self.parseProgram()
            return len(self.errors) == 0
        except Exception as e:
            self.errors.append(str(e) + ' [FATAL]')
            return False
        finally:
            # 写入文件
            self.writeCorrectTokens()
            self.writeVariables()
            self.writeProcedures()
            self.writeErrors()

def parseProgram(self):
    # <程序>→<分程序><EOF>
    self.parseSubprogram()
    self.match(TokenType.END_OF_FILE)

def parseSubprogram(self):
    # <分程序>→begin <说明语句表><执行语句表> end
    self.match(TokenType.BEGIN)
    self.parseDeclarations()
    self.parseExecutions()
    self.match(TokenType.END)

def parseDeclarations(self):
    # <说明语句表>→<说明语句><说明语句表'>
    self.parseDeclaration()
    self.parseDeclarations_()

def parseDeclarations_(self):
    # <说明语句表'>→<说明语句><说明语句表'>|e
    if self.hasType(TokenType.INTEGER):
        self.parseDeclaration()
        self.parseDeclarations_()

def parseDeclaration(self):
    # <说明语句>→integer <说明语句'>;
    self.match(TokenType.INTEGER)
    self.parseDeclaration_()
    self.match(TokenType.SEMICOLON)

def parseDeclaration_(self):
    # <说明语句'>→<变量说明>|<函数说明>
    if self.hasType(TokenType.IDENTIFIER): # 变量声明
```

```
            self.parseVariableDeclaration()
            return
        elif self.hasType(TokenType.FUNCTION):
            self.parseProcedureDeclaration()
            return
        value = self.consumeToken().value
        self.throwError(f"'{value}' is not a valid variable name")

    def parseVariableDeclaration(self):
        # <变量说明>→<变量>
        value = self.match(TokenType.IDENTIFIER).value
        self.registerVariable(value)

    def parseVariable(self):
        # <变量>→<标识符>
        value = self.match(TokenType.IDENTIFIER).value
        if not self.findVariable(value):
            self.addError(f"Undefined variable '{value}'")

    def parseProcedureDeclaration(self):
        # <函数说明>→function <函数名>(<参数>);<函数体>
        self.match(TokenType.FUNCTION)
        self.parseProcedureNameDeclaration()
        self.match(TokenType.LEFT_PARENTHESES)
        self.parseParameterDeclaration()
        #  右括号需要与左括号匹配
        self.match(TokenType.RIGHT_PARENTHESES, "Unmatched '('")
        self.match(TokenType.SEMICOLON)
        self.parseProcedureBody()

    def parseProcedureNameDeclaration(self):
        # <函数名>→<标识符>
        value = self.match(TokenType.IDENTIFIER).value
        self.registerProcedure(value)

    def parseProcedureName(self):
        # # <函数名>→<标识符>
        value = self.match(TokenType.IDENTIFIER).value
        if not self.findProcedure(value):
            self.addError(f"Unidefined procedure '{value}'")

    def parseParameterDeclaration(self):
        # <形参>→<变量>
        value = self.match(TokenType.IDENTIFIER).value
        self.registerParameter(value)

    def parseProcedureBody(self):
```

```
    # <函数体>→begin <说明语句表><执行语句表> end
    self.match(TokenType.BEGIN)
    self.parseDeclarations()
    # 进入执行语句前检查函数的形参是否在说明语句表中定义
    self.checkParameterDeclared()
    self.parseExecutions()
    self.match(TokenType.END)
    # 函过程调用栈的处理
    self.callStack.pop()

def parseExecutions(self):
    # <执行语句表>→<执行语句><执行语句表'>
    self.parseExecution()
    self.parseExecutions_()

def parseExecutions_(self):
    # <执行语句表'>→;<执行语句><执行语句表'>|e
    if self.hasType(TokenType.SEMICOLON):
        self.match(TokenType.SEMICOLON)
        self.parseExecution()
        self.parseExecutions_()

def parseExecution(self):
    # <执行语句>→<读语句>|<写语句>|<赋值语句>|<条件语句>
    if self.hasType(TokenType.READ):
        self.parseRead()
        return
    elif self.hasType(TokenType.WRITE):
        self.parseWrite()
        return
    elif self.hasType(TokenType.IDENTIFIER):
        self.parseAssignment()
        return
    elif self.hasType(TokenType.IF):
        self.parseCondition()
        return
    # 出错处理
    value = self.consumeToken().value
    self.throwError(f"Expect executions, but got '{value}'. Please move all declarations to the beginning of the procedure")

def parseRead(self):
    # <读语句>→read(<变量>)
    self.match(TokenType.READ)
    self.match(TokenType.LEFT_PARENTHESES)
```

```
        self.parseVariable()
        self.match(TokenType.RIGHT_PARENTHESES)

    def parseWrite(self):
        # <写语句>→write(<变量>)
        self.match(TokenType.WRITE)
        self.match(TokenType.LEFT_PARENTHESES)
        self.parseVariable()
        self.match(TokenType.RIGHT_PARENTHESES)

    def parseAssignment(self):
        # <赋值语句>→<变量>:=<算术表达式>|<函数名>:=<算术表达式>
        if self.findVariable(self.cursor.current().value):
            self.parseVariable()
        elif self.findProcedure(self.cursor.current().value):
            self.parseProcedureName()
        else:
            value = self.consumeToken().value
            self.addError(f"Unidefined variable or procedure '{value}'")

        self.match(TokenType.ASSIGN)
        self.parseArithmeticExpression()

    def parseArithmeticExpression(self):
        # <算术表达式>→<项><算术表达式'>
        self.parseTerm()
        self.parseArithmeticExpression_()

    def parseArithmeticExpression_(self):
        # <算术表达式'>→-<项><算术表达式'>|e
        if self.hasType(TokenType.SUBTRACT):
            self.match(TokenType.SUBTRACT)
            self.parseTerm()
            self.parseArithmeticExpression_()

    def parseTerm(self):
        # <项>→<因子><项'>
        self.parseFactor()
        self.parseTerm_()

    def parseTerm_(self):
        # <项'>→*<因子><项'>|e
        if self.hasType(TokenType.MULTIPLY):
            self.match(TokenType.MULTIPLY)
            self.parseFactor()
            self.parseTerm_()
```

```
def parseFactor(self):
    # <因子>→<变量>|<常数>|<函数调用>
    if self.hasType(TokenType.CONSTANT):
        self.match(TokenType.CONSTANT)
        return
    elif self.hasType(TokenType.IDENTIFIER):
        if self.findVariable(self.cursor.current().value):
            self.parseVariable()
            return
        elif self.findProcedure(self.cursor.current().value):
            self.parseProcedureCall()
            return
        value = self.consumeToken().value
        self.throwError(f"Undefined variable or procedure '{value}'")

    value = self.consumeToken().value
    self.throwError(f"Expect variable, procedure or constant, but got '{value}'")

def parseProcedureCall(self):
    # <函数调用>→<函数名>(<算术表达式>)
    self.parseProcedureName()
    self.match(TokenType.LEFT_PARENTHESES)
    self.parseArithmeticExpression()
    self.match(TokenType.RIGHT_PARENTHESES, "Unmatched '('")

def parseCondition(self):
    # <条件语句>→if<条件表达式>then<执行语句>else<执行语句>
    self.match(TokenType.IF)
    self.parseConditionExpression()
    self.match(TokenType.THEN)
    self.parseExecution()
    self.match(TokenType.ELSE)
    self.parseExecution()

def parseConditionExpression(self):
    # <条件表达式>→<算术表达式><关系运算符><算术表达式>
    self.parseArithmeticExpression()
    self.parseOperator()
    self.parseArithmeticExpression()

def parseOperator(self):
    # <关系运算符> →<|<=|>|>=|=|<>
    if self.hasType(TokenType.EQUAL):
        self.match(TokenType.EQUAL)
```

```
            return
        elif self.hasType(TokenType.NOT_EQUAL):
            self.match(TokenType.NOT_EQUAL)
            return
        elif self.hasType(TokenType.LESS_THAN):
            self.match(TokenType.LESS_THAN)
            return
        elif self.hasType(TokenType.LESS_THAN_OR_EQUAL):
            self.match(TokenType.LESS_THAN_OR_EQUAL)
            return
        elif self.hasType(TokenType.GREATER_THAN):
            self.match(TokenType.GREATER_THAN)
            return
        elif self.hasType(TokenType.GREATER_THAN_OR_EQUAL):
            self.match(TokenType.GREATER_THAN_OR_EQUAL)
            return

        value = self.consumeToken().value
        self.addError(f"'{value}' is not a valid operator")

    def registerVariable(self, name: str):
        # 如果是形参，则已经加入过了，将 defined 属性改为 True
        var = self.findParameter(name)
        if var: var.defined = True;return

        # 检查变量重定义的错误
        duplicateVariable = self.findDuplicateVariable(name)
        if duplicateVariable:
            self.addError(f"Variable '{name}' has already been declared")
            return

        # 需要加入的情况，将变量信息加入 variables 列表中，然后
        self.variables.append(Variable(name, self.callStack[-1], 0, "integer", len(self.callStack), self.currentVariableAddress+1))
        self.currentVariableAddress += 1
        # 动态更新相应过程的 first 和 last
        self.updateProcedureVariableAddresses()

    def findDuplicateVariable(self, name: str):
        # 检查是否有在同一个过程中变量多次定义的情况
        for var in self.variables:
            if var.name == name and var.procedure == self.callStack[-1]:
                return var

    def findVariable(self, name: str):
        # 检查是否在该层定义过该变量，使用一个变量前都应该检查一下
        for var in self.variables:
            if var.name == name and var.level == len(self.callStack):
```

```
                return var

    def registerParameter(self, name: str):
        # 一开始直接按惯性思维认为函数可以传多个形参，之后再看文法才发现一个函数只能传一个参数
        duplicateParameter = self.findDuplicateParameter(name)
        if duplicateParameter:
            self.addError(f"Parameter '{name}' has already been declared")
            return

        self.variables.append(Variable(name, self.callStack[-1], 1, "integer", len(self.callStack), self.currentVariableAddress+1,
self.line))
        self.currentVariableAddress += 1
        self.updateProcedureVariableAddresses()

    def findDuplicateParameter(self, name: str):
        for var in self.variables:
            if var.name == name and var.kind == 1 and var.procedure == self.callStack[-1]:
                return var

    def findParameter(self, name: str):
        # 检查当前变量是否是形参
        for var in self.variables:
            if var.name == name and var.kind == 1 and var.level <= len(self.callStack):
                return var

    def registerProcedure(self, name: str):
        # 检查是否存在同级过程重定义的情况。有就报错
        duplicateProcedure = self.findDuplicateProcedure(name)
        if duplicateProcedure:
            self.addError(f"Procedure '{name}' has already been declared")
            return
        # 在进行该操作的时候还没有处理形参，所以 first 和 last 都先初始化为-1。后面动态更新
        self.procedures.append(Procedure(name, "integer", len(self.callStack)+1, -1, -1))
        self.callStack.append(name)

    def findDuplicateProcedure(self, name: str):
        for pro in self.procedures:
            if pro.name == name and pro.level == len(self.callStack)+1:
                return pro

    def findProcedure(self, name: str):
        for pro in self.procedures:
            if pro.name == name and pro.level <= len(self.callStack)+1:
                return pro
```

```
def updateProcedureVariableAddresses(self):
    procedure = self.findProcedure(self.callStack[-1])
    if not procedure: return

    # first 只在第一次初始化
    if procedure.first == -1:
        procedure.first = self.currentVariableAddress
    # last 需要不断更新
    procedure.last = self.currentVariableAddress

def checkParameterDeclared(self):
    # 检查函数的形参是否定义。如果没定义就报错
    for var in self.variables:
        if var.level == len(self.callStack) and var.kind == 1 and var.defined == False:
            errmsg = f"Line {var.line}: parameter '{var.name}' is not declared in the function body"
            self.errors.append(errmsg)



def hasType(self, expectation: TokenType):
    return expectation == self.cursor.current().mytype

def match(self, expectation: TokenType, msg = None):
    # 期望的与实际的符号不同，报错。如果传入了错误信息，则使用传入的
    if not self.hasType(expectation):
        if msg is None:
            msg = f"Expect {Parser.translateToken(expectation)}, but got '{self.cursor.current().value}'"
        self.addError(msg)
    # 即使符号与预期的不同也接收，只是增加报错信息。便于检查出尽可能多的错误
    return self.consumeToken()

def consumeToken(self):
    # self.goToNextLine()
    token = self.cursor.consume()
    # 将正确的符号加入符号表
    self.correctTokens.append(token)
    # 加入后检查一下是否是行尾
    self.goToNextLine()
    return token

def goToNextLine(self):
    # 文件没结束且当前的符号是'\n'才移动到下一行，否则就什么也不做
    while self.cursor.isOpen() and self.hasType(TokenType.END_OF_LINE):
        token = self.cursor.consume() # 移到下一个符号
        self.correctTokens.append(token) # 将'\n'加入符号表
        self.line += 1 # 移动到下一行
        self.shouldAddError = True # 每一行只报一个错
```

```
def addError(self, error: str):
    # 将错误信息添加到列表中，便于之后写入文件
    if not self.shouldAddError: return
    self.shouldAddError = False
    self.errors.append(f'Line {self.line}: {error}')

def throwError(self, error: str):
    raise Exception(f'Line {self.line}: {error}')

def writeCorrectTokens(self):
    path = ''
    if __name__ == "__main__":
        path = '../'
    f = open(path+DYS_PATH, mode='w')
    for token in self.correctTokens:
        f.write(f'{token.value.rjust(16)} {str(token.mytype.value).rjust(2, "0")}\n')

def writeVariables(self):
    path = ''
    if __name__ == "__main__": path = '../'
    f = open(path+VAR_PATH, mode='w')
    for var in self.variables:
        f.write(f'{var.name.rjust(16)} {var.procedure.rjust(16)} {var.kind} {var.type} {var.level} {var.address}\n')

def writeProcedures(self):
    path = ''
    if __name__ == "__main__": path = '../'
    f = open(path+PRO_PATH, mode='w')
    for pro in self.procedures:
        f.write(f'{pro.name.rjust(16)} {pro.type} {pro.level} {pro.first} {pro.last}\n')

def writeErrors(self):
    path = ''
    if __name__ == "__main__": path = '../'
    f = open(path+ERR_PATH, mode='w')
    for err in self.errors:
        f.write(err+'\n')

@staticmethod
def readTokens():
    path = ''
    if __name__ == "__main__": path = '../'
    f = open(path+DYD_PATH)
```

```
            text = f.read().strip()
            tokens = []
            for line in text.split('\n'):
                value, mytype = line.strip().split(" ")
                if (not value) or (not value): continue
                tokens.append(Token(TokenType(int(mytype)), value))
            return tokens

        @staticmethod
        def translateToken(mytype: TokenType):
            # 将 TokenType 与具体的符号联系起来，用于报错信息
            tokenTranslation = {
                TokenType.BEGIN: "'begin'",
                TokenType.END: "'end'",
                TokenType.INTEGER: "'integer'",
                TokenType.IF: "'if'",
                TokenType.THEN: "'then'",
                TokenType.ELSE: "'else'",
                TokenType.FUNCTION: "'function'",
                TokenType.READ: "'read'",
                TokenType.WRITE: "'write'",
                TokenType.IDENTIFIER: "identifier",
                TokenType.CONSTANT: "constant",
                TokenType.EQUAL: "'='",
                TokenType.NOT_EQUAL: "'<>'",
                TokenType.LESS_THAN_OR_EQUAL: "'<='",
                TokenType.LESS_THAN: "'<'",
                TokenType.GREATER_THAN_OR_EQUAL: "'>='",
                TokenType.GREATER_THAN: "'>'",
                TokenType.SUBTRACT: "'-'",
                TokenType.MULTIPLY: "'*'",
                TokenType.ASSIGN: "':='",
                TokenType.LEFT_PARENTHESES: "'('",
                TokenType.RIGHT_PARENTHESES: "')'",
                TokenType.SEMICOLON: "';'",
                TokenType.END_OF_LINE: "EOLN",
                TokenType.END_OF_FILE: "EOF",
            }
            return tokenTranslation[mytype];

if __name__ == "__main__":
    p = Parser()
```

cursor.py

```
class Cursor():
    def __init__(self, src):
        self.position = 0
```

```
        self.source = src

    # 获取接下来应该读取的字符
    def current(self):
        return self.source[self.position]

    # 获取接下来应该读取的字符，并指针前移
    def consume(self):
        current = self.current()
        self.position += 1
        return current

    # 判断文件是否读取完
    def isOpen(self):
        return self.position < len(self.source)

    def __repr__(self):
        return f'position: {self.position}\nsource:\n{self.source}'
```

Token.py

```
from enum import Enum

class TokenType(Enum):
    BEGIN = 1
    END = 2
    INTEGER = 3
    IF = 4
    THEN = 5
    ELSE = 6
    FUNCTION = 7
    READ = 8
    WRITE = 9
    IDENTIFIER = 10
    CONSTANT = 11
    EQUAL = 12
    NOT_EQUAL = 13
    LESS_THAN_OR_EQUAL = 14
    LESS_THAN = 15
    GREATER_THAN_OR_EQUAL = 16
    GREATER_THAN = 17
    SUBTRACT = 18
    MULTIPLY = 19
    ASSIGN = 20
    LEFT_PARENTHESES = 21
```

```
    RIGHT_PARENTHESES = 22
    SEMICOLON = 23
    END_OF_LINE = 24
    END_OF_FILE = 25
    ERROR = 26

class Token(object):
    def __init__(self, mytype: TokenType, value: str):
        self.mytype = mytype
        self.value = value

    def __repr__(self):
        return f'[mytype: {self.mytype}, value: {self.value}]'

class Variable():
    def __init__(self, name, procedure, kind, mytype, level, address, line = -1, defined = False):
        self.name = name
        self.procedure = procedure
        self.kind = kind
        self.type = mytype
        self.level = level
        self.address = address
        self.defined = defined
        self.line = line

    def __repr__(self) -> str:
        return '{'+f'name: {self.name}, procedure: {self.procedure}, kind: {self.kind}, type: {self.type}, level: {self.level}, address: {self.address}'+'}'

class Procedure():
    def __init__(self, name, mytype, level, first, last):
        self.name = name
        self.type = mytype
        self.level = level
        self.first = first
        self.last = last

    def __repr__(self) -> str:
        return '{'+f'name: {self.name}, type: {self.type}, level: {self.level}, first: {self.first}, last: {self.last}'+'}'

if __name__ == "__main__":
    t = Token(TokenType.BEGIN, '+')
    print(t.mytype.value[0])
```

config.py

```
SOURCE_PATH = "./input/source.pas";
ERR_PATH = "output/source.err";
```

```
DYD_PATH = "output/source.dyd";
DYS_PATH = "output/source.dys";
VAR_PATH = "output/source.var";
PRO_PATH = "output/source.pro";
```