# 语法/语义分析设计报告

目录

[语法/语义分析设计报告 1](#_Toc24674)

[概述 1](#_Toc29308)

[设计目标 1](#_Toc25919)

[核心组件 2](#_Toc22963)

[1. 产生式（Production Rules） 2](#_Toc32336)

[2. 动作函数（Action Functions） 3](#_Toc4600)

[3. 错误处理规则 3](#_Toc31798)

[实现细节 3](#_Toc4304)

[1. 集成词法分析器 3](#_Toc20532)

[2. 定义AST 3](#_Toc29917)

[3. 构建AST 8](#_Toc3862)

[4. 错误恢复 9](#_Toc21955)

[测试与验证 9](#_Toc30693)

## 概述

YACC（Yet Another Compiler Compiler）是一个基于Python的解析器生成器，它使用LALR解析算法来构建语法分析器。本报告将详细介绍编译原理课设语法分析器（YACC）的设计目标、实现细节以及如何与词法分析器（Lexer）协同工作，构建语法树。

## 设计目标

- 模块化：将语法分析器的设计分解为独立的模块，以便于管理和维护。

- 可扩展性：设计易于扩展的架构，以便未来可以添加新的语法规则和特性。

- 错误处理：提供有效的错误检测和报告机制，以提高解析过程的健壮性。

- 与Lexer的集成：确保YACC能够与已有的词法分析器无缝集成，共享标记（tokens）和错 误信息。

-生成AST：在语法分过程中建立抽象语法树以便在后续将代码转换成C语言

## 核心组件

### 1. 产生式（Production Rules）

YACC通过一套产生式规则来定义语言的语法结构。每个规则描述了如何将一系列的非终结符和终结符组合成更复杂的结构。例如，一个简单的产生式规则可以是：

def p\_expression\_term(p):

    'expression : term'

    p[0] = p[1]  # 将解析的结果赋值给最左边的非终结符

为了是的pascal语言的文法符合yacc的产生式规则我们首先将指导书上的PASCAL文法进行改写，使其成为由非终结符和tokens组成的文法。比如如下文法的yacc产生式规则：



 def p\_programstruct(p):

            'programstruct : program\_head SEMICOLON program\_body POINT'

 def p\_program\_head(p):

            'program\_head : PROGRAM ID LPAREN idlist RPAREN '

  def p\_program\_head\_jusi\_id(p):

            'program\_head :  PROGRAM ID'

还有为了弥补pascal文法中的一些问题或者为了能与词法分析器衔接，对文法进行了如下更改/添加：

***const\_value →* ADDOPNUM | NUM | ADDOP *DIGITS* | DIGITS | LETTERS**

**factor → NUM | DIGITS**

**expression → simple\_expression relop simple\_expression**

**relop → RELOP | EQUAL**

**( LETTERS → ′ letter ′ )**

**func\_id -> ID**

### 2. 动作函数（Action Functions）

与每个产生式规则相关联的是一个动作函数，它定义了在应用该规则时应该执行的操作。动作函数通常用于构建抽象语法树（AST）节点。

比如：

  def p\_expression\_plus(p):

    'expression : term PLUS expression'

    # p[0] 将存储整个表达式的值，p[1] 是term的值，p[3] 是expression的值

    p[0] = Expression('Plus', p[1], p[3])

### 3. 错误处理规则

为了处理解析过程中可能出现的错误，YACC提供了一个特殊的产生式规则`p\_error`，用于定义当输入不符合任何规则时应该采取的行动。我们对p\_error的定义如下，这样写的目的是便于排错。

 def p\_error(p):

       raise yacc.YaccError("语法错误在第 %d 行，第 %d 列 %s" % (p.lineno, p.lexpos,p))

## 实现细节

### 1. 集成词法分析器

YACC需要与词法分析器（Lexer）紧密集成，以便使用由Lexer生成的标记（tokens）。这通常通过PLY工具集来实现，其中Lexer和Parser可以共享一个公共的数据结构。我们先引入词法分析器和里面的tokens，AST模块。

把pascal文法改写成yacc规则产生式，为每个产生式定义动作用于建立符号表和语法树。

### 2. 定义AST

为了方便后面的代码生成我们目前规定的ast结构如下，其中也包含了符号表结构，用于确定代码生成过程中变量的作用域等。这不是最终的ast结构，我们会随着语法分析过程的推进可能对部分数据的格式进行更改。

这是AST语法树的结构图：[abdusalam1.github.io/pyUML.svg](https://abdusalam1.github.io/pyUML.svg)

#符号表

class SymbolTable:

    def \_\_init\_\_(self):

        self.area=''

        self.Constants={}#常量 常量名：{Constant}

        self.Variants={} #变量 变量名：{Type}

        self.subFuncs={} #函数 函数名：{SymbolTable}

#语法树

class Program:

    """

    主程序类

    """

    def \_\_init\_\_(self,a,b,c,d):

        self.programId:str = a#程序标题声明名

        self.line:int = b      #程序标题行号

        self.paraDict:dict = c #参数标识符:行号

        self.subProgram:SubProgram =d

class SubProgram:

    """

    子程序类

    """

    def \_\_init\_\_(self,a,b,c,d):

        self.constList: dict = a if a else {}  # 常量定义列表

        self.varList: dict = b if b else {}    # 变量定义列表

        self.subDefList: dict = c if c else {}  # 子程序和子函数

        self.block: Compound = Compound(d)  # 主程序体

class Constant:

    """

    常量定义类

    """

    def \_\_init\_\_(self,a,b,c,d):

        self.constId:str = a  #标识符

        self.line:int = b       #行号

        self.type:str = c      #类型int,char,real,string（这个string指的是常量标识符的ID，因为可能由其他量赋值）

        self.Value= d           #值

class Variant:

    """

    变量定义类

    """

    def \_\_init\_\_(self, idlist,type):

        self.idlist: dict = idlist  # 标识符名称

        self.type: Type = type  # 变量类型

class Type:

    """

    变量类型类

    """

    def \_\_init\_\_(self,a='',b=0,c=False,d=0,e=0):

        self.type:str = a       #基本类型，int、char、real、boolean

        self.line:int = b       #行号

        self.arrFlag:bool= c    #0不是数组，1是数组

        self.lowerBound:list[int] = d  #数组下界（n维）

        self.upperBound:list[int] = e  #数组上界

class FucDefn:

    """

    过程与函数定义类

    """

    def \_\_init\_\_(self,name,lineno,formal,types,con,var,block):

        self.fucId:str = name      #名称

        self.line:int = lineno        #行号

        self.formalParaList:dict = formal if formal else {}#形参字典 数字：FormalParameter

        self.type:str = types       #如果type是空串，则为过程，否则为函数,取值为"integer","real","boolean","char"四种

        self.constList:dict = con if con else {} #常数定义列表

        self.varList:dict = var if var else {} #变量定义列表

        self.block:Compound = Compound(block) #程序体

class FormalParameter:

    """

    形参类

    """

    def \_\_init\_\_(self):

        self.paraId:list  #标识符

        self.line:int    #行号

        self.type:str  #基本类型，int、char、real、boolean

        self.flag:bool = False#false代表传值调用，true代表引用调用

class Expression:

    """

    表达式类

    """

    def \_\_init\_\_(self):

        self.type:str = ''

        """

        表达式类型,"var"表示变量,"int"表示整数,"real"表示浮点数,"function"表示函数调用,

        "compound"表示复合表达式,compound有普通的二目运算符

        还有minus、not、bracket等单目运算符

        """

        self.line:int = 0

        self.varRef:VarReference = VarReference()

        self.intNum:int = 0#整数

        self.realNum:float = 0#浮点

        self.strOfNum:str = ''#字符串的数

        self.charVal:str = ''#字符的值

        self.fucCall:FucCall = FucCall()

        self.operation:str = ''#复合表达式

        self.opType:str = ''#操作符类型

        self.subE1:Expression = Expression()#子式1

        self.subE2:Expression = Expression()#子式2

class FucCall:

    """

    函数调用

    """

    def \_\_init\_\_(self):

        self.fucId:str = ''#函数名称

        self.line:int = 0

        self.actParaList:list[Expression] = Expression()#参数列表

class VarReference:

    """

    变量引用

    """

    def \_\_init\_\_(self):

        self.varId:str = ''#变量名

        self.line:int = 0

        self.expList:list[Expression] = []#

        self.flag:bool = False #0表示非数组,1表示数组

class Statement:

    """

    语句类

    """

    def \_\_init\_\_(self):

        self.line:int = 0#行号

        self.type:str = ''#类型"compound","repeat","while","for","if","assign","procedure"

        self.stateType:str = ''#区别于type，取值为"void"或"error"

class Assign(Statement):

    """

    赋值

    """

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.varRef:VarReference = VarReference()#左值变量

        self.exp:Expression = Expression()#右边表达式

class If(Statement):

    """

    条件语句

    """

    def \_\_init\_\_(self):

        self.condition:Expression =  Expression()#条件表达式

        self.then:Statement = Statement()#then语句

        self.els:Statement = Statement()#else语句

class While(Statement):

    """

    循环语句

    """

    def \_\_init\_\_(self):

        self.condition:Expression =  Expression()#条件表达式

        self.do:Statement = Statement()#循环体

class For(Statement):

    """

    for循环语句

    """

    def \_\_init\_\_(self):

        self.id:str = ''#循环变量

        self.state:Expression = Expression()#起始值

        self.end:Expression = Expression()#结束值

        self.do:Statement = Statement()#执行语句

class Repeat(Statement):

    """

    repeat语句

    """

    def \_\_init\_\_(self):

        self.condition:Expression = Expression()#条件

        self.do:Statement = Statement()

class Compound(Statement):

    """

    复合语句类

    """

    def \_\_init\_\_(self,block=[]):

        self.statements:list[Statement] = block#语句列表

class ProcCall(Statement):

    """

    过程调用

    """

    def \_\_init\_\_(self):

        self.procId:str = ''#名称

        self.actParaList:list[Expression] = []#实参

### 构建AST

在YACC中，每个产生式规则都可以有一个与之关联的动作函数，该函数在规则被应用时执行，这些动作的定义在产生式规则下面实现。这些动作函数负责创建和返回AST节点，这些节点随后可以用于语义分析和代码生成。

这是对头的两个文法规定动作后的代码：



        def p\_programstruct(p):

            'programstruct : program\_head SEMICOLON program\_body POINT'

            # 语法树节点信息，记录上述产生式下非终结符的节点信息，下同

            p[0] = {

                        "length": len(p),

                        "type": "programstruct",

                         "program\_head": p[1],

                         "program\_body": p[3],

                        "program": Program(p[1]["id"],p[1]["line"],p[1]["idlist"],p[3])

                    }

        def p\_program\_head(p):

            'program\_head : PROGRAM ID LPAREN idlist RPAREN '

            p[0] = {

             "id" :p[2],

            "line" :p.lineno(1),

            "idlist":p[4]

             }

        def p\_program\_head\_jusi\_id(p):

            'program\_head :  PROGRAM ID'

            p[0] = {

             "id" :p[2],

            "line" :p.lineno(1),

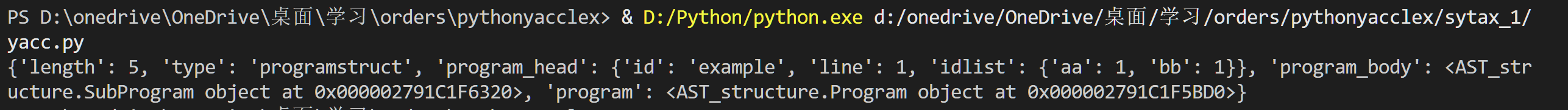
            "idlist":None

             }

通过自底向上的分析可以看看出规则动作首先会为program\_head建立一个字典，字典里的值结合program\_body用于初始化一个Program（）实例，此实例也是最终ast的根节点。

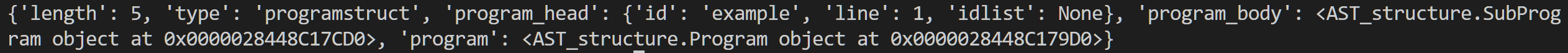
比如对这样一个program\_head有这样的result：

program example(aa,bb);



对这样的有：

program example;



### 4. 错误恢复

为了提高解析器的健壮性，YACC设计了错误恢复机制。当检测到语法错误时，解析器会尝试跳过错误并继续解析。这通常通过回溯和跳过一定数量的标记来实现。

目前在语法分析里实现的错误检测策略有：

1）yacc的检错规则。当遇到一个无法匹配的文法规则时抛出错误信息，告诉用户出错的地方：

  def p\_error(p):

            raise yacc.YaccError("语法错误在第 %d 行，第 %d 列 %s" % (p.lineno, p.lexpos,p))

1. id重名检测。当检测到标识符重名时定义一个“变量重复”的错误类型并输出位置

        def p\_idlist(p):

            '''idlist : ID

                    | idlist COM ID'''

            if len(p)==2:

                p[0]={p[1]:p.lineno(1)}

            else:

                if p[1].get(p[3])== None : #判断重复

                    p[1][p[3]]=p.lineno(3)

                    p[0]=p[1]

                else:

                    self.error.append({

                        'type':'变量重复',

                        'line':p.lineno(3)

                    })

## 测试与验证

YACC的测试和验证是通过运行一组预定义的测试用例来完成的。这些测试用例覆盖了各种合法和非法的输入情况，以确保解析器能够正确处理所有可能的场景。

目前已完成了对基础的pascal程序的测试：

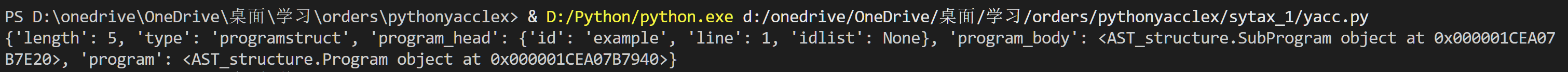
program example;

var x, y: integer;

begin

read(x, y);

end.



注释：

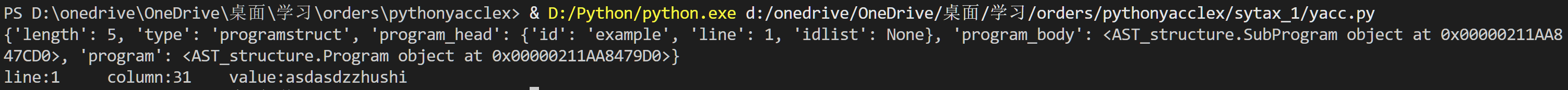
program example; {asdasdz注释}

var x, y: integer;

begin

read(x, y);

end.



非法字符：

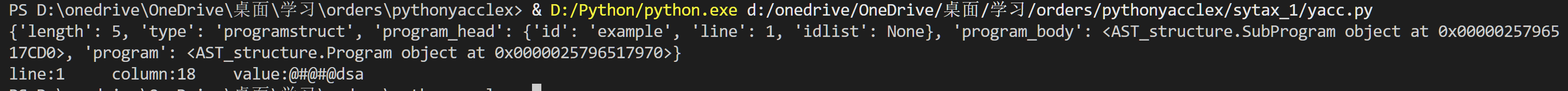
program example; {@#@#@dsa

var x, y: integer;

begin

read(x, y);

end.



program example();

var x, y: integer;

begin

read(x, y);

end.

