苏州大学实验报告

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 院系 | 计算机学院 | | 年级专业 | | 21计科 | | 姓名 | 方浩楠 | 学号 | 2127405048 |
| 课程名称 | | 编译原理课程实践 | | | | | | | 成绩 |  |
| 指导教师 | | 王中卿 | | 同组实验者 | | 无 | | 实验日期 | 2023.12.18 | |

|  |  |
| --- | --- |
| 实 验 名 称 | 实验 |

1. 实验目的

目的概述

本实验旨在通过编程实现一个简易Python解释器，特别着重于类的解析及其内部成员（变量和函数）的语法制导翻译。通过这个实验，学生将深入理解面向对象编程中类的概念、类的内部结构以及类如何在编程语言中被解析和翻译的过程。

具体目标

理解类的结构：深入理解类的定义、类成员变量和方法的概念。

语法分析和词法分析：掌握如何使用词法分析器和语法分析器处理类定义及其成员。

语法制导翻译技术：学习并实现语法制导翻译技术，将类结构从源代码级转换为可执行代码。

编码实践：提高Python编程能力，特别是在处理复杂数据结构和算法方面。

1. 实验内容

实验环境和工具

编程语言：Python 3.9.6

主要工具：

**Package Version**

**---------- -------**

**pip 23.3.2**

**ply 3.11**

**setuptools 68.2.0**

**wheel 0.41.2**

开发环境：任意支持Python的IDE,本人使用的是PyCharm Professional Edition 2023.3.1

类的解析

实现词法分析器（py\_lex.py）以识别类定义的关键词和符号。

在语法分析器（py\_yacc.py）中定义类的语法规则，如何解析类定义、类成员变量和方法。

类中变量的翻译

设计数据结构（在node.py中）来表示类中的变量。

在翻译阶段（translation.py），实现对类变量的处理逻辑，如变量声明、初始化和访问。

类中函数的翻译

扩展node.py中的数据结构以包含类中的函数定义。

在translation.py中实现对类中函数的翻译，包括函数调用、参数传递和作用域管理。

实验要点

确保词法分析器和语法分析器能够准确解析类定义及其成员。

关注类成员（变量和函数）的作用域和可见性。

考虑类的继承和多态性在翻译过程中的处理。

1. 实验步骤和结果

本次实验将之前的部分代码使用typing模块重写,同时加上了Google风格的docstring来增加代码的可读性

项目结构图:

**基于PLY的Python解析(4)**

**├── python\_parser**

**│   ├── main.py # 项目的入口文件。处理命令行参数，读取Python脚本文件**

**│   ├── node.py # 定义了用于构建抽象语法树（AST）的节点类**

**│   ├── parser.out**

**│   ├── parsetab.py**

**│   ├── py\_lex.py # 定义了如何将Python代码拆分成一个个的词法单元（tokens）**

**│   ├── py\_yacc.py # 定义了Python语言的语法规则**

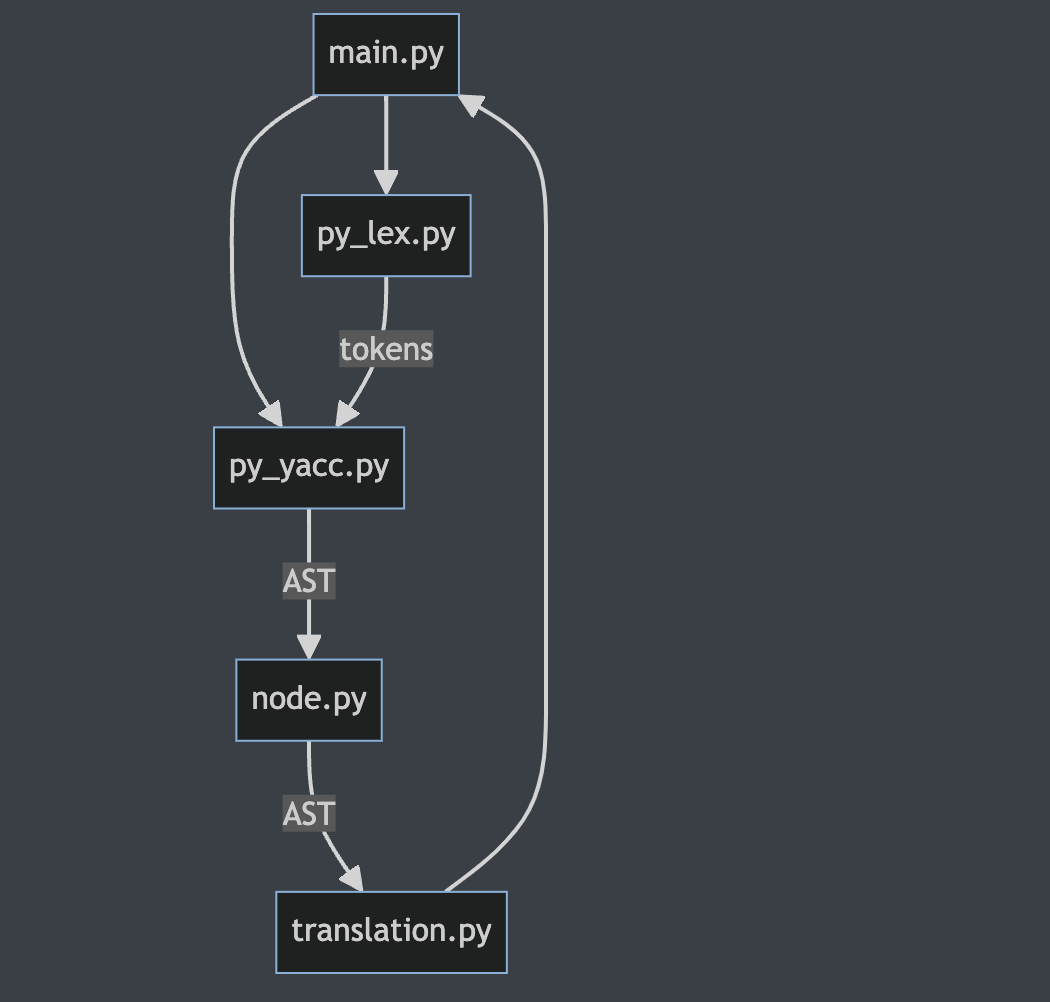
**│   ├── stu.py # 示例Python脚本，用于测试解释器**

**│   └── translation.py # 负责将抽象语法树转换成可执行的代码**

**├── readme.md**

**└── requirements.txt**

**项目流程图**

****

**流程图分析:**

**流程图分析**

**main.py (主入口)**

**main.py 作为整个程序的入口点，它首先调用 py\_lex.py 和 py\_yacc.py。**

**它负责接收输入文件（Python代码），并控制整个程序的流程。**

**py\_lex.py (词法分析器)**

**py\_lex.py 负责将输入的Python代码分解为词法单元（tokens），例如关键字、标识符、字面量等。**

**词法分析的结果（tokens）随后传递给 py\_yacc.py。**

**py\_yacc.py (语法分析器)**

**py\_yacc.py 接收来自 py\_lex.py 的tokens，进行语法分析。**

**它根据定义的语法规则，将这些tokens组装成一个抽象语法树（AST）。**

**生成的AST随后传递给 node.py。**

**node.py (节点定义)**

**node.py 定义了AST的节点结构。**

**它接收来自 py\_yacc.py 的AST，并可能对其进行进一步的处理或补充，使其适用于翻译过程。**

**translation.py (语法制导翻译)**

**translation.py 接收从 node.py 来的AST。**

**它负责将AST转换为可执行代码。这个过程涉及到遍历AST并执行相应的操作。**

**回到 main.py**

**完成翻译后，控制权回到 main.py，可能包括执行转换后的代码，或者进行其他后续操作，如打印输出等。**

**数据流向和处理**

**Tokens的生成和传递：**

**从 py\_lex.py 到 py\_yacc.py：py\_lex.py 生成tokens，传递给 py\_yacc.py 进行语法分析。**

**AST的构建和应用：**

**从 py\_yacc.py 到 node.py：py\_yacc.py 构建AST，node.py 根据需要处理或补充这个AST。**

**从 node.py 到 translation.py：处理后的AST被传递到 translation.py 进行语法制导翻译。**

translate函数的工作流程:

图示

描述已自动生成

以下是对每个主要步骤的具体分析：

开始:

函数开始执行，首先检查 break\_flag 或 return\_flag。

如果其中一个标志为真，则立即返回 return\_value。

检查是否为NonTerminal节点:

检查 \_tree 是否是一个 NonTerminal 节点。

如果不是，则流程结束。

如果是，根据 \_tree 节点的类型执行不同的操作。

节点类型判断:

函数会根据节点的类型（如 Function, Class, If, Else, While, For, Break, Return）来决定下一步操作。

处理不同类型的节点:

对于每种类型的节点，函数会执行相应的操作：

Function：处理函数定义节点，创建新的 Function 对象，并将其存储在变量表中。

Class：处理类定义节点，创建新的 Class 对象，并将其存储在变量表中。

If/Else：处理条件判断节点，根据条件值执行相应的语句块。

While：处理循环节点，只要条件为真，就不断循环执行语句块。

For：处理 for 循环节点，根据初始化、条件和更新表达式执行循环体。

Break：设置 break\_flag，表示中断循环。

Return：处理返回语句节点，设置 return\_value 和 return\_flag。

递归处理子节点:

如果节点是其他类型，函数将递归地处理其所有子节点。

执行翻译操作:

根据当前节点和其子节点的信息，执行相应的翻译操作。

结束:

最后，函数返回 return\_value。

node.py模块作用:

该模块定义了用于构建抽象语法树（AST）的各种节点类型。

主要包含的类有：

- \_node：所有节点的基类，提供基本的节点数据结构。

- NonTerminal：非终结符节点，用于表示具有特定类型和可选值的非终结符。

- Variable：左值节点，表示引用的变量，包含类型和标识符。

- Number：数字节点，直接包含一个数字值。

- ID：标识符节点，包含标识符的名称和值。

- Terminal：终结符节点，用于表示除标识符外的其他终结符，包含其文本内容。

这些节点类型在解析Python代码并构建其AST时发挥核心作用。

py\_lex.py模块中定义的关键词

reserved\_words = {

'print': 'PRINT',

'if': 'IF',

'elif': 'ELIF',

'else': 'ELSE',

'for': 'FOR',

'while': 'WHILE',

'len': 'LEN',

'break': 'BREAK',

'and': 'AND',

'or': 'OR',

'def': 'DEF',

'return': 'RETURN',

'class': 'CLASS',

}

tokens = ['NUMBER', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN', 'RPAREN', 'LBRACKET', 'RBRACKET', 'ASSIGN',

'LBRACE', 'RBRACE', 'SEMICOLON', 'COMMA', 'DPLUS', 'DMINUS', 'ID', 'EDIVIDE', 'MINEQUAL', 'PLUSEQUAL',

'LT', 'LE', 'GT', 'GE', 'EQ', 'NE', 'DOT', 'STRING', ] + list(reserved\_words.values())

t\_PLUSEQUAL = r'\+='

t\_MINEQUAL = r'-='

t\_PLUS = r'\+'

t\_MINUS = r'-'

t\_TIMES = r'\\*'

t\_EDIVIDE = r'//'

t\_DIVIDE = r'/'

t\_LPAREN = r'\('

t\_RPAREN = r'\)'

t\_LBRACE = r'\{'

t\_RBRACE = r'\}'

t\_LBRACKET = r'\['

t\_RBRACKET = r'\]'

t\_ASSIGN = r'='

t\_DPLUS = r'\+\+'

t\_DMINUS = r'--'

t\_COMMA = r','

t\_SEMICOLON = r';'

t\_LT = r'<'

t\_LE = r'<='

t\_GT = r'>'

t\_GE = r'>='

t\_EQ = r'=='

t\_NE = r'!='

t\_DOT = r'\.'

该解释器中用到的语法规则:

**Grammar**

**Rule 0 S' -> program**

**Rule 1 program -> statements**

**Rule 2 statements -> statements statement**

**Rule 3 statements -> statement**

**Rule 4 statement -> assignment**

**Rule 5 statement -> expr**

**Rule 6 statement -> print**

**Rule 7 statement -> if**

**Rule 8 statement -> while**

**Rule 9 statement -> for**

**Rule 10 statement -> break**

**Rule 11 statement -> function**

**Rule 12 statement -> class**

**Rule 13 statement -> return**

**Rule 14 assignment -> variable ASSIGN expr**

**Rule 15 assignment -> variable MINEQUAL expr**

**Rule 16 assignment -> variable PLUSEQUAL expr**

**Rule 17 assignment -> variable DPLUS**

**Rule 18 assignment -> variable DMINUS**

**Rule 19 variable -> variable LBRACKET expr RBRACKET**

**Rule 20 variable -> ID DOT ID**

**Rule 21 variable -> ID**

**Rule 22 expr -> expr PLUS term**

**Rule 23 expr -> expr MINUS term**

**Rule 24 expr -> term**

**Rule 25 expr -> string**

**Rule 26 expr -> array**

**Rule 27 term -> term TIMES factor**

**Rule 28 term -> term DIVIDE factor**

**Rule 29 term -> term EDIVIDE factor**

**Rule 30 term -> factor**

**Rule 31 factor -> variable**

**Rule 32 factor -> NUMBER**

**Rule 33 factor -> len**

**Rule 34 factor -> call**

**Rule 35 factor -> LPAREN expr RPAREN**

**Rule 36 exprs -> exprs COMMA expr**

**Rule 37 exprs -> expr**

**Rule 38 len -> LEN LPAREN variable RPAREN**

**Rule 39 print -> PRINT LPAREN exprs RPAREN**

**Rule 40 print -> PRINT LPAREN RPAREN**

**Rule 41 array -> LBRACKET exprs RBRACKET**

**Rule 42 array -> LBRACKET RBRACKET**

**Rule 43 condition -> condition OR join**

**Rule 44 condition -> join**

**Rule 45 join -> join AND equality**

**Rule 46 join -> equality**

**Rule 47 equality -> equality EQ rel**

**Rule 48 equality -> equality NE rel**

**Rule 49 equality -> rel**

**Rule 50 rel -> expr LT expr**

**Rule 51 rel -> expr LE expr**

**Rule 52 rel -> expr GT expr**

**Rule 53 rel -> expr GE expr**

**Rule 54 rel -> expr**

**Rule 55 if -> IF LPAREN condition RPAREN LBRACE statements RBRACE**

**Rule 56 if -> IF LPAREN condition RPAREN LBRACE statements RBRACE else**

**Rule 57 else -> ELIF LPAREN condition RPAREN LBRACE statements RBRACE**

**Rule 58 else -> ELIF LPAREN condition RPAREN LBRACE statements RBRACE else**

**Rule 59 else -> ELSE LBRACE statements RBRACE**

**Rule 60 while -> WHILE LPAREN condition RPAREN LBRACE statements RBRACE**

**Rule 61 for -> FOR LPAREN assignment SEMICOLON condition SEMICOLON assignment RPAREN LBRACE statements RBRACE**

**Rule 62 break -> BREAK**

**Rule 63 function -> DEF ID LPAREN args RPAREN LBRACE statements RBRACE**

**Rule 64 function -> DEF ID LPAREN RPAREN LBRACE statements RBRACE**

**Rule 65 args -> args COMMA ID**

**Rule 66 args -> ID**

**Rule 67 call -> ID LPAREN exprs RPAREN**

**Rule 68 call -> ID LPAREN RPAREN**

**Rule 69 call -> ID DOT ID LPAREN exprs RPAREN**

**Rule 70 call -> ID DOT ID LPAREN RPAREN**

**Rule 71 return -> RETURN**

**Rule 72 return -> RETURN exprs**

**Rule 73 class -> CLASS ID LBRACE functions RBRACE**

**Rule 74 functions -> functions function**

**Rule 75 functions -> function**

**Rule 76 string -> STRING**

translation.py中的一些类的接口:

**class Function:**

**"""**

**表示一个函数的类。**

**此类用于封装函数的定义，包括函数名、参数名列表和函数体。**

**Attributes:**

**name (str): 函数的名称。**

**arg\_names (List[str]): 函数的参数名列表。**

**body (\_node): 函数的主体，是一个节点对象。**

**"""**

**class Class:**

**"""**

**表示一个Python类的对象。**

**此类负责解析并存储类定义中的函数。它使用 'Translator' 来翻译类中的函数，**

**并将翻译后的函数存储在自身的 'functions' 属性中。**

**Attributes:**

**name (str): 类的名称。**

**functions (dict): 一个包含类中函数的字典，其中键是函数名称，值是函数的表示。**

**"""**

**class PyObject:**

**"""**

**代表一个Python对象。**

**此类用于模拟Python中的对象，包括类属性和构造函数的执行。**

**Attributes:**

**cls (Any): 该对象所属的类。**

**props (Dict[str, Any]): 存储对象属性的字典。**

**"""**

项目运行配置:

电脑萤幕的截图

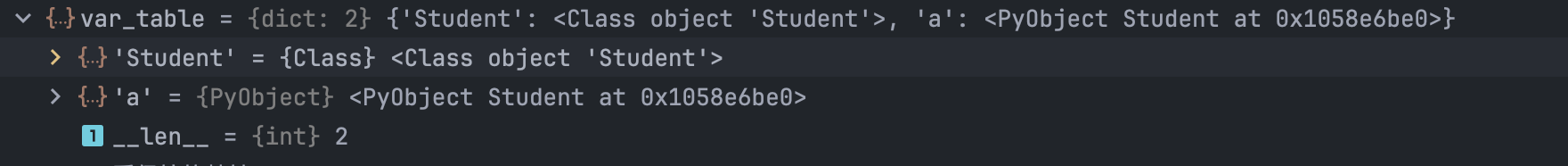
描述已自动生成

该项目运行配置的xml文件:

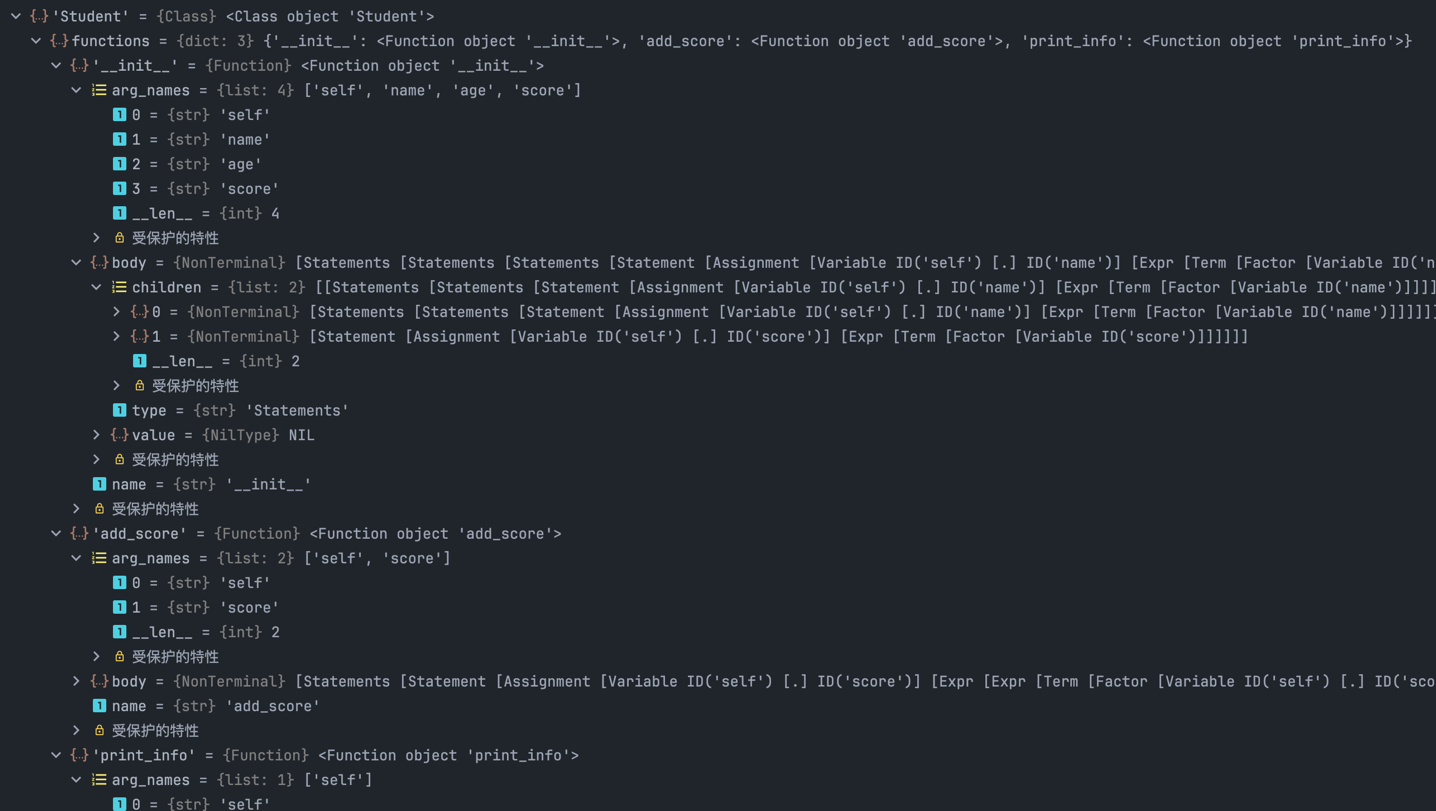
**<component name="ProjectRunConfigurationManager">  
 <configuration default="false" name="main" type="PythonConfigurationType" factoryName="Python">  
 <module name="experiment13" />  
 <option name="ENV\_FILES" value="" />  
 <option name="INTERPRETER\_OPTIONS" value="" />  
 <option name="PARENT\_ENVS" value="true" />  
 <envs>  
 <env name="PYTHONUNBUFFERED" value="1" />  
 </envs>  
 <option name="SDK\_HOME" value="" />  
 <option name="WORKING\_DIRECTORY" value="$PROJECT\_DIR$/python\_parser" />  
 <option name="IS\_MODULE\_SDK" value="true" />  
 <option name="ADD\_CONTENT\_ROOTS" value="true" />  
 <option name="ADD\_SOURCE\_ROOTS" value="true" />  
 <EXTENSION ID="PythonCoverageRunConfigurationExtension" runner="coverage.py" />  
 <option name="SCRIPT\_NAME" value="$PROJECT\_DIR$/python\_parser/main.py" />  
 <option name="PARAMETERS" value="stu.py" />  
 <option name="SHOW\_COMMAND\_LINE" value="false" />  
 <option name="EMULATE\_TERMINAL" value="false" />  
 <option name="MODULE\_MODE" value="false" />  
 <option name="REDIRECT\_INPUT" value="false" />  
 <option name="INPUT\_FILE" value="" />  
 <method v="2" />  
 </configuration>  
</component>**

项目运行后:

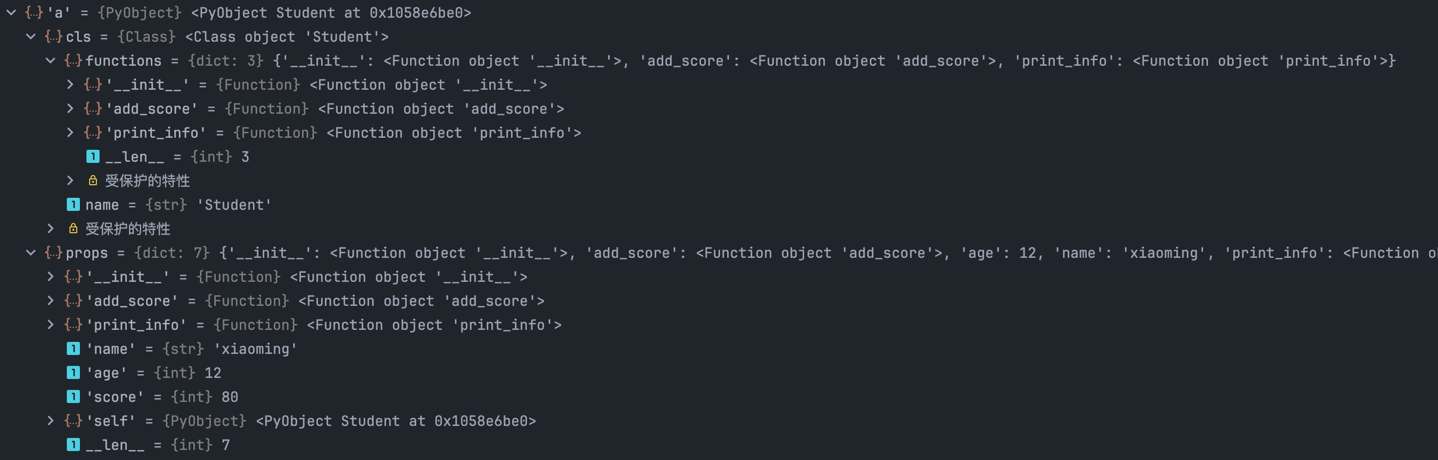
变量表中存储的内容:



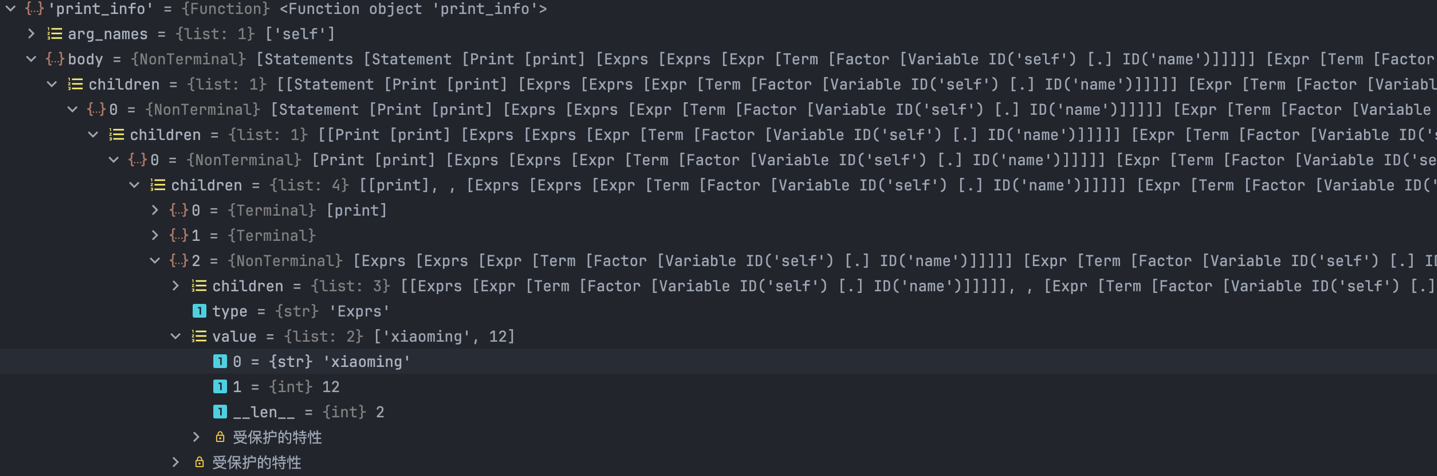
其中,Class Student中的内容:



其中,a中的内容:



该class中的self变量的值:



图中可以看到,该class中的非终结符中的两个值分别为”xiaoming”和12,这两个值分别对应着self.name和self.age

将该python项目打包为Linux可执行文件后,在zsh终端运行该项目:

**./dist/main/main python\_parser/stu.py**

**执行结果:**

**文本

描述已自动生成**

**可以看到stu.py的内容被成功解析**

1. 实验总结

成果

成功实现了一个能够解析和翻译Python类的简易解释器。

深入理解了类的内部结构以及类在编程语言中的处理方式。

遇到的挑战

类的作用域和可见性规则在实现时比较复杂。

保持代码的模块化和清晰结构在项目扩展时具有一定的挑战。

收获与反思

对面向对象编程和编译原理有了更深刻的理解。

实践中学习到的编码技巧和问题解决方法对未来的学习和研究有极大帮助。

未来的工作可以更加注重代码的可维护性和模块化设计。

展望

进一步扩展解释器，支持更多高级特性，如异常处理、装饰器等。

将这个项目作为学习编译原理和Python编程的基础，继续深入研究相关领域。