# mini-Python3解释器说明文档

## 词法分析器部分

### 概述

词法分析器是将字符序列转换为token序列的工具。本词法分析器（Scanner类）基于python语言所使用的token类型建立符号表，读取并缓存输入字符流，从中识别出语素，最后生成不同类型的标记，并有简单的报错功能和分析缩进层级功能。

### 实现的功能

这颗抽象语法树能进行如下操作：

* 通过文件输入流或标准输入流读取字符序列
* 对缓存的序列进行逐行分析
* 区分标识符、数值、字符串、保留字以及其他符号
* 过滤无效的行以及注释
* 分析缩进关系

### API

对词法分析器将基于符号表进行分析，符号表通过枚举类型建立（enum Symbol），包括了以下几类符号：

* 无语义的类型（nul）
* 标识符（ident）
* 整数（number\_int）
* 浮点数（number\_double）
* 字符串（str）
* 算术及位移运算符（plus等）
* py保留字（false等）
* 其他字符（tab等）

共68个。词法分析器在初始化时将在内部创建字符串型保留字列表和符号值列表，以便分析需求。

词法分析器内部存储的变量包括：

* 输入的字符流（stringstream in）
* 单字符的符号表指针（Symbol\* ssym）
* 保留字的符号表指针（Symbol\* wsym）
* 字符串型保留字列表指针（string\* word）
* 当前的单词符号（Symbol sym）
* 当前字符在行中的位置（int cc）
* 当前分析的行（string line）
* 当前行的长度（int ll）
* 当前读入的单个字符（char ch）
* 判断分析结束的flag（bool is\_empty）
* 缩进记数栈（vector\<int> tab\_counter）
* 缩进回退数（int tab\_back）

以上变量均为private权限的分析数据，对外不可见。public权限的外部接口如下所示：

### 标识符名字（string id）

### 数值大小（int num\_int;double num\_double）

### 字符串内容（string str\_content）

### 构造与初始化（Scanner();Scanner(ifstream&);Scanner(istream&);）

### 析构函数（~Scanner();）

### 读取一个字符（void get\_char();）

### 获取词法符号（void get\_sym();）

### 分析关键字或标识符（void check\_keyword\_or\_ident();）

### 分析数字（void check\_number();）

### 分析字符串（void check\_operator();）

### 分析操作符（void check\_operator();）

### 分析缩进个数（int check\_tab(int);）

### 交互输入接口（void stream\_input(istream&);）

### 返回分析结果（tuple<Symbol,int,double,string> return\_output();）

### 返回分析结束flag（bool return\_flag();）

### 用户可以在所需的域内构造一个Scanner的对象。可以通过Scanner（ifstream&）构造，适用于脚本解释；也可以通过Scanner（istream&）或默认构造，适用于交互式解释；在交互模式下，可以使用void stream\_input(istream&)函数以动态输入新的代码。

### 通过使用tuple<Symbol,int,double,string> return\_output()函数以得到返回结果；返回值为一个四元组，分别为符号类型、整数数值（如果是整数）、浮点数数值（如果是浮点数）、词的原始内容（如果词本身是string常量，则返回的是其内容）；

### 用户还可以使用bool is\_empty()函数以检查代码是否分析完毕；

### 可以发现一些数据和函数接口是不必被用户使用的，但仍然给予了public权限。这是为了开发人员能够更加灵活地进行调试。

### 实现细节

a对于以输入流为参数的构造函数，类对象将把输入流的内容存入内部的缓存区，以备后续分析。

Scanner类对象在构造时将进行内部数据的初始化。包括对单字符符号值的一一设置，对保留字列表的设置（用于保留字查找），对保留字符号的设置。这些数据将方便于对字符的分析以及符号信息的更新。

分析的步骤是分为两个层次的：逐行读入缓冲区数据，逐字分析该行。之所以设置逐行读入，是为了减少磁盘I/O次数，同时也和交互式输入一致。

面向用户的输出函数为tuple<Symbol,int,double,string> return\_output()；以及bool return\_flag()；其余函数均为这两个函数所调用的辅助函数。

void get\_sym()；函数在忽略空格及注释的基础上，通过调用void get\_char()；来更新当前字符，并调用相关判断函数判明其类别并更新当前符号信息。

各个判断函数使用了不同的算法来确认类别合法并记录信息。一些比较重要的点如下：在寻找保留字时使用折半查找；在判断字符串内容时特别对转义字符的情况进行了分析，同时兼容双引号字符串常量和单引号字符串常量；在判断数值类型时对整数或浮点数进行了判断，对最大位为零的数的合法性进行了判断；特别对缩进层次的进位与回退做了巧妙的判断。具体算法可以参看源代码与注释。

void get\_char()；是更新当前字符的函数。若当前行结束，则更新行。在更新行的过程中完成了去除（带缩进的）空行和（带缩进的）注释行的功能，同时也对有效行进行了缩进记数，以便后续对层次关系的判断。

## 抽象语法树部分

### 概述

本解释器使用的抽象语法树是由c++标准库构建的，其中为了函数调用的方便，使用了c++17标准的一些新特性（即std::invoke)，此外没有依赖。

由于程序的解析能力和抽象语法树的实现密切相关，也就是说本解释器所能运行的代码只能在抽象语法树范围内。而本项目的语法树只实现了一个面向过程语言的功能，即抛弃了python中面向对象、异步等高级功能。

从这一方面来说，这些数据结构可以被用在任何一个编程语言上。不过对于python3，语法树预置了一些内置函数，从而更适配这门语言。

### 实现的功能

这颗抽象语法树能进行如下操作：

* 定义变量
* 运算，包括算术、移位、逻辑运算、比较运算
* 定义函数
* 删除变量（delete关键字）
* 分支语句（if-else，elif被展开而不是单独实现）
* 基本的两种容器：tuple和list
* 循环（for和while）

此外有一下特性：

* 垃圾回收
* 初步类型错误检测

内置函数只有两个：

* print
* range

他们是用c++的特性实现的

### API

对使用者而言，只需要使用ast的两种数据结构：表达式和语句。这两个类具备运行时多态的能力。

由于在定义节点时需要提供子节点，因此，语法分析器生成抽象语法树的过程应该是自底向上的。

表达式分为(类名)：

* UnaryOperation 单元运算
* BinaryOperation 双元运算
* CompareOperation 比较
* BooleanOperation 逻辑运算
* FunctionCall 函数调用
* Slice 切片
* Name 自定义标识符
* Number 整数或浮点数字面量
* String 字符串字面量
* NameConstant 包括：True False None () [] 后两个是空元组和空列表
* Formatted\_String 格式化字符串

语句分为（类名）：

* Pass\_Statement 空语句
* Delete\_Statement 删除语句
* Assign\_Statement 赋值语句
* AugAssign\_Statement 带有运算符的赋值语句，比如a+=1
* FunctionDefinition\_Statement 函数定义
* Expression\_Statement 如果表达式被计算，但其结果没有被使用，则属于此表达式语句
* Suite 语句块
* If\_Statement 分支语句
* While\_Statement 循环语句
* For\_Statement 带有容器范围的循环语句
* Break\_Statement 产生break信号
* Continue\_Statement 产生continue信号
* Return\_Statement 产生return信号，并返回指定的返回值

只使用表达式和语句两类结构就可以完成ast的构建。ast运行过程中使用的额外结构是隐藏的。

要创建这些结构，使用在AST.h中定义的宏CREATE。这个宏实际上是make\_shared的别名。ast内部的各个节点，必须使用智能指针相连，只也是其效率为何如此低下的原因。需要注意，不能在任何时候使用原生指针，否则可能会出现意想不到的内存泄漏，或者双重delete。

为创建函数调用时参数列表的方便，宏ArgList同时被定义。

要让节点附着到ast上，需要调用factory.addStatement( )。其中factory是对astfactory单例类静态方法调用的宏。

### 实现细节

ast的数据结构部分并不复杂，实际上上述列表已经指出了ast的大概结构，通过继承实现。这有点类似于模板方法，在数量不多的时候勉强可用。

ast在运行的时候会需要一些额外的结构的辅助，其一是函数类Function，它是内部类，由函数定义类创建、函数调用类调用，不被外部所见。函数类由函数名称、一系列参数和函数体组成，它的实现比较粗暴：每一次调用函数，就创建一个上下文（见下文），在其中手动把参数赋值为传入的值，这也意味着这里的传参数都是按值传参数，不能按引用传参数。在函数结束的时候，把本地上下文删除。

这些过程都是由函数调用类自动完成的（压栈等）。其中使用的第二个重要单例类：astfactory，虽然叫它factory，但这只是历史上的原因（名称已经固定），实际上它是ast本身的抽象，有点像是pool。它存储所有将要被运行的节点，其方法run会实际上运行这个程序。在此之前的操作是由语法分析器完成的，在此之后则是语法树本身的完成。它会动态地创建删除变量、定义调用函数。

其内部维护了函数的列表，上下文的列表，变量表等。每一个都对应相应的语句。上下文是很重要的，它是一个stl栈，存储了函数调用关系。每一个上下文对应一张变量表，变量表是变量名到其值（见下文）的stl map。由上所说，函数调用会创建一个临时的上下文。

需要说明的是，ast在其内部抽象了变量表，其对外的变量创建、赋值、删除方法是和上下文无关的。规则是：如果当前上下文的变量表不支持操作，则会转向global变量表，如果也不支持，则报错。这样做的好处是简化了赋值、创建、删除的语句实现。

第三个重要的类是returnvalue，这是一个所有类都用到的工具类，它是一系列值的集合，代表了返回值（或者说任何一个可能的值），它的类型是动态的，可以是数、字符串、容器、布尔值等，取决于调用者调用的构造函数。虽然这么做很浪费资源，但确保了python的动态类型。returnvalue作为返回值被嵌入ast基类astnode的虚函数exec（）中，当ast运行的时候，exec（）总会返回returnvalue，从而在树的内部保证动态。

## 语法分析器部分

### 概述

本python解释器使用的语法分析器（以下简称Parser）使用bison生成

bison是基于C的LALR(1)语法分析器生成器。

### 语法实现

（python语法参考自https://docs.python.org/3/reference/grammar.html）

本Parser使用python标准语法经过简化所得到的简化版语法

详见 parser.y (已经改写为适与LALR(1)的形式)

使用bison生成的parser需要实现返回词法token的int yylex(void)函数，以及出错信息函数void yyerror(string)

yylex使用Scanner包装实现，yyerror仅将错误信息打印到屏幕

\*注意：使用bison生成Parser后，需要将.h文件中的 typedef int YYSTYPE 改为 typedef myvalue YYSTYPE