

实验三 编制简单语法分析程序

1. 介绍

1.1 实验的内容与要求

实验内容：通过了解语法分析程序的功能，在词法分析的基础之上，能够基于识别出单词以及单词的分类得到实现程序的语法短语，如程序、短语、表达式等。

实验要求：掌握语法分析程序的功能和构造方法，编制实验报告。

1.2 主体介绍

词法分析将一段源程序（例如 C/C++）分解为不同类型的词素（token），这些词素有不同性质，表达不同的语义，分别为保留字，标识符，运算符，界符以及常量。而我们对于程序含义的理解确是基于句子或表达式为单位的，因此在编译过程中的第二步，需要将这些词素进行组合起来，进行语法分析，提取出语法解析树（Grammar Parse Tree），也就是从形式上而非语义上表达程序的执行逻辑。

语法分析中，我们需要完成以下工作：

- 将词素组合为[1]函数定义，[2]表达式，[3]变量声明，[4]分支结构，[5]函数调用，[6]特殊语句等等。
- 采用特定的文法结构，将其进行嵌套组合。
- 对于表达式，需要将其进行细粒度划分，最典型的是基于运算符优先级，将表达式中的每一个结点分解为树的形式。
- 承担相应的语法报错任务，例如，语句中非法字符，名词性冗余，表达式括号不匹配，以及运算符解析错误等。

当然，在实验中进行了简化考虑，没有对于面向对象的程序进行分析，没有建立完善的类模板查填修改相关的数据结构与机制，而仅仅是针对一些基础的程序进行分析。在编译原理的理论部分，介绍了两类语法分析的方法学，一种是自顶向下的分析方法，其中不包含左递归等结构的解析。另一种是自下而上的方法学，需要考虑递归结构。并且在理论部分，介绍了上下文无关文法的概念，其左端只有一个非终结符进行推导，很好稀了解析递归结构的复杂性。

实验中，具体做法是将自顶向下以及自下而上的语法分析方法相结合，采取这种策略是根据 C 语言这种特殊的语言规范：

- 在进行函数定义，特殊句式，变量声明，分支结构的解析，首先采用的是自顶向下的策略，因为若根据分号，大括号，缩进来进行句式分割，那么这四种结构外部不存在递归，而复杂的递归结构发生在其内部，我们从其内部提取出表达式与函数调用的杂糅结构。
- 在进行表达式以及函数调用时，对整体采用自下而上的递归分解策略，基于上下文无关文法，将其作树状解析，以运算符优先级顺序表为原则，将其分析为一个一个结点。这一部分以下会详述设计原则。

本文第二部分进行了语法分析器设计思路进行了详细阐述，内容包括以下：[1]上下文无关文法理论介绍，[2]语法分析器整体设计框架，以及语法解析树的结构，[3]函数定义的处理逻辑，[4]变量声明的处理逻辑，[5]分支语句的处理逻辑，[6]基于自下而上方法的表达式处理逻辑，[7]实验中考虑到的语法分析报错处理。

本文第三部分则进行了实验的样本集介绍以及结果分析，且介绍了部分代码，第四部分对于本文设计的语法分析器及进行总结，分析其优势与不足，第五部分则是基于上一次实验工程结构进行更新，并附上新增代码。

2. 程序思想与设计

2.1 上下文无关文法理论简介

上下文无关文法理论，作为一种特定的分解模式，贯穿了语法分析的理论基础。正是由于 C/C++ 等语言中，大量存在不确定的左递归结构，比如说“++func(a,b)*y = m % = *add”等语句的存在，我们不能按照循序逻辑的方式及逆行思考，需要尝试递归分解。

$$G=(VT,VN,S,P)$$

语法分析理论部分，将上下文无关文法定义为一个四元组，VT 表示终结符集合，而 VN 表示非终结符集合，在实际编程语言中，终结符通常是词法分析的五大类词素，而终结符通常是五大类词素的组合规则。S 则是代表开始符号，而上下文无关文法通常是非终结符的一员，而 P 则代表推导的表达式。

$$V_T = \{Keyword, Id, Operator, Seperator, Const\}$$

比如我们将刚才的递归结构“++func(a,b)*y = m % = *add”进行上下文无关文法的推导：

$$E \rightarrow E = E$$

$$E \rightarrow E \% = E$$

$$E \rightarrow E * E$$

$$E \rightarrow ++ E$$

$$E \rightarrow E(E, E)$$

$$E \rightarrow func$$

$$E \rightarrow a, E \rightarrow b$$

$$E \rightarrow y$$

$$E \rightarrow m$$

$$E \rightarrow *E$$

$$E \rightarrow add$$

以上推导虽然有些繁琐，但是相对来说比较朴素的一种推导，层层递归，分解到终结符，在分解的过程中，解析树的结构也通过递归被记录了。

2.2 语法分析器整体架构与解析树

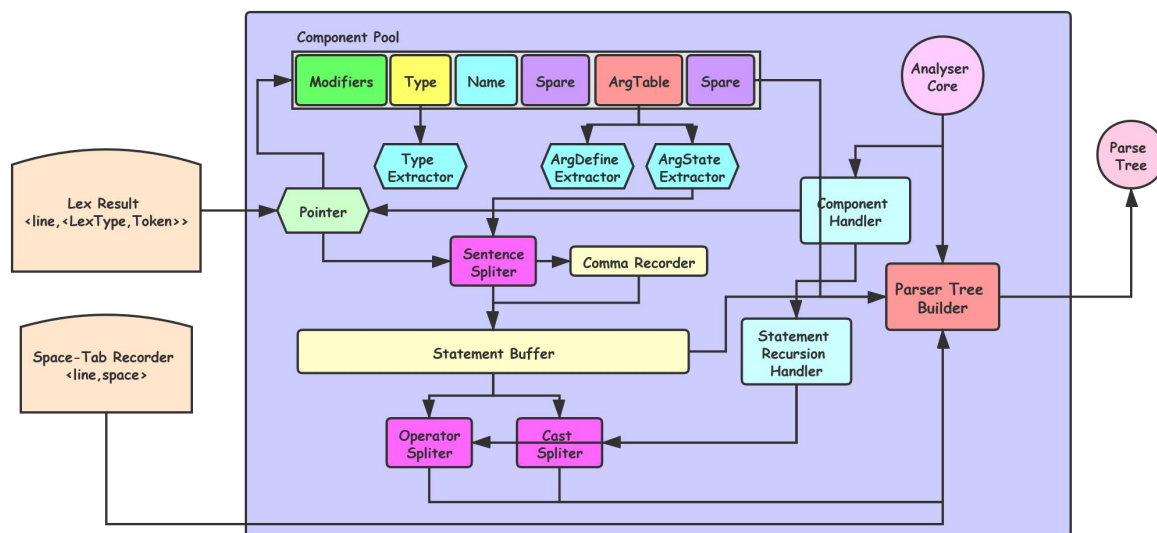


Fig.1 实验设计的语法分析器架构模型

上图展示了实验设计的语法分析器，其设计原则是自顶向下的顺序方法与自下而上的递归方法相结合。其中，输入分析器的参数一共两个，一个是词法分析器得到的 **token** 列表，一个是记录每一行缩进信息的缩进表。因为句法是相对有结构可循的，所以没有直接建立一个通用 **Buffer**，而是使用了组件池（**Component Pool**）：

- ◆ [1] 组件池中首先是修饰符，例如像 **public, inline, static, unsigned** 等词在函数定义，和变量声明时起到修饰作用，且修饰符可能不止一个。
- ◆ [2] 其次是变量类型，这里设计时也考虑了泛型，和类名作为类型，并且指针也必须包括在内（和 **gcc** 源码里处理方法不一样，实验简化了），这里为了减小程序耦合性，特别设计了 **Type Extractor** 这个结构。
- ◆ [3] **Name** 一般是标识符类的 **token**。
- ◆ [4] **Spare** 这里时区分几类句法的关键，如果存储“{”界符说明接下来，要么是函数定义声明，要么是调用构造器，当然这里不考虑大多数面向对象特性，默认为函数定义；而如果是变量声明时，则存储“;”界符或者“,”界符。
- ◆ [5] 如果是函数定义语句，那么这里就是进行参数列表进行提取，我们设计 **argTable Extractor**，用二元对的形式来存储参数名和参数类型。[6] 这里的 **spare** 语句，是存储“{”界符的。

组件池由 **Component Handler** 控制 **Pointer** 的移动，简介进行控制，可以直接将内容输出到 **parser Tree Builder** 中来构造解析树的一个结点，这是一种结构化的自顶向下思维模式的体现；**Component Handler** 除了解析函数定义，变量声明之外，还处理 **if, while, switch, case, default, for, foreach** 等循环分支，还处理 **break, return** 等特殊句式，这些在建立解析树的方法与前者又不一樣，下面章节会进行详述。**Component handler** 最最关键的是处理普通语句，也是语法分析器设计的难点。

而当进行到普通语句执行语句，分支语句的判断语句部分，会出现一类非结构化结构，是表达式和函数调用语句相互嵌套，相互耦合的结果。因此，我们使用 **Statement Recursion Parser** 这个结构：

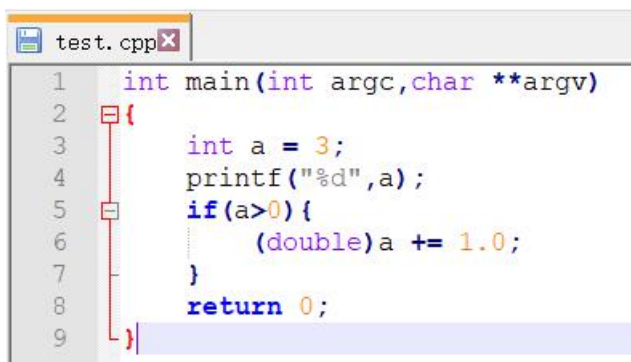
- 首先将 **Pointer** 传入的 **token** 进行 **split** 进行句法分词。
- 分词结果传入 **Statement Buffer, Buffer** 中使用索引来记录起始位置和结束位置。
- 然后基于运算符优先级表，和括号的优先级改变行为，将 **Buffer** 输入到 **Operator Spliter** 中进行结点的递归分解，可以分解为[1] **id** 结点，[2] 运算符结点，[3] 函数调用结点。其中函数调结点的参数列表还可以进行进一步的再分解。
- 此处，特殊考虑两类情况，强制转型，和数组符号，这两类，我们发现以后，是按照运算符来处理的。

语法分析器部分的架构大致如上所述，语法分析的结果是以解析树（**Parse Tree**）的结构来出现的，本实验也定义了自己的解析树结点规范：

- ✧ 根节点（**RootStatement**）：这类结点与语法分析器相联，除了 **child** 存储孩子结点指针以外无任何属性。
- ✧ 函数定义语句结点（**FuncDefineStatement**）：这类结点代表函数定义语句，拥有函数名，返回类型，修饰符，参数列表，定义行数，结束行数等属性，其孩子节点结点存储的是函数内部的语句。
- ✧ 变量声明语句结点（**VarDeclartion**）：这类结点代表变量声明语句，拥有变量名，变量类型，修饰符，以及声明行数等属性，孩子结点不存储信息。
- ✧ 分支节点（**branchStatement**）：这类结点代表分支循环语句，拥有分支类型，判断表达式（**if, while, else if, switch, case** 使用），起始，终止，迭代表达式（三者都是 **for** 循环使用），**foreach** 表达式（增强 **for** 循环使用），进入行数，推出行数，以及是否使用大括号界符来分割。孩子列表存储分支内部语句。

- ✧ 运算符结点（opNodeStatrment）：这类结点代表运算符，拥有运算符类型，运算符方向，使用行数，强制转型类型等属性，孩子列表存储两侧的子节点信息。
- ✧ 名词性结点（IdNodeStatement）：这类结点代表标识符，常量，部分关键字，拥有名称，词法类型，使用行数等信息，无孩子结点。
- ✧ 函数调用结点（FunctionStatement）：这类结点代表函数的调用，拥有名称，参数列表，使用函数等属性，无孩子结点，参数列表也是表达式。
- ✧ Break 特殊语句结点：代表 break 语句，除了使用行数，无其他属性。
- ✧ Return 特殊语句结点：代表 Return 语句，其孩子结点存储返回内容，除了使用行数无其他属性。

以上给出了解析树的结点类型，我们通过将结点树状组织，就可以构建解析树。在实现时，使用继承的方式，统一定义父类 Statement，并且使用虚函数反射的机制，来进行结点类型的解读（相当于 java 中的 InstanceOf）。



```

1  int main(int argc, char **argv)
2  {
3      int a = 3;
4      printf("%d", a);
5      if(a > 0){
6          (double) a += 1.0;
7      }
8      return 0;
9  }
  
```

Fig.2.1 示例代码

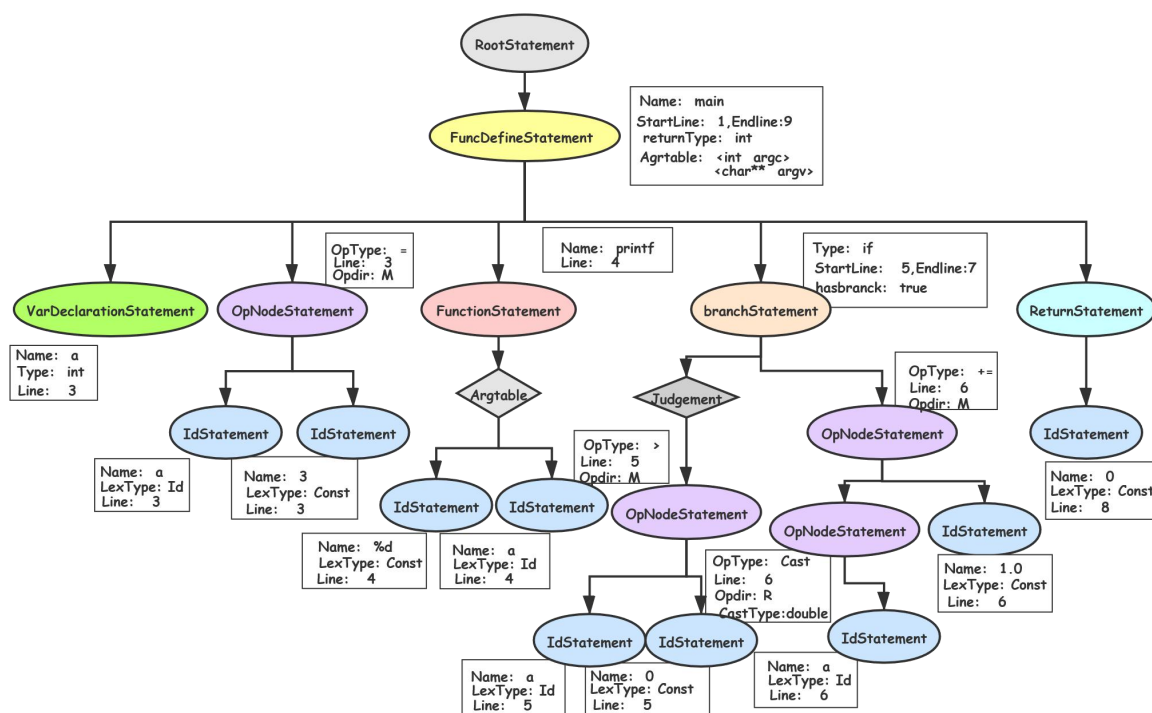


Fig.2.2 示例代码对应解析树

上图展示了按照本文的语法解析树的定义，一段基础代码对应的解析树图形。

2.3 函数定义处理逻辑

在介绍具体的分块逻辑之前，先讲解一下对于每一个语句组件（Component）的综合解析原则

进行说明。本语法分析器以组件为基本单位，以自由的游标为指针，遍历整个词法分析结果列表来构建解析树：

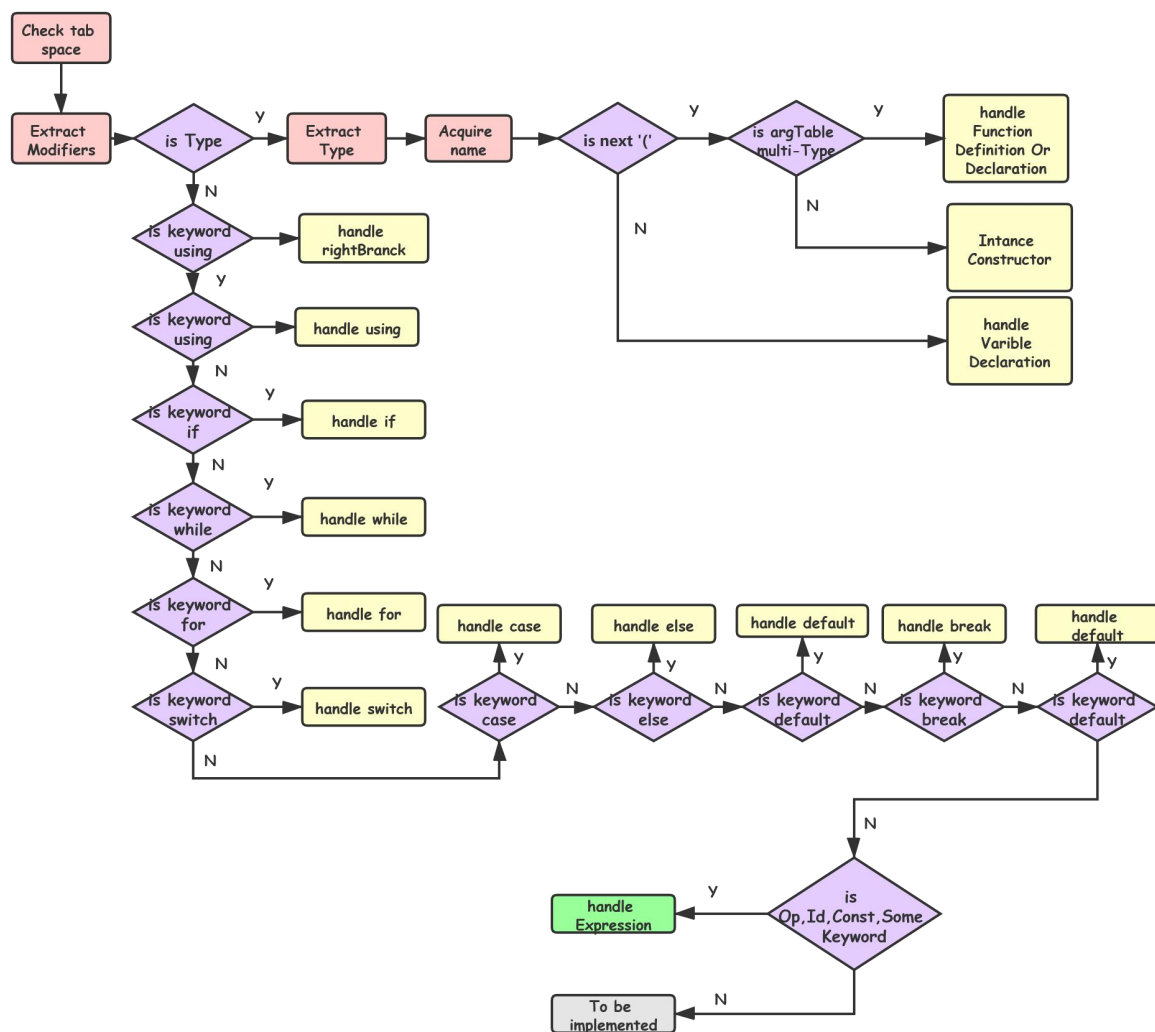


Fig.3 对于每一个 Component 综合处理逻辑

如上，我们首先提取开头的修饰符一类的词，接下来判断剩余组件首个 token 是否可以作为类型（返回类型，变量类型）。如果是，那么久有四种可能性：变量声明，函数声明，函数定义，对象的构造器调用。其中，函数定义在名称后面为小括号，小括号内部的参数类标是“类型+参数名”的模式，而最后跟随的是大括号：

$\langle Modifiers \rangle \langle Type \rangle \langle Name \rangle ([\langle argType \rangle \langle argName \rangle]^*) \{$

其处理方式也非常简单：

- 在判断是否为类型之前提取修饰符。
- 在判断为类型以后，提取类型。
- 或者名称。
- 在判断存在小括号以后，提取参数列表，以<参数类型,参数名>二元组形式存储。
- 之后判断是'{'结尾，就建立函数定义结点。
- 将当前该结点挂接到当前解析结点的孩子列表中。
- 修改当前解析结点指针，为新建结点。

2.4 变量声明处理逻辑

变量声明语句本身并不复杂，只要提取出修饰符，类型，名称，在构建结点即可。但是，事实上 C/C++ 支持在声明同时就进行初始化赋值，并且，可以使用逗号隔开的几个类型相同的声明加赋值语句省略后续语句的类型，这就给我们解析带来了麻烦。并且，这样的句式结构中类型可以缺省，指针却不行。

int a = 3; (1)

int a = 3, b = 4 (2)

int a = 3, double b = 1.0 (3)

int* a = 3, *b = 4; (4)

以上四个语句解释了这种语法结构，(1) 是一个普通的声明语句，(2) 是一个符合复合缺省声明，同时声明了两个 int，(3) 中则是用逗号分隔了两个不同类型的变量声明，(4) 中定义了两个指针，但是虽然 int 被缺省，* 无法被缺省。

为了考虑这一现象，故进行以下流程：

- 在判断类型前提取修饰符。
- 判断为类型，则进行类型的提取。
- 根据分号进行分句，在句子内部，再根据逗号进行二次切割。
- 每一个子句进行切割，如果是声明同时初始化了，那么就将声明和赋值分开来，如果某一个子句没有声明类型，就沿用前一个子句的类型进行变量的声明。
- 每一个子句，建立变量声明结点，挂接到当前解析树结点的孩子列表中。

2.5 分支语句逻辑处理

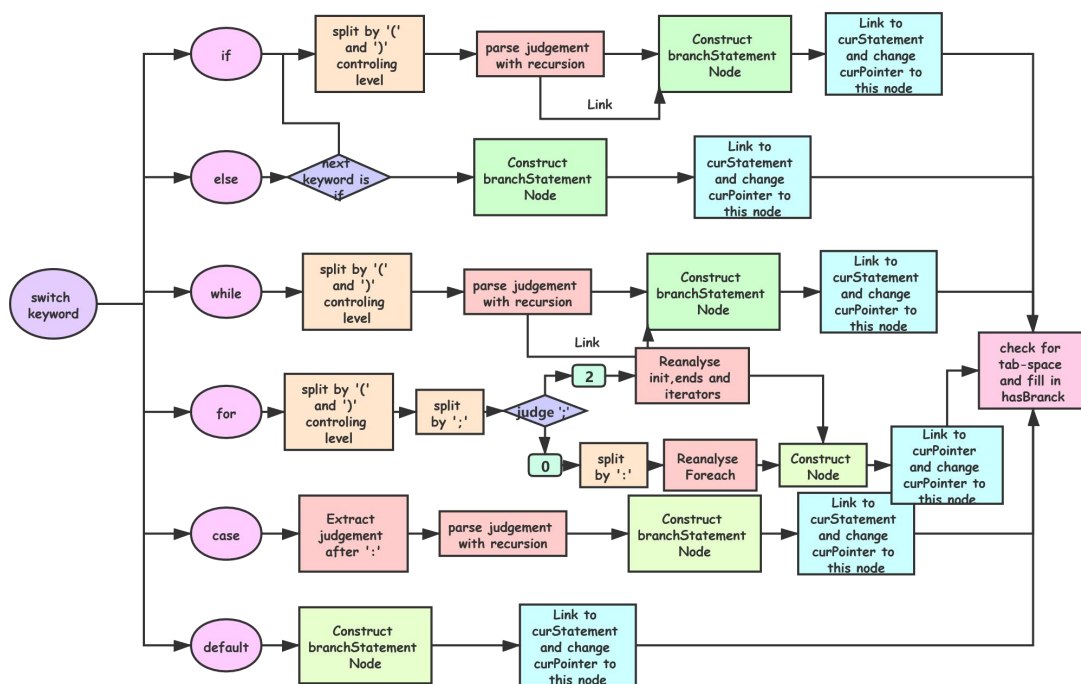


Fig.4 分支语句处理逻辑

由于分支语句与函数一样，都是将程序进一步划分出域，所以再构建语法解析树时，需要向下进行分裂。非常值得注意的是，C 语言中有一类特殊的语法，不采用大括号作为界符，通过缩进来控制域的进入与进出，这就是此语法分析器引入缩进表的作用。在这个逻辑里，我们在结尾处只要判断有无大括号作界符即可，将其填入分支结点的属性。

不同分支判断语句的提炼方法不同，if, while, switch, else if 直接就在括号中，在语义分析中会检查 switch 判断语句的类型，语法分析部分一视同仁。For 循环普通的就直接通过分号提取初始执行语句，

结束判断语句，以及迭代附加语句，增强 `for` 循环可以得到声明的变量语句，以及变量迭代语句。`Case` 通过冒号和缩进来分割判断条件，`else, default` 无判断条件，直接进入子域即可。

这里顺便说下，在整个流程开始部分会检查缩进表，如果改行的缩进大于等于上一行的缩进，且当前域为分支，并且 `hasBranck` 属性为 `false`,那么就是将当前分析结点指针向其父亲结点退一格。

2.6 基于自下而上方法的表达式（函数调用）处理逻辑

表达式虽然比较复杂，但是其中函数调用作为一个划分单元，可以看作是一个整体，而真正可以进行递归划分的是运算符，标识符，常量的组合，如果将标识符和常量看作是名词，那么运算符就是谓语，基于谓语的划分是自然语言处理中非常重要的一个环节。其中，核心就是运算符优先级顺序表，以及可以改变运算符运算顺序的括号机制。

Table.1 本次实验使用的运算符优先表

运算符	::	[]	++	--	.	->	++	--
运算符含义	类的成员	数组	后缀自增	后缀自减	结构体成员	结构体指针	前缀自增	前缀自减
方向	M	M	L	L	M	M	R	R
优先级	1	2	2	2	2	2	3	3
运算符	+	-	!	~	Cast	*	&	*
运算符含义	正号	负号	逻辑非	按位取反	强制转型	指针	取地址符	乘号
方向	R	R	R	R	R	R	R	M
优先级	3	3	3	3	3	3	3	5
运算符	/	%	+	-	>>	<<	<	<=
运算符含义	除号	取余	加号	减号	向右位移	向左位移	小于	小于等于
方向	M	M	M	M	M	M	M	M
优先级	5	5	6	6	7	7	8	8
运算符	>	>=	==	!=	&	^		&&
运算符含义	大于	大于等于	等于	不等于	位与	位异或	位或	与
方向	M	M	M	M	M	M	M	M
优先级	8	8	9	9	10	11	12	13
运算符		?:	=	+=	-=	*=	/=	%=
运算符含义	或	三目选择	赋值	加等于	减等于	乘等于	除等于	取余等于
方向	M	M	M	M	M	M	M	M
优先级	14	15	16	16	16	16	16	16
运算符	<<=	>>=	&=	=	^=			
运算符含义	左移等于	右移等于	位与等于	位或等于	位异或等于			
方向	M	M	M	M	M			
优先级	16	16	16	16	16			

对于表达式处理的第一步是寻找括号最外层，并且运算符优先级最低的运算符位置，若遇到相同优先级，16 级和 3 级从右向左进行定位，剩余级别从左向右定位。实验非常注重对于括号的处理，每一步都定义了 `leftbranck` 标记，用于记录小括号的嵌套信息。

处理流程如下：

- 若最外层存在括号，就先去除最外层的括号
- 对于表达式进行语法检查，主要检查非法界符，名词性冗余，括号不匹配这三类情况。
- 如果表达式只有一个 `token`，若为标识符，常量，部分关键字（如 `true, false`）直接建立结点，向上传结点的指针并返回，若是运算符，则进行报错处理。

- 查找在括号嵌套之外的优先级最低的运算符 **token**，此过程考虑到了运算符的方向，**M,L,R** 分别代表双目运算符，单目运算符左边有值，单目运算符右边有值。
- 查找表达式中是否存在强制转型，将其考虑为运算符，共同参与查找运算符的决策。
- 如果查找到的运算符，为数组，则进行整合处理：

$$arr[expression] \rightarrow arr[]expression$$

- 如果查找到运算符，先生成运算符结点，然后根据方向，对于两边的剩余部分进行递归解析，解析结果连接到运算符结点的孩子结点上。
- 若未查找到运算符，就视为函数的调用，建立函数调用结点，递归对于每一个参数运算符进行递归解析，递归结果连接到调用结点的 **argTable** 属性上。

这一部分细节比较多，包含对于数组的处理，从「查找到」逻辑相对来说是比较复杂的，这一部分也负责了相当多的报错处理。

2.7 语法分析部分所考虑的报错处理

语法分析部分主要考虑的报错是句法形式上的错误，是从 **token** 的组合与搭配上来看，仍然属于 **syntax Error** 的范畴，考虑到的报错如下表所示：

Table.2 实验考虑的语法分析报错检查

错误编号	错误类型	错误名称	错误具体信息	处理模块
8	Syntax Error	函数缺少开始标记	函数定义完成以后，没有界符{就开始了内部的语句	GrammarAnalyser
9	Syntax Error	函数变量声明不完整	在填充组件池的时候，提前出现的界符分号与逗号等	GrammarAnalyser
10	Syntax Error	过多的右大括号	右大括号出现时，并未有可以退出的域	GrammarAnalyser
11	Syntax Error	循环表达式括号不封闭	在提取判定表达式时，未扫描到)就进入到分支子域中	GrammarAnalyser
12	Syntax Error	声明语句名称缺省	声明语句类型之后非标标识符	GrammarAnalyser
13	Syntax Error	名词性冗余	表达式中两个名词性 token 直接逻辑相邻	GrammarAnalyser
14	Syntax Error	表达式非法字符	表达式中存在非法的界符	GrammarAnalyser
15	Syntax Error	表达式括号不匹配	表达式中，做小括号与右小括号数量并不相等	GrammarAnalyser

相对于词法分析来说，语法分析的格式更加严格，函数声明，定义，调用，变量声明，调用，表达式的组合，运算符左右，都存在严格的规范，如果要进行完善的报错机制，经过长时间的路径覆盖测试时最行之有效的。由于不是实际的项目，报错的考虑进行了简化，只考虑了八种语法错误。

3. 实验结果与分析

3.1 部分源码解析

本次实验设计的 **GrammarAnalyser** 根据 2.2 节中的设计思路，但在实际工程中，为了方便使用，也添加了一些具有别的功能的函数，以下给出 **GrammarAnalyser** 的部分头文件：

```

1.  class GrammarAnalyser
2.  {
3.  public:
4.      GrammarAnalyser(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> lexRes, std::vector<std::pair<int, int>> spaceRecord);
5.      ~GrammarAnalyser();
6.      void grammar();
7.      void printParseTree();
8.  private:
9.      std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> lexRes; // 词法分析的结果
10.     std::vector<std::pair<int, int>> spaceRecord; // 存放缩进表
11.     void init();

```



```

12.   int cur = 0; // token 的指针索引
13.   Statement* root = NULL; // 解析树的根节点
14.   Statement* curStatement = NULL; // 在构造解析树时，当前结点
15.   void parseComponent();
16.   void extractType(std::string& type);
17.   void extractType(std::string& type, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& tmpResBin, int& cursor);
18.   void extractArgTable(std::vector<std::pair<std::string, std::string>>& argTable);
19.   void extractJudgement(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& tmpResBin);
20.   void searchForCast(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>bin, int startPos, int endPos, int& castPosStart, int& CastPosEnd, std::string& CastType);
21.   void split(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& tmpResBin, std::vector<int>& commaedPos, std::string end, std::string mid);
22.   void split(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& tmpResBin, std::string end);
23.   void split(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> src, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> &dst, std::vector<int>& commaedPos, std::string end, std::string mid, int& cursor);
24.   void handleUsing();
25.   void handleFunc(std::vector<std::string> modifiers, std::string type, std::string name, int startLine);
26.   void handleVarDeclaration(std::vector<std::string> modifiers, std::string type, std::string name, int startLine);
27.   void handleNode(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& bin, int startPos, int endPos, Statement*& handle);
28.   void handleEndsWithBracket();
29.   void handleIf();
30.   void handleElse();
31.   void handleWhile();
32.   void handleFor();
33.   void handleAssign();
34.   void handleSwitch();
35.   void handleCase();
36.   void handleBreak();
37.   void handleDefault();
38.   void handleIndentation();
39.   void handleReturn();
40. };

```

从头文件的设计中，我们可以发现：

- 实验的数据导入接口在 GrammarAnalyser 这个构造器中，而调用 grammar 开始进行语法分析的外部结构，printParseTree 则是打印语法解析树。
- 本工程以分析一个组件（component）为单位进行分析的（一般是一个语句），parseComponent 就是根据 cur 游标索引的位置来进行一个组件的解析分块逻辑的入口，这一部分是使用自顶向下的模式设计的。
- lexRes, 和 spaceRecord 分别记录了词法分析部分的 token 结果和缩进表，而 cur 则为词法分析 token 容器的游标索引，root 存储解析树根节点，curStatement 存储的是当前正在分析的解析树结点指针。
- 分块逻辑使用 handle 来打头的函数，其中，handleVarDeclaration, handleFunc 是负责处理变量声明语句以及函数定义语句；而 handleIf, handleElse, handleWhile, handleSwitch, handleFor 等等则是处理循环分支语句的，handleUsing, handleBreak, handleReturn 则是处理特殊句式，handleEndsWithBracket, handleIndentation 测试除了右大括号和检查缩进信息的，管理域的退出和解析树向父亲结点的迁移。
- 整个工程最精髓的设计在于 handleNode 函数对于表达式的分割，设计思路已经阐述过，根绝运算符优先级以及括号的嵌套来进行递归。而 handleAssign 函数则是根据逗号进行普通陈述语句的分割，分别调用 handleNode。

- 剩余的私有函数都是起到一些工具作用，三个重载的 `split` 适应三种不同情况下的分句，`extractType` 是提取类型，因为要考虑到泛型指针等等，`extractArgTable` 用于填充参数列表，`extractJudge` 用于提取判断类型表达式，`searchForCast` 是用于搜索是否存在强制转型的工具。

整个工程的设计出于降低耦合性的原则，尽可能增加可扩展的接口，尽可能考虑周全。

3.2 实验结果分析

实验一共选择了三个 `cpp` 进行分析，一个是比较基础的测试，一个是语法分析器 `LexAnalyser.cpp`，另一个是 MPI 的 PSRS 算法的 `cpp` 文件。其中，`LexAnalyser` 由于采用了一小部分面向对象的设计方法，而本语法分析器还没有进行设计，因此就做了轻微的改动。

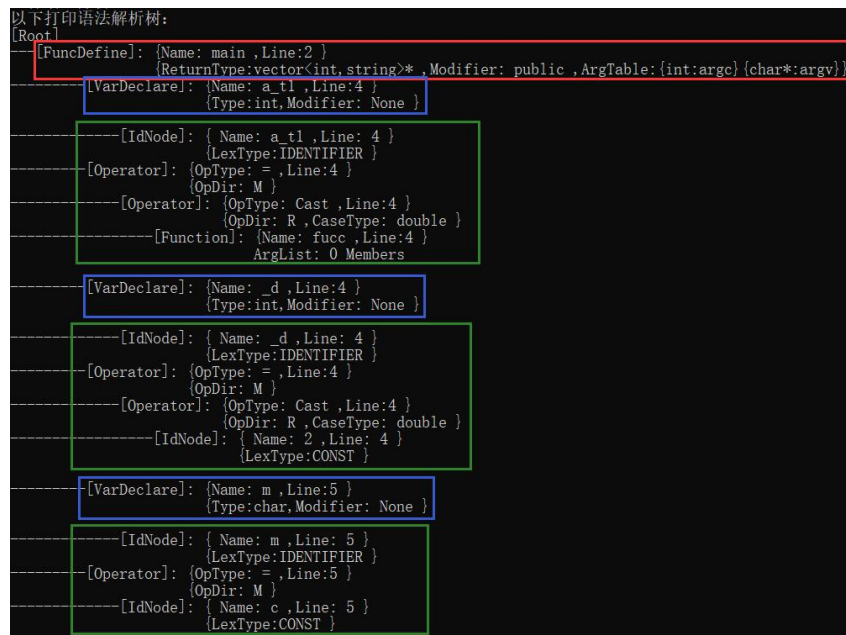


Fig.5 基础测试部分截图

由于语法解析树太大了，所以直接了一小部分截图，上面用红色框出的部分是函数定义结点的信息，蓝色部分则是一个变量声明语句，绿色部分框出的一个一个表达式，第二个绿框里面包含一个强制转型结点，和一个函数调用结点。

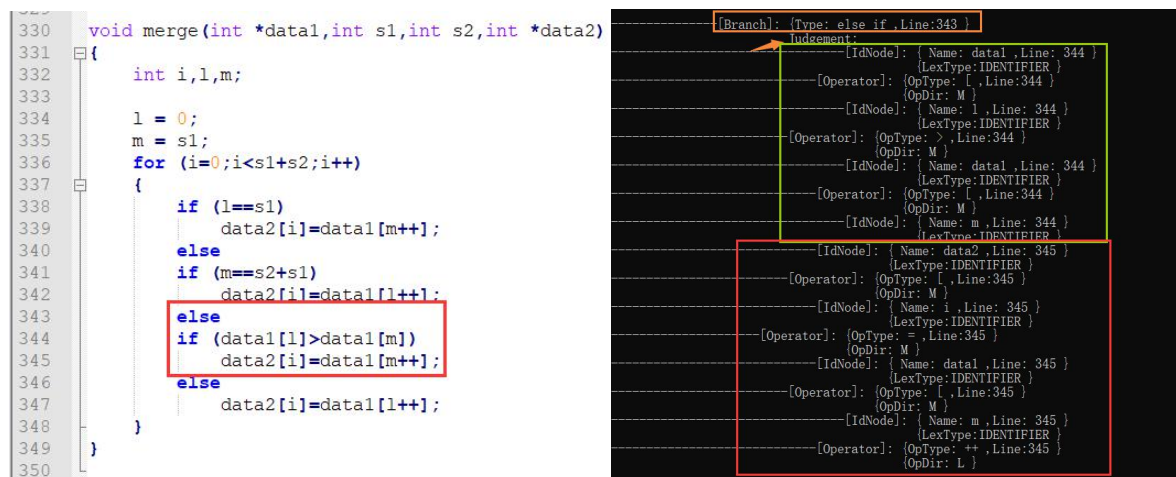


Fig.6 PSRS 算法代码的解析部分截图

上图可以看到，这一个部分是一个 **else if** 分支语句，其中包含了数组的访问，以及数组内嵌套表达式，右面的解析树对于这一部分进行了完整的解析。

4. 结束语

本次语法分析主要基于句子为一个组件进行了解析，考虑到了相当多的因子，变量声明的特殊句式，函数调用域表达式的相互之间嵌套，强制转型与小括号改变优先级，打印出的解析树也比较合理。但是，实验考虑的东西还是有所欠缺：

- 比如 `sizeof`，我们将其作为函数来考虑，但是其内部跟随的是基础类型，比如 `sizeof(double)`，程序未将其判断为函数调用格式发生异常，但实际上，C++设计时是作为一个运算符来考虑的。
- 比如面向对象中的构造器，例如“`std::ifstream infile(fin)`”这个函数，实际上声明一个 `infile` 类型的变量，并且将 `infile(fin)`构造器函数的结果传给这个对象，而语法分析对于构造器的语法并没有做得很鲁棒。

5. 附录

5.1 工程结构于配置文件说明

在附件中，主要展示的是实验参数以及部分源代码，以及工程的说明，以下是项目的结构：（添加了语法分析部分四个文件）

```
|---- Compiler
|   |---- Compiler
|   |   |---- headFile
|   |   |   |---- Base.h
|   |   |   |---- LexAnalyser.h
|   |   |   |---- parseTree.h
|   |   |   |---- GrammarAnalyser.h
|   |   |---- SourceFile
|   |   |---- LexAnalyser.cpp
|   |   |---- parseTree.cpp
|   |   |---- GrammarAnalyser.cpp
|   |   |---- main.cpp
|   |---- data
|   |   |---- testCompiler1.cpp
|   |   |---- testCompiler2.cpp
|   |   |---- testCompiler3.cpp
|   |   |---- Num.txt
|   |   |---- Op.txt
|   |---- result
|   |---- Readme.txt
|   |---- run.bat
```

本实验的文件输入路径用宏定义的方式写在 `Base.h` 头文件中，测试环境使用了 Windows10，VS2019，内存 16G，处理器 i7 第九代，配置中等偏上。

5.2 parseTree.h

```
1.  /*****
2.  * 模块名:    parseTree
3.  * 文件名:    parseTree.h
4.  * 依赖文件:  Base.h
5.  * 作用:      对于 C/C++语言进行语法分析的基础数据机构--解析树
6.  * 作者:      陈骥飞
7.  * 时间:      2020.6.2
8.  * 版本:      Version 1.1
9.  *****/
```

```

10. #ifndef _PARSETREE_H_
11. #define _PARSETREE_H_
12. #include "Base.h"
13. #include <vector>
14. #include <iostream>
15.
16. /*****
17.  * 类名:      Statement
18.  * 作用:      解析树基础结点父类
19.  *****/
20. class Statement
21. {
22. public:
23.     /*****
24.     * 函数名:  Statement
25.     * 作用:    构造器
26.     * 参数:    无
27.     *****/
28.     Statement();
29.     /*****
30.     * 函数名:  ~Statement
31.     * 作用:    析构函数
32.     *****/
33.     ~Statement();
34.     /*****
35.     * 函数名:  getParent
36.     * 作用:    获取父亲结点
37.     * 参数:    无
38.     * 返回值:  Statement*
39.     *****/
40.     Statement* getParent();
41.     /*****
42.     * 函数名:  addChild
43.     * 作用:    添加结点到自己的孩子, 同时修改孩子结点的父指针
44.     * 参数:    child      孩子结点指针[in]
45.     * 返回值:  无
46.     *****/
47.     void addChild(Statement* child);
48.     /*****
49.     * 函数名:  getInstanceName
50.     * 作用:    虚函数反射, 获取类名
51.     * 参数:    无
52.     * 返回值:  string
53.     *****/
54.     virtual std::string getInstanceName() {
55.         return "Statement";
56.     }
57.     /*****
58.     * 函数名:  getInfo
59.     * 作用:    虚函数, 打印解析树时打印自己结点的信息
60.     * 参数:    level      当前所处层次[in]
61.     * 返回值:  无
62.     *****/
63.     virtual void getInfo(int level);
64. protected:
65.     std::vector<Statement*> childs; // 孩子结点指针列表
66.     Statement* parent = NULL; // 父亲结点指针
67. };
68.
69. /*****
70.  * 类名:      RootStatement
71.  * 作用:      解析树根结点
72.  * 父类:      Statement
73.  *****/
74. class RootStatement :public Statement{
75. public:
76.     /*****
77.     * 函数名:  RootStatement
78.     * 作用:    构造器
79.     * 参数:    无
80.     *****/
81.     RootStatement();
82.     /*****
83.     * 函数名:  ~RootStatement
84.     * 作用:    析构函数
85.     *****/
86.     ~RootStatement();
87.     /*****
88.     * 函数名:  getInstanceName
89.     * 作用:    虚函数反射, 获取类名
90.     * 参数:    无
91.     * 返回值:  string
92.     *****/
93.     virtual std::string getInstanceName() {
94.         return "RootStatement";
95.     }
96.     /*****
97.     * 函数名:  getInfo
98.     * 作用:    虚函数, 打印解析树时打印自己结点的信息
99.     * 参数:    level      当前所处层次[in]
100.    * 返回值:  无
101.    *****/
102.    void getInfo(int level);
103. };
104.
105. /*****
106.  * 类名:      FuncDefineStatement
107.  * 作用:      解析树函数定义语句结点
108.  * 父类:      Statement
109.  *****/
110. class FuncDefineStatement :public Statement {
111. public:
112.     /*****

```

```

113.     * 函数名:   FuncDefineStatement
114.     * 作用:    构造器
115.     * 参数:    name      函数名      [in]
116.     *           returnType 函数返回值类型[in]
117.     *           modifier   函数修饰符   [in]
118.     *           startLine  函数定义行数 [in]
119.     *****/
120.     FuncDefineStatement(std::string name, std::string returnType, std::vector<std::string> modifier, int startLine);
121.     /***/
122.     * 函数名:   ~FuncDefineStatement
123.     * 作用:    析构函数
124.     *****/
125.     ~FuncDefineStatement();
126.     /***/
127.     * 函数名:   setArgTable
128.     * 作用:    添加函数参数列表
129.     * 参数:    argTable   参数列表[in]
130.     * 返回值:   无
131.     *****/
132.     void setArgTable(std::vector<std::pair<std::string, std::string>> argTable);
133.     /***/
134.     * 函数名:   setEndLine
135.     * 作用:    添加函数结尾行数
136.     * 参数:    endLine     函数结尾行数[in]
137.     * 返回值:   无
138.     *****/
139.     void setEndLine(int endLine);
140.     /***/
141.     * 函数名:   getInstanceName
142.     * 作用:    虚函数反射, 获取类名
143.     * 参数:    无
144.     * 返回值:   string
145.     *****/
146.     virtual std::string getInstanceName() {
147.         return "FuncDefineStatement";
148.     }
149.     /***/
150.     * 函数名:   getInfo
151.     * 作用:    虚函数, 打印解析树时打印自己结点的信息
152.     * 参数:    level      当前所处层次[in]
153.     * 返回值:   无
154.     *****/
155.     void getInfo(int level);
156. private:
157.     std::string returnType; // 函数的返回值类型
158.     std::vector<std::string> modifier; // 函数的修饰符
159.     std::vector<std::pair<std::string, std::string>> argTable; // 函数的参数列表
160.     std::string name; // 函数名
161.     int startLine; // 函数定义的起始行
162.     int endLine; // 函数定义的结尾行
163. };
164.
165. /***/
166. * 类名:   VarDeclarationStatement
167. * 作用:   解析树变量声明语句结点
168. * 父类:   Statement
169. *****/
170. class VarDeclarationStatement :public Statement{
171. private:
172.     std::string name; // 变量名
173.     std::string type; // 变量类型
174.     std::vector<std::string> modifiers; // 变量修饰符
175.     int startLine; // 变量声明行数
176. public:
177.     /***/
178.     * 函数名:   VarDeclarationStatement
179.     * 作用:    构造器
180.     * 参数:    name      变量名      [in]
181.     *           type      变量类型    [in]
182.     *           modifier   变量修饰符 [in]
183.     *           startLine  变量声明行数 [in]
184.     *****/
185.     VarDeclarationStatement(std::string name, std::string type, std::vector<std::string> modifier, int startLine);
186.     /***/
187.     * 函数名:   ~VarDeclarationStatement
188.     * 作用:    析构函数
189.     *****/
190.     ~VarDeclarationStatement();
191.     /***/
192.     * 函数名:   getInstanceName
193.     * 作用:    虚函数反射, 获取类名
194.     * 参数:    无
195.     * 返回值:   string
196.     *****/
197.     virtual std::string getInstanceName() {
198.         return "FuncDefineStatement";
199.     }
200.     /***/
201.     * 函数名:   getInfo
202.     * 作用:    虚函数, 打印解析树时打印自己结点的信息
203.     * 参数:    level      当前所处层次[in]
204.     * 返回值:   无
205.     *****/
206.     void getInfo(int level);
207. };
208.
209. /***/
210. * 类名:   branchStatement
211. * 作用:   解析树分支语句结点
212. * 父类:   Statement
213. *****/
214. class branchStatement :public Statement {

```

```

215. private:
216.     std::string branchType; // 分支的类型
217.     int startLine; // 分支语句进入行数
218.     int endLine; // 分支语句结束行数
219.     bool hasBrack = false; // 分支语句是否使用大括号作为界符
220. public:
221.     /*****
222.     * 函数名:   branchStatement
223.     * 作用:     构造器
224.     * 参数:     type      分支类型      [in]
225.               startLine  变量声明行数  [in]
226.     *****/
227.     branchStatement(std::string type,int startLine);
228.     /*****
229.     * 函数名:   ~branchStatement
230.     * 作用:     析构函数
231.     *****/
232.     ~branchStatement();
233.     /*****
234.     * 函数名:   setHasBrack
235.     * 作用:     设定是否使用大括号属性
236.     * 参数:     hasBrack   是否使用大括号属性[in]
237.     * 返回值:   无
238.     *****/
239.     void setHasBrack(bool hasBrack);
240.     /*****
241.     * 函数名:   setEndLine
242.     * 作用:     设定分支语句结束行数
243.     * 参数:     endLine    结束行数[in]
244.     * 返回值:   无
245.     *****/
246.     void setEndLine(int endLine);
247.     /*****
248.     * 函数名:   getHasBrack
249.     * 作用:     分支与是否使用大括号属性
250.     * 参数:     无
251.     * 返回值:   bool
252.     *****/
253.     bool getHasBrack();
254.     /*****
255.     * 函数名:   getStartLine
256.     * 作用:     获取分支语句起始行数
257.     * 参数:     无
258.     * 返回值:   int
259.     *****/
260.     int getStartLine();
261.     /*****
262.     * 函数名:   getInstanceName
263.     * 作用:     虚函数反射, 获取类名
264.     * 参数:     无
265.     * 返回值:   string
266.     *****/
267.     virtual std::string getInstanceName() {
268.         return "branchStatement";
269.     }
270.     /*****
271.     * 函数名:   getInfo
272.     * 作用:     虚函数, 打印解析树时打印自己结点的信息
273.     * 参数:     level    当前所处层次[in]
274.     * 返回值:   无
275.     *****/
276.     void getInfo(int level);
277.     Statement* judgement; // 用于 if,while,switch 循环的判断语句
278.     std::vector<Statement*> init; // 用于 for 循环起始语句
279.     Statement* end; // 用于 for 循环结尾判断语句
280.     std::vector<Statement*> iterate; // 用于 for 循环迭代语句
281.     std::vector<Statement*> forEachDecl; // 用于增强 for 判断语句
282. };
283.
284. /*****
285. * 类名:   breakStatement
286. * 作用:   解析树 break 特殊语句结点
287. * 父类:   Statement
288. *****/
289. class breakStatement :public Statement {
290. public:
291.     /*****
292.     * 函数名:   breakStatement
293.     * 作用:     构造器
294.     * 参数:     无
295.     *****/
296.     breakStatement(int line);
297.     /*****
298.     * 函数名:   ~breakStatement
299.     * 作用:     析构函数
300.     *****/
301.     ~breakStatement();
302.     /*****
303.     * 函数名:   getInstanceName
304.     * 作用:     虚函数反射, 获取类名
305.     * 参数:     无
306.     * 返回值:   string
307.     *****/
308.     virtual std::string getInstanceName() {
309.         return "breakStatement";
310.     }
311.     /*****
312.     * 函数名:   getInfo
313.     * 作用:     虚函数, 打印解析树时打印自己结点的信息
314.     * 参数:     level    当前所处层次[in]
315.     * 返回值:   无
316.     *****/

```



```

317.     void getInfo(int level);
318. private:
319.     int line; // break 语句所属行数
320. };
321.
322. /*****
323. * 类名:     returnStatement
324. * 作用:     解析树 return 特殊语句结点
325. * 父类:     Statement
326. *****/
327. class returnStatement :public Statement {
328. public:
329.     /*****
330.     * 函数名:     returnStatement
331.     * 作用:     构造器
332.     * 参数:     line      return 语句执行行数[in]
333.     *****/
334.     returnStatement(int line);
335.     /*****
336.     * 函数名:     ~returnStatement
337.     * 作用:     析构函数
338.     *****/
339.     ~returnStatement();
340.     /*****
341.     * 函数名:     getInstanceName
342.     * 作用:     虚函数反射, 获取类名
343.     * 参数:     无
344.     * 返回值:     string
345.     *****/
346.     virtual std::string getInstanceName() {
347.         return "returnStatement";
348.     }
349.     /*****
350.     * 函数名:     getInfo
351.     * 作用:     虚函数, 打印解析树时打印自己结点的信息
352.     * 参数:     level      当前所处层次[in]
353.     * 返回值:     无
354.     *****/
355.     void getInfo(int level);
356. private:
357.     int line; // reutrn 语句所属行数
358. };
359.
360. /*****
361. * 类名:     OpNodeStatement
362. * 作用:     解析树运算符结点
363. * 父类:     Statement
364. *****/
365. class OpNodeStatement :public Statement {
366. private:
367.     std::string OpType; // 运算符内容
368.     std::string OpDir; // 运算符方向 M 双目, L 单目, 左边右表达式, R 双目, 右边有表达式
369.     int startLine; // 运算符执行行数
370.     std::string castType; // (附加属性) 强制转型类型
371. public:
372.     /*****
373.     * 函数名:     OpNodeStatement
374.     * 作用:     构造器
375.     * 参数:     OpType      运算符内容[in]
376.     *           opDir       运算符方向[in]
377.     *           startLine   起始行数[in]
378.     *****/
379.     OpNodeStatement(std::string OpType, std::string opDir,int startLine);
380.     /*****
381.     * 函数名:     OpNodeStatement
382.     * 作用:     构造器
383.     * 参数:     OpType      运算符内容[in]
384.     *           opDir       运算符方向[in]
385.     *           startLine   起始行数[in]
386.     *           castType    强制转型类型[in]
387.     *****/
388.     OpNodeStatement(std::string OpType, std::string opDir, int startLine,std::string castType);
389.     /*****
390.     * 函数名:     ~OpNodeStatement
391.     * 作用:     析构函数
392.     *****/
393.     ~OpNodeStatement();
394.     /*****
395.     * 函数名:     getInstanceName
396.     * 作用:     虚函数反射, 获取类名
397.     * 参数:     无
398.     * 返回值:     string
399.     *****/
400.     virtual std::string getInstanceName() {
401.         return "OpNodeStatement";
402.     }
403.     /*****
404.     * 函数名:     getInfo
405.     * 作用:     虚函数, 打印解析树时打印自己结点的信息
406.     * 参数:     level      当前所处层次[in]
407.     * 返回值:     无
408.     *****/
409.     void getInfo(int level);
410. };
411.
412. /*****
413. * 类名:     IdNodeStatement
414. * 作用:     解析树标识符, 常量, 部分保留字结点
415. * 父类:     Statement
416. *****/
417. class IdNodeStatement :public Statement {
418. private:
419.     std::string name; // 名称

```

```

420.     LEX_TYPE lex; // 词法类型
421.     int startLine; // 起始行数
422. public:
423.     /*****
424.      * 函数名:   IdNodeStatement
425.      * 作用:    构造器
426.      * 参数:    name          名称[in]
427.      *          lex           词法类型[in]
428.      *          startLine     起始行数[in]
429.      *****/
430.     IdNodeStatement(std::string name, LEX_TYPE lex, int startLine);
431.     /*****
432.      * 函数名:   ~IdNodeStatement
433.      * 作用:    析构函数
434.      *****/
435.     ~IdNodeStatement();
436.     /*****
437.      * 函数名:   getInstanceName
438.      * 作用:    虚函数反射, 获取类名
439.      * 参数:    无
440.      * 返回值:   string
441.      *****/
442.     virtual std::string getInstanceName() {
443.         return "IdNodeStatement";
444.     }
445.     /*****
446.      * 函数名:   getInfo
447.      * 作用:    虚函数, 打印解析树时打印自己结点的信息
448.      * 参数:    level         当前所处层次[in]
449.      * 返回值:   无
450.      *****/
451.     void getInfo(int level);
452. };
453.
454. /*****
455.  * 类名:       FunctionStatement
456.  * 作用:       解析树函数调用结点
457.  * 父类:       Statement
458.  *****/
459. class FunctionStatement : public Statement {
460. private:
461.     std::string name; // 函数名
462.     std::vector<Statement*> argList; // 函数参数列表
463.     int startLine; // 起始行数
464. public:
465.     /*****
466.      * 函数名:   FunctionStatement
467.      * 作用:    构造器
468.      * 参数:    name          名称[in]
469.      *          startLine     起始行数[in]
470.      *****/
471.     FunctionStatement(std::string name, int startLine);
472.     /*****
473.      * 函数名:   ~FunctionStatement
474.      * 作用:    析构函数
475.      *****/
476.     ~FunctionStatement();
477.     /*****
478.      * 函数名:   addArg
479.      * 作用:    在参数列表中添加表达式
480.      * 参数:    arg          参数表达式[in]
481.      * 返回值:   无
482.      *****/
483.     void addArg(Statement *arg);
484.     /*****
485.      * 函数名:   getInstanceName
486.      * 作用:    虚函数反射, 获取类名
487.      * 参数:    无
488.      * 返回值:   string
489.      *****/
490.     virtual std::string getInstanceName() {
491.         return "FunctionStatement";
492.     }
493.     /*****
494.      * 函数名:   getInfo
495.      * 作用:    虚函数, 打印解析树时打印自己结点的信息
496.      * 参数:    level         当前所处层次[in]
497.      * 返回值:   无
498.      *****/
499.     void getInfo(int level);
500. };
501. #endif

```

5.3 GrammarAnalyser.h

```

1.     /*****
2.      * 模块名:   GrammarAnalyser
3.      * 文件名:   GrammarAnalyser.h
4.      * 依赖文件: Base.h, parseTree.h
5.      * 作用:    对于 C/C++ 语言进行语法分析
6.      * 作者:    陈骁飞
7.      * 时间:    2020.6.1
8.      * 版本:    Version 1.1
9.      *****/
10.    #ifndef _GRAMMARANALYSER_H_
11.    #define _GRAMMARANALYSER_H_
12.    #include "Base.h"
13.    #include "parseTree.h"
14.    #include <vector>
15.    #include <string>
16.
17.    /*****

```

```

18. * 函数名: isModifier
19. * 作用: 判断是否为修饰符类关键词
20. * 参数: word: 词语的内容[in]
21. * 返回值: bool
22. *****/
23. bool isModifier(std::string word);
24. /*****/
25. * 函数名: isType
26. * 作用: 判断是否能够定义类型
27. * 参数: lex: 词语的词法类型[in]
28. * 参数: word: 词语的内容 [in]
29. * 返回值: bool
30. *****/
31. bool isType(LEX_TYPE lex, std::string word);
32. /*****/
33. * 函数名: isProtoType
34. * 作用: 判断是否是基本类型
35. * 参数: word: 词语的内容 [in]
36. * 返回值: bool
37. *****/
38. bool isProtoType(std::string word);
39. /*****/
40. * 函数名: isClassName
41. * 作用: 判断是否是注册的类名
42. * 参数: word: 词语的内容 [in]
43. * 返回值: bool
44. *****/
45. bool isClassName(std::string word);
46. /*****/
47. * 函数名: hasTemplate
48. * 作用: 判断类名是否存在泛型（模板）
49. * 参数: word: 词语的内容 [in]
50. * 返回值: bool
51. *****/
52. bool hasTemplate(std::string word);
53. /*****/
54. * 函数名: searchForSpace
55. * 作用: 查找某一行的缩进
56. * 参数: spaceTable: 缩进表 [in]
57. * 参数: line: 行号 [in]
58. * 返回值: int
59. *****/
60. int searchForSpace(std::vector<std::pair<int, int>> spaceTable, int line);
61. /*****/
62. * 函数名: searchForLeastPriOp
63. * 作用: 查找一行优先级最低的运算符索引
64. * 参数: bin: token 容器 [in]
65. * 参数: startPos: 起始索引位置 [in]
66. * 参数: endPos: 结束索引位置 [in]
67. * 参数: splitPos: 目标运算符所在索引 [in/out]
68. * 参数: splitDir: 目标运算符方向 [in/out]
69. * (M 代表双目, L 代表左边有表达式的单目, R 代表右边有表达式的单目)
70. * 返回值: 无
71. *****/
72. void searchForLeastPriOp(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> bin, int startPos, int endPos, int &splitPos, std::string &splitDir);
73. /*****/
74. * 函数名: judgeInvalidExpression
75. * 作用: 判断表达式是否合法, 若不合法, 调用中断
76. * 参数: bin: token 容器 [in]
77. * 参数: startPos: 起始索引位置 [in]
78. * 参数: endPos: 结束索引位置 [in]
79. * 返回值: 无
80. *****/
81. void judgeInvalidExpression(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& bin, int startPos, int endPos);
82. /*****/
83. * 类名: GrammarAnalyser
84. * 作用: 进行语法分析
85. *****/
86. class GrammarAnalyser
87. {
88. public:
89. /*****/
90. * 函数名: GrammarAnalyser
91. * 作用: 构造器
92. * 参数: lexRes: 词法分析结果[in]
93. * 参数: spaceRecord 每一行的缩进[in]
94. *****/
95. GrammarAnalyser(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> lexRes, std::vector<std::pair<int, int>> spaceRecord);
96. /*****/
97. * 函数名: ~GrammarAnalyser
98. * 作用: 析构函数
99. *****/
100. ~GrammarAnalyser();
101. /*****/
102. * 函数名: grammar
103. * 作用: 开始进行语法分析
104. * 参数: 无
105. * 返回值: 无
106. *****/
107. void grammar();
108. /*****/
109. * 函数名: printParseTree
110. * 作用: 打印语法解析树
111. * 参数: 无
112. * 返回值: 无
113. *****/
114. void printParseTree();
115. private:
116. std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> lexRes; // 词法分析的结果
117. std::vector<std::pair<int, int>> spaceRecord; // 存放缩进表
118.

```

```

119.  /*****
120.  * 函数名:   init
121.  * 作用:    进行语法分析时, 分批额初始数据结构
122.  * 参数:    无
123.  * 返回值:  无
124.  *****/
125.  void init();
126.  int cur = 0; // token 的指针索引
127.  Statement* root = NULL; // 解析树的根节点
128.  Statement* curStatement = NULL; // 在构造解析树时, 当前结点
129.  /*****
130.  * 函数名:   parseComponent
131.  * 作用:    解析一个语法组件, 自顶向下的思想
132.  * 参数:    无
133.  * 返回值:  无
134.  *****/
135.  void parseComponent();
136.  /*****
137.  * 函数名:   extractType
138.  * 作用:    从 lexRes 中解析当前指针指向的类型
139.  * 参数:    type      类型输出句柄[in/out]
140.  * 返回值:  无
141.  *****/
142.  void extractType(std::string& type);
143.  /*****
144.  * 函数名:   extractType
145.  * 作用:    从 lexRes 中解析当前指针指向的类型
146.  * 参数:    type      类型输出句柄[in/out]
147.  *          tmpResBin  token 容器   [in]
148.  *          cursor    token 指针   [in/out]
149.  * 返回值:  无
150.  *****/
151.  void extractType(std::string& type, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& tmpResBin, int& cursor);
152.  /*****
153.  * 函数名:   extractArgTable
154.  * 作用:    从 lexRes 中解析函数定义时的参数列表
155.  * 参数:    argTable   参数列表输出句柄[in/out]
156.  * 返回值:  无
157.  *****/
158.  void extractArgTable(std::vector<std::pair<std::string, std::string>>& argTable);
159.  /*****
160.  * 函数名:   extractJudgement
161.  * 作用:    从 lexRes 中分离出分支语句判断表达式
162.  * 参数:    tmpResBin  判断表达式输出句柄[in/out]
163.  * 返回值:  无
164.  *****/
165.  void extractJudgement(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& tmpResBin);
166.  /*****
167.  * 函数名:   searchForCast
168.  * 作用:    查找表达式中存在的强制转型情况
169.  * 参数:    bin        token 容器[in/out]
170.  *          startPos   起始位置索引[in]
171.  *          endPos     结束位置索引[in]
172.  *          castPosStart 强制转型起始位置输出句柄[in/out]
173.  *          castPosEnd  强制转型结束位置输出句柄[in/out]
174.  *          CastType    强制转型类型输出句柄[in/out]
175.  * 返回值:  无
176.  *****/
177.  void searchForCast(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>bin, int startPos, int endPos, int& castPosStart, int& castPosEnd, std::string& CastType);
178.  /*****
179.  * 函数名:   split
180.  * 作用:    从 lexRes 中按照目标 token 进行二重语句切割
181.  * 参数:    tmpResBin   语句输出句柄[in/out]
182.  *          commaedPos  内层切割的首元素索引输出句柄[in/out]
183.  *          end        外层切割标记[in]
184.  *          mid        内层切割标记[in]
185.  * 返回值:  无
186.  *****/
187.  void split(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& tmpResBin, std::vector<int>& commaedPos, std::string end, std::string mid);
188.  /*****
189.  * 函数名:   split
190.  * 作用:    从 lexRes 中按照目标 token 进行单重语句切割
191.  * 参数:    tmpResBin   语句输出句柄[in/out]
192.  *          end        切割标记[in]
193.  * 返回值:  无
194.  *****/
195.  void split(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& tmpResBin, std::string end);
196.  /*****
197.  * 函数名:   split
198.  * 作用:    在目标容器中按照目标 token 进行二重语句切割
199.  * 参数:    src        目标 token 容器[in]
200.  *          dst        语句输出句柄[in/out]
201.  *          commaedPos  内层切割的首元素索引输出句柄[in/out]
202.  *          end        外层切割标记[in]
203.  *          mid        内层切割标记[in]
204.  *          cursor     目标容器起始指针[in/out]
205.  * 返回值:  无
206.  *****/
207.  void split(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> src, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& dst, std::vector<int>& commaedPos, std::string end, std::string mid, int& cursor);
208.  /*****
209.  * 函数名:   handleUsing
210.  * 作用:    处理 using 关键字开始语句
211.  * 参数:    无
212.  * 返回值:  无
213.  *****/
214.  void handleUsing();
215.  /*****
216.  * 函数名:   handleFunc

```

```

217. * 作用:      处理函数定义语句
218. * 参数:      无
219. * 返回值:    无
220. *****/
221. void handleFunc(std::vector<std::string> modifiers, std::string type, std::string name,int startLine);
222. /*****/
223. * 函数名:    handleVarDeclaration
224. * 作用:      处理变量声明语句
225. * 参数:      无
226. * 返回值:    无
227. *****/
228. void handleVarDeclaration(std::vector<std::string> modifiers, std::string type, std::string name, int startLine);
229. /*****/
230. * 函数名:    handleNode
231. * 作用:      自下而上处理表达式与函数调用混合语句
232. * 参数:      无
233. * 返回值:    无
234. *****/
235. void handleNode(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& bin,int startPos,int endPos,Statement *&handle);
236. /*****/
237. * 函数名:    handleEndsWithBracket
238. * 作用:      处理右大括号符号
239. * 参数:      无
240. * 返回值:    无
241. *****/
242. void handleEndsWithBracket();
243. /*****/
244. * 函数名:    handleIf
245. * 作用:      处理 if 分支
246. * 参数:      无
247. * 返回值:    无
248. *****/
249. void handleIf();
250. /*****/
251. * 函数名:    handleElse
252. * 作用:      处理 else 以及 else if 分支
253. * 参数:      无
254. * 返回值:    无
255. *****/
256. void handleElse();
257. /*****/
258. * 函数名:    handleWhile
259. * 作用:      处理 while 循环
260. * 参数:      无
261. * 返回值:    无
262. *****/
263. void handleWhile();
264. /*****/
265. * 函数名:    handleFor
266. * 作用:      处理 for 循环
267. * 参数:      无
268. * 返回值:    无
269. *****/
270. void handleFor();
271. /*****/
272. * 函数名:    handleAssign
273. * 作用:      处理普通表达式入口
274. *           (内部还有一次对于逗号的切割)
275. * 参数:      无
276. * 返回值:    无
277. *****/
278. void handleAssign();
279. /*****/
280. * 函数名:    handleSwitch
281. * 作用:      处理 switch 分支
282. * 参数:      无
283. * 返回值:    无
284. *****/
285. void handleSwitch();
286. /*****/
287. * 函数名:    handleCase
288. * 作用:      处理 case 分支
289. * 参数:      无
290. * 返回值:    无
291. *****/
292. void handleCase();
293. /*****/
294. * 函数名:    handleBreak
295. * 作用:      处理 break 语句
296. *           (内部还有一次对于逗号的切割)
297. * 参数:      无
298. * 返回值:    无
299. *****/
300. void handleBreak();
301. /*****/
302. * 函数名:    handleDefault
303. * 作用:      处理 default 分支
304. * 参数:      无
305. * 返回值:    无
306. *****/
307. void handleDefault();
308. /*****/
309. * 函数名:    handleIndentation
310. * 作用:      对缩进进行检查
311. * 参数:      无
312. * 返回值:    无
313. *****/
314. void handleIndentation();
315. /*****/
316. * 函数名:    handleReturn
317. * 作用:      处理 return 语句
318. * 参数:      无

```

```

319.     * 返回值: 无
320.     *****/
321.     void handleReturn();
322. };
323. #endif

```

5.4 GrammarAnalyser.cpp

```

1.  /*****
2.  * 模块名: GrammarAnalyser
3.  * 文件名: GrammarAnalyser.cpp
4.  * 依赖文件: Base.h, parseTree.h
5.  * 作用: 对于C/C++语言进行语法分析
6.  * 作者: 陈晓飞
7.  * 时间: 2020.6.2
8.  * 版本: Version 1.1
9.  *****/
10. #include "GrammarAnalyser.h"
11. #include "Base.h"
12. #include "parseTree.h"
13. #include <iostream>
14. using namespace std;
15. GrammarAnalyser::GrammarAnalyser(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> lexRes, std::vector<std::pair<int, int>> spaceRecord) {
16.     this->lexRes = lexRes;
17.     this->spaceRecord = spaceRecord;
18.     init();
19. }
20. GrammarAnalyser::~GrammarAnalyser() {}
21. /*****
22. * 函数名: init
23. * 作用: 进行语法分析时, 分批额初始数据结构
24. * 参数: 无
25. * 返回值: 无
26. *****/
27. void GrammarAnalyser::init() {
28.     root = new RootStatement();
29.     curStatement = root;
30. }
31. /*****
32. * 函数名: grammar
33. * 作用: 开始进行语法分析
34. * 参数: 无
35. * 返回值: 无
36. *****/
37. void GrammarAnalyser::grammar() {
38.     cout << "进行语法分析 !! " << endl;
39.     while (cur < lexRes.size()) {
40.         parseComponent();
41.     }
42. }
43. /*****
44. * 函数名: handleUsing
45. * 作用: 处理 using 关键字开始语句
46. * 参数: 无
47. * 返回值: 无
48. *****/
49. void GrammarAnalyser::handleUsing() {
50.     string type1 = lexRes[++cur].second.second;
51.     string type2 = lexRes[++cur].second.second;
52.     cur++;
53. }
54. /*****
55. * 函数名: handleFunc
56. * 作用: 处理函数定义语句
57. * 参数: 无
58. * 返回值: 无
59. *****/
60. void GrammarAnalyser::handleFunc(std::vector<std::string> modifiers, std::string type, std::string name, int startLine) {
61.     std::vector<std::pair<std::string, std::string>> argTable;
62.     extractArgTable(argTable);
63.     if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "{") {
64.         Statement* func = new FuncDefineStatement(name, type, modifiers, startLine);
65.         static_cast<FuncDefineStatement*>(func)->setArgTable(argTable);
66.         curStatement->addChild(func);
67.         curStatement = func;
68.         cur++;
69.     }
70.     else if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == ";") {
71.         cout << "函数声明" << endl;
72.         cur++;
73.     }
74.     else {
75.         cout << "函数无结束标记" << endl;
76.         abort();
77.     }
78. }
79. /*****
80. * 函数名: handleVarDeclaration
81. * 作用: 处理变量声明语句
82. * 参数: 无
83. * 返回值: 无
84. *****/
85. void GrammarAnalyser::handleVarDeclaration(std::vector<std::string> modifiers, std::string type, std::string name, int startLine) {
86.     if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == ";") {
87.         Statement* func = new VarDeclarationStatement(name, type, modifiers, startLine);
88.         curStatement->addChild(func);
89.         cur++;
90.     }
91.     else {
92.         Statement* func = new VarDeclarationStatement(name, type, modifiers, startLine);
93.         curStatement->addChild(func);
94.         cur--;

```



```

95.         std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> resTable;
96.         std::vector<int> commaedPos;
97.         split(resTable, commaedPos, ",", ",");
98.         for (int i = 0; i < commaedPos.size(); i++) {
99.             int cursor = commaedPos[i];
100.            int ends = (i == commaedPos.size() - 1) ? resTable.size() : commaedPos[i + 1];
101.            if (i > 0) {
102.                if (isModifier(resTable[commaedPos[i]].second.second)) {
103.                    modifiers.clear();
104.                    while (isModifier(resTable[cursor].second.second)) {
105.                        cursor++;
106.                    }
107.                }
108.                if (isType(resTable[commaedPos[i]].second.first, resTable[commaedPos[i]].second.second)) {
109.                    type = "";
110.                    extractType(type, resTable, cursor);
111.                }
112.                name = resTable[cursor].second.second;
113.                Statement* func = new VarDeclarationStatement(name, type, modifiers, startLine);
114.                curStatement->addChild(func);
115.            }
116.            Statement* sentence;
117.            handleNode(resTable, cursor, ends, sentence);
118.            curStatement->addChild(sentence);
119.        }
120.    }
121. }
122. /*****
123.  * 函数名:    handleNode
124.  * 作用:      自下而上处理表达式与函数调用混合语句
125.  * 参数:      无
126.  * 返回值:    无
127.  *****/
128. void GrammarAnalyser::handleNode(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& bin, int startPos, int endPos, Statement*&
handle) {
129.     judgeInvalidExpression(bin, startPos, endPos); // 进行一系列检查
130.     // 消除多余括号（最外层）
131.     if (bin[startPos].second.first == LEX_TYPE::SEPARATOR && bin[startPos].second.second == "(") {
132.         int leftBra = 1, cursor = startPos + 1;
133.         for (; cursor < endPos; cursor++) {
134.             if (bin[cursor].second.first == LEX_TYPE::SEPARATOR && bin[cursor].second.second == "(") {
135.                 leftBra++;
136.             }
137.             if (bin[cursor].second.first == LEX_TYPE::SEPARATOR && bin[cursor].second.second == ")") {
138.                 leftBra--;
139.             }
140.             if (leftBra == 0) {
141.                 break;
142.             }
143.         }
144.         if (cursor == endPos - 1) {
145.             startPos++;
146.             endPos--;
147.         }
148.     }
149.     // 搜索最惰性运算符
150.     int splitPos = -1, castPosStart = -1, castPosEnd = -1;
151.     std::string splitDir = "", CastType = "";
152.     searchForLeastPriOp(bin, startPos, endPos, splitPos, splitDir); // 搜索分裂运算符
153.     searchForCast(bin, startPos, endPos, castPosStart, castPosEnd, CastType);
154.     if (splitPos != -1) {
155.         Statement* op = new OpNodeStatement(bin[splitPos].second.second, splitDir, bin[splitPos].first);
156.         handle = op;
157.         if (OpTable[bin[splitPos].second.second + splitDir] == 0) {
158.             cout << "解析结点中运算符出现错误" << endl;
159.             abort();
160.         }
161.         else if (OpTable[bin[splitPos].second.second + splitDir] == 1 && castPosEnd != -1) { // 优先级低于强制转型
162.             Statement* st;
163.             handleNode(bin, castPosEnd + 1, endPos, st);
164.             op->addChild(st);
165.         }
166.         else {
167.             if (splitDir == "M") {
168.                 if (bin[splitPos].second.second == "[") { // 处理数组
169.                     Statement* stL, * stR;
170.                     handleNode(bin, startPos, splitPos, stL);
171.                     handleNode(bin, splitPos + 1, endPos - 1, stR);
172.                     op->addChild(stL);
173.                     op->addChild(stR);
174.                 }
175.                 else { // 非数组的运算符
176.                     Statement* stL, * stR;
177.                     handleNode(bin, startPos, splitPos, stL);
178.                     handleNode(bin, splitPos + 1, endPos, stR);
179.                     op->addChild(stL);
180.                     op->addChild(stR);
181.                 }
182.             }
183.             else if (splitDir == "L") {
184.                 Statement* stL;
185.                 handleNode(bin, startPos, splitPos, stL);
186.                 op->addChild(stL);
187.             }
188.             else if (splitDir == "R") {
189.                 Statement* stR;
190.                 handleNode(bin, splitPos + 1, endPos, stR);
191.                 op->addChild(stR);
192.             }
193.             else {
194.                 cout << "结点解析错误" << endl;
195.                 abort();
196.             }
197.         }
198.     }
199.     else if (castPosEnd != -1) {
200.         Statement* cast = new OpNodeStatement("Cast", "R", bin[castPosStart].first, CastType);

```

```

201.         handle = cast;
202.         Statement* stR;
203.         handleNode(bin, castPosEnd+1, endPos, stR);
204.         cast->addChild(stR);
205.     }
206.     else { // 处理单个结点
207.         if (endPos-startPos==1) { // 常量, 标识符结点
208.             string name = bin[startPos].second.second;
209.             LEX_TYPE lex = bin[startPos].second.first;
210.             Statement* id = new IdNodeStatement(name, lex, bin[startPos].first);
211.             handle = id;
212.         }
213.         else { // 函数结点
214.             string funcName = bin[startPos].second.second;
215.             int cursor = startPos + 1;
216.             Statement* func = new FunctionStatement(funcName, bin[startPos].first);
217.             handle = func;
218.             Statement* arg;
219.             if (bin[cursor].second.first == LEX_TYPE::SEPERATOR && bin[cursor].second.second == "(") {
220.                 cursor++;
221.                 std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> tmp;
222.                 std::vector<int> comma;
223.                 split(bin, tmp, comma, "(", " ", cursor);
224.                 for (int i = 0; i < comma.size(); i++) {
225.                     int end = (i == comma.size() - 1) ? tmp.size() : comma[i+1];
226.                     handleNode(tmp, comma[i], end, arg);
227.                     func->addChild(arg);
228.                 }
229.             }
230.             else {
231.                 cout << "解析结点出错, 误入函数调用处理逻辑" << endl;
232.                 abort();
233.             }
234.         }
235.     }
236. }
237. /*****
238. * 函数名:   handleEndsWithBracket
239. * 作用:     处理右大括号界符
240. * 参数:     无
241. * 返回值:   无
242. *****/
243. void GrammarAnalyser::handleEndsWithBracket() {
244.     int endLine = lexRes[cur].first;
245.     if (curStatement->getInstanceName()=="FuncDefineStatement") {
246.         static_cast<FuncDefineStatement*>(curStatement)->setEndLine(endLine);
247.     }
248.     else if (curStatement->getInstanceName() == "branchStatement") {
249.         static_cast<branchStatement*>(curStatement)->setEndLine(endLine);
250.     }
251.     if (this->curStatement->getParent()!=NULL) {
252.         this->curStatement = this->curStatement->getParent();
253.     }
254.     else {
255.         cout << "过多的大括号" << endl;
256.         abort();
257.     }
258. }
259. cur++;
260. }
261. /*****
262. * 函数名:   handleIf
263. * 作用:     处理 if 分支
264. * 参数:     无
265. * 返回值:   无
266. *****/
267. void GrammarAnalyser::handleIf() {
268.     int line = lexRes[cur].first;
269.     Statement* branch = new branchStatement("if", line);
270.     cur++;
271.     if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "(") {
272.         std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> resTable;
273.         extractJudgement(resTable);
274.         Statement* judgement;
275.         handleNode(resTable, 0, resTable.size(), judgement);
276.         static_cast<branchStatement*>(branch)->judgement = judgement;
277.         curStatement->addChild(branch);
278.         curStatement = branch;
279.     }
280.     else {
281.         cout << "If 循环没有结尾" << endl;
282.         abort();
283.     }
284.     if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "{") {
285.         static_cast<branchStatement*>(branch)->setHasBrack(true);
286.         cur++;
287.     }
288.     else {
289.         static_cast<branchStatement*>(branch)->setHasBrack(false);
290.     }
291. }
292. /*****
293. * 函数名:   handleElse
294. * 作用:     处理 else 以及 else if 分支
295. * 参数:     无
296. * 返回值:   无
297. *****/
298. void GrammarAnalyser::handleElse() {
299.     int line = lexRes[cur].first;
300.     cur++;
301.     if (lexRes[cur].second.first == LEX_TYPE::KEYWORD && lexRes[cur].second.second == "if") {
302.         Statement* branch = new branchStatement("else if", line);
303.         cur++;
304.         if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "(") {
305.             std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> resTable;
306.             extractJudgement(resTable);

```

```

307.         Statement* judgement;
308.         handleNode(resTable, 0, resTable.size(), judgement);
309.         static_cast<branchStatement*>(branch)->judgement = judgement;
310.         curStatement->addChild(branch);
311.         curStatement = branch;
312.     }
313.     else {
314.         cout << "If 循环没有结尾" << endl;
315.         abort();
316.     }
317.     if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "{") {
318.         static_cast<branchStatement*>(branch)->setHasBrack(true);
319.         cur++;
320.     }
321.     else {
322.         static_cast<branchStatement*>(branch)->setHasBrack(false);
323.     }
324. }
325. else if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "{") {
326.     int line = lexRes[cur].first;
327.     Statement* branch = new branchStatement("else", line);
328.     curStatement->addChild(branch);
329.     curStatement = branch;
330.     static_cast<branchStatement*>(branch)->setHasBrack(true);
331.     cur++;
332. }
333. else {
334.     int line = lexRes[cur].first;
335.     Statement* branch = new branchStatement("else", line);
336.     curStatement->addChild(branch);
337.     curStatement = branch;
338.     static_cast<branchStatement*>(branch)->setHasBrack(false);
339. }
340. }
341. /*****
342. * 函数名:   handleWhile
343. * 作用:    处理 while 循环
344. * 参数:    无
345. * 返回值:  无
346. *****/
347. void GrammarAnalyser::handleWhile() {
348.     int line = lexRes[cur].first;
349.     Statement* branch = new branchStatement("while", line);
350.     cur++;
351.     if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "(") {
352.         std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> resTable;
353.         extractJudgement(resTable);
354.         Statement* judgement;
355.         handleNode(resTable, 0, resTable.size(), judgement);
356.         static_cast<branchStatement*>(branch)->judgement = judgement;
357.         curStatement->addChild(branch);
358.         curStatement = branch;
359.     }
360.     else {
361.         cout << "While 循环没有结尾" << endl;
362.         abort();
363.     }
364.     if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "{") {
365.         static_cast<branchStatement*>(branch)->setHasBrack(true);
366.         cur++;
367.     }
368.     else {
369.         static_cast<branchStatement*>(branch)->setHasBrack(false);
370.     }
371. }
372. /*****
373. * 函数名:   handleFor
374. * 作用:    处理 for 循环
375. * 参数:    无
376. * 返回值:  无
377. *****/
378. void GrammarAnalyser::handleFor() {
379.     int line = lexRes[cur].first;
380.     Statement* branch = new branchStatement("for", line);
381.     cur++;
382.     if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "(") {
383.         curStatement->addChild(branch);
384.         curStatement = branch;
385.         std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> resTable;
386.         extractJudgement(resTable);
387.         int timesL1 = 0, timesL2 = 0;
388.         for (int i = 0; i < resTable.size(); i++) {
389.             if (resTable[i].second.first == LEX_TYPE::SEPERATOR && resTable[i].second.second == ";") {
390.                 timesL1++;
391.             }
392.             else if (resTable[i].second.first == LEX_TYPE::SEPERATOR && resTable[i].second.second == ":") {
393.                 timesL2++;
394.             }
395.         }
396.         // 处理普通 For 循环 for(int i=0,j=1;i<50;i++,j++)
397.         if (timesL1 == 2 && timesL2 == 0) {
398.             int start1 = 0, end1 = 0, start2 = 0, end2 = 0, start3 = 3, end3 = 0;
399.             int cursor = start1;
400.             while ((resTable[cursor].second.first != LEX_TYPE::SEPERATOR || resTable[cursor].second.second != ";") && cursor <= resTable.size()) {
401.                 cursor++;
402.             }
403.             end1 = cursor++;
404.             start2 = cursor;
405.             while ((resTable[cursor].second.first != LEX_TYPE::SEPERATOR || resTable[cursor].second.second != ";") && cursor <= resTable.size()) {
406.                 cursor++;
407.             }
408.             end2 = cursor++;
409.             start3 = cursor;
410.             end3 = resTable.size();
411.             if (end1 > start1 && isType(resTable[start1].second.first, resTable[start1].second.second)) {

```

```

412.         string type = "";
413.         cursor = start1;
414.         vector<string> modifiers;
415.         while (resTable[cursor].second.first == LEX_TYPE::KEYWORD && isModifier(resTable[cursor].second.second)) {
416.             modifiers.push_back(resTable[cursor++].second.second);
417.         }
418.         extractType(type, resTable, cursor);
419.         string name = "";
420.         int line = -1;
421.         if (resTable[cursor].second.first == LEX_TYPE::IDENTIFIER) {
422.             line = resTable[cursor].first;
423.             name = resTable[cursor++].second.second;
424.         }
425.         else {
426.             abort();
427.             cout << "声明类型之后的一次词不是标识符" << endl;
428.         }
429.         if (resTable[cursor].second.first == LEX_TYPE::SEPERATOR && resTable[cursor].second.second == ";") {
430.             Statement* func = new VarDeclarationStatement(name, type, modifiers, line);
431.             static_cast<branchStatement*>(branch)->init.push_back(func);
432.             cursor++;
433.         }
434.         else {
435.             Statement* func = new VarDeclarationStatement(name, type, modifiers, line);
436.             static_cast<branchStatement*>(branch)->init.push_back(func);
437.             cursor--;
438.             std::vector<int> commaedPos;
439.             int startt = cursor;
440.             for (int j = cursor; j < end1; j++) {
441.                 if (resTable[j].second.first == LEX_TYPE::SEPERATOR && resTable[j].second.second == ",") {
442.                     Statement* initSent;
443.                     handleNode(resTable, startt, j, initSent);
444.                     static_cast<branchStatement*>(branch)->init.push_back(initSent);
445.                     startt = j + 1;
446.                 }
447.             }
448.             Statement* initSent;
449.             handleNode(resTable, startt, end1, initSent);
450.             static_cast<branchStatement*>(branch)->init.push_back(initSent);
451.         }
452.     }
453.     else if (end1 > start1) {
454.         Statement* initSent;
455.         handleNode(resTable, start1, end1, initSent);
456.         static_cast<branchStatement*>(branch)->init.push_back(initSent);
457.     }
458.     if (end2 > start2 && isType(resTable[start2].second.first, resTable[start2].second.second)) {
459.         cout << "for 循环的继续条件不能为声明语句" << endl;
460.         abort();
461.     }
462.     else if (end2 > start2) {
463.         Statement* endSent;
464.         handleNode(resTable, start2, end2, endSent);
465.         static_cast<branchStatement*>(branch)->end = endSent;
466.     }
467.     if (end3 > start3 && isType(resTable[start3].second.first, resTable[start3].second.second)) {
468.         string type = "";
469.         cursor = start3;
470.         vector<string> modifiers;
471.         while (resTable[cursor].second.first == LEX_TYPE::KEYWORD && isModifier(resTable[cursor].second.second)) {
472.             modifiers.push_back(resTable[cursor++].second.second);
473.         }
474.         extractType(type, resTable, cursor);
475.         string name = "";
476.         int line = -1;
477.         if (resTable[cursor].second.first == LEX_TYPE::IDENTIFIER) {
478.             line = resTable[cursor].first;
479.             name = resTable[cursor++].second.second;
480.         }
481.         else {
482.             abort();
483.             cout << "声明类型之后的一次词不是标识符" << endl;
484.         }
485.         if (resTable[cursor].second.first == LEX_TYPE::SEPERATOR && resTable[cursor].second.second == ";") {
486.             Statement* func = new VarDeclarationStatement(name, type, modifiers, line);
487.             static_cast<branchStatement*>(branch)->init.push_back(func);
488.             cursor++;
489.         }
490.         else {
491.             Statement* func = new VarDeclarationStatement(name, type, modifiers, line);
492.             static_cast<branchStatement*>(branch)->init.push_back(func);
493.             cursor--;
494.             std::vector<int> commaedPos;
495.             int startt = cursor;
496.             for (int j = cursor; j < end3; j++) {
497.                 if (resTable[j].second.first == LEX_TYPE::SEPERATOR && resTable[j].second.second == ",") {
498.                     Statement* iterSent;
499.                     handleNode(resTable, startt, j, iterSent);
500.                     static_cast<branchStatement*>(branch)->iterate.push_back(iterSent);
501.                     startt = j + 1;
502.                 }
503.             }
504.             Statement* iterSent;
505.             handleNode(resTable, startt, end3, iterSent);
506.             static_cast<branchStatement*>(branch)->iterate.push_back(iterSent);
507.         }
508.     }
509.     else if (end3 > start3) {
510.         Statement* iterSent;
511.         handleNode(resTable, start3, end3, iterSent);
512.         static_cast<branchStatement*>(branch)->iterate.push_back(iterSent);
513.     }
514. }
515. // 处理增强 for 循环 for(auto i : list)
516. else if (timesL1 == 0 && timesL2 == 1) {
517.     int cursor = -1;
518.     for (int i = 0; i < resTable.size(); i++) {
519.         if (resTable[i].second.first == LEX_TYPE::SEPERATOR && resTable[i].second.second == ":") {

```

```

520.         cursor = i;
521.     }
522.     }
523.     if (isType(resTable[0].second.first, resTable[0].second.second)) {
524.         string type = "";
525.         int cursor1 = 0;
526.         vector<string> modifiers;
527.         while (resTable[cursor1].second.first == LEX_TYPE::KEYWORD && isModifier(resTable[cursor1].second.second)) {
528.             modifiers.push_back(resTable[cursor1++].second.second);
529.         }
530.         extractType(type, resTable, cursor1);
531.         string name = "";
532.         int line = -1;
533.         if (resTable[cursor1].second.first == LEX_TYPE::IDENTIFIER) {
534.             line = resTable[cursor1].first;
535.             name = resTable[cursor1].second.second;
536.         }
537.         else {
538.             abort();
539.             cout << "声明类型之后的一次词不是标识符" << endl;
540.         }
541.         Statement* func = new VarDeclarationStatement(name, type, modifiers, line);
542.         static_cast<branchStatement*>(branch)->forEachDeclare.push_back(func);
543.     }
544.     else {
545.         cout << "ForEach 循环格式错误" << endl;
546.     }
547.     Statement* forEachSent;
548.     handleNode(resTable, cursor + 1, resTable.size(), forEachSent);
549.     static_cast<branchStatement*>(branch)->forEachDeclare.push_back(forEachSent);
550. }
551. else {
552.     cout << "For 循环格式错误" << endl;
553.     abort();
554. }
555. }
556. else {
557.     cout << "For 循环没有结尾" << endl;
558.     abort();
559. }
560. if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "{") {
561.     static_cast<branchStatement*>(branch)->setHasBrack(true);
562.     cur++;
563. }
564. else {
565.     static_cast<branchStatement*>(branch)->setHasBrack(false);
566. }
567. }
568. /*****
569. * 函数名:   handleAssign
570. * 作用:     处理普通表达式入口
571. *          （内部还有一次对于逗号的切割）
572. * 参数:     无
573. * 返回值:   无
574. *****/
575. void GrammarAnalyser::handleAssign() {
576.     std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> resTable;
577.     std::vector<int> commaedPos;
578.     split(resTable, commaedPos, ";", ",", ",");
579.     for (int i = 0; i < commaedPos.size(); i++) {
580.         int ends = (i == commaedPos.size() - 1) ? resTable.size() : commaedPos[i + 1];
581.         Statement* sent;
582.         handleNode(resTable, commaedPos[i], ends, sent);
583.         curStatement->addChild(sent);
584.     }
585. }
586. /*****
587. * 函数名:   handleSwitch
588. * 作用:     处理 switch 分支
589. * 参数:     无
590. * 返回值:   无
591. *****/
592. void GrammarAnalyser::handleSwitch() {
593.     int line = lexRes[cur].first;
594.     Statement* branch = new branchStatement("switch", line);
595.     cur++;
596.     if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "(") {
597.         std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> resTable;
598.         extractJudgement(resTable);
599.         Statement* judgement;
600.         handleNode(resTable, 0, resTable.size(), judgement);
601.         static_cast<branchStatement*>(branch)->judgement = judgement;
602.         curStatement->addChild(branch);
603.         curStatement = branch;
604.     }
605.     else {
606.         cout << "Switch 循环没有结尾" << endl;
607.         abort();
608.     }
609.     if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "{") {
610.         static_cast<branchStatement*>(branch)->setHasBrack(true);
611.         cur++;
612.     }
613.     else {
614.         static_cast<branchStatement*>(branch)->setHasBrack(false);
615.     }
616. }
617. /*****
618. * 函数名:   handleCase
619. * 作用:     处理 case 分支
620. * 参数:     无
621. * 返回值:   无
622. *****/
623. void GrammarAnalyser::handleCase() {
624.     int line = lexRes[cur].first;
625.     Statement* branch = new branchStatement("case", line);

```

```

626.     static_cast<branchStatement*>(branch)->setHasBrack(false);
627.     cur++;
628.     std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> m;
629.     m.push_back(lexRes[cur++]);
630.     if (lexRes[cur].second.first == LEX_TYPE::OPERATOR && lexRes[cur].second.second == "::-") {
631.         m.push_back(lexRes[cur++]);
632.         m.push_back(lexRes[cur++]);
633.     }
634.     Statement* judgement;
635.     handleNode(m, 0, m.size(), judgement);
636.     static_cast<branchStatement*>(branch)->judgement = judgement;
637.     if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "::-") {
638.         curStatement->addChild(branch);
639.         curStatement = branch;
640.         cur++;
641.     }
642.     else {
643.         cout << "Case 的格式错误:后面只能跟随一个常量" << endl;
644.         abort();
645.     }
646. }
647. /*****
648.  * 函数名:   handleBreak
649.  * 作用:     处理 break 语句
650.  *          (内部还有一次对于逗号的切割)
651.  * 参数:     无
652.  * 返回值:   无
653.  *****/
654. void GrammarAnalyser::handleBreak() {
655.     int line = lexRes[cur].first;
656.     Statement* breakst = new breakStatement(line);
657.     curStatement->addChild(breakst);
658.     cur++;
659. }
660. /*****
661.  * 函数名:   handleDefault
662.  * 作用:     处理 default 分支
663.  * 参数:     无
664.  * 返回值:   无
665.  *****/
666. void GrammarAnalyser::handleDefault() {
667.     int line = lexRes[cur].first;
668.     Statement* branch = new branchStatement("default", line);
669.     static_cast<branchStatement*>(branch)->setHasBrack(false);
670.     cur++;
671.     if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "::-") {
672.         curStatement->addChild(branch);
673.         curStatement = branch;
674.         cur++;
675.     }
676.     else {
677.         cout << "Default 的格式错误:无冒号" << endl;
678.         abort();
679.     }
680. }
681. /*****
682.  * 函数名:   handleReturn
683.  * 作用:     处理 return 语句
684.  * 参数:     无
685.  * 返回值:   无
686.  *****/
687. void GrammarAnalyser::handleReturn() {
688.     int line = lexRes[cur++].first;
689.     Statement* st = new returnStatement(line);
690.     curStatement->addChild(st);
691.     std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> resTable;
692.     split(resTable, ",");
693.     if (resTable.size() > 0) {
694.         Statement* sent;
695.         handleNode(resTable, 0, resTable.size(), sent);
696.         st->addChild(sent);
697.     }
698. }
699. /*****
700.  * 函数名:   handleIndentation
701.  * 作用:     对缩进进行检查
702.  * 参数:     无
703.  * 返回值:   无
704.  *****/
705. void GrammarAnalyser::handleIndentation() {
706.     if (curStatement->getInstanceName() == "branchStatement") {
707.         if (static_cast<branchStatement*>(curStatement)->getHasBrack()==false) {
708.             int branchLine = static_cast<branchStatement*>(curStatement)->getStartLine();
709.             int branchIndentation = searchForSpace(spaceRecord, branchLine);
710.             int curLine = lexRes[cur].first;
711.             int curIndentation = searchForSpace(spaceRecord, curLine);
712.             if (curIndentation <= branchIndentation) {
713.                 curStatement = curStatement->getParent();
714.             }
715.         }
716.     }
717. }
718.
719. /*****
720.  * 函数名:   parseComponent
721.  * 作用:     解析一个语法组件, 自顶向下的思想
722.  * 参数:     无
723.  * 返回值:   无
724.  *****/
725. void GrammarAnalyser::parseComponent() {
726.     vector<string> modifiers;
727.     handleIndentation();
728.     while (lexRes[cur].second.first == LEX_TYPE::KEYWORD && isModifier(lexRes[cur].second.second)) {
729.         modifiers.push_back(lexRes[cur++].second.second);
730.     }
731.     if (lexRes[cur].second.first == LEX_TYPE::KEYWORD && lexRes[cur].second.second == "using") {

```



```

732.     handleUsing();
733. }
734. else if (isType(lexRes[cur].second.first, lexRes[cur].second.second)) {
735.     string type = "";
736.     extractType(type);
737.     string name = "";
738.     int line = -1;
739.     if (lexRes[cur].second.first == LEX_TYPE::IDENTIFIER) {
740.         line = lexRes[cur].first;
741.         name = lexRes[cur++].second.second;
742.     }
743.     else {
744.         cout << "声明类型之后的一次词不是标识符" << endl;
745.         abort();
746.     }
747.     if (lexRes[cur].second.first == LEX_TYPE::SEPARATOR && lexRes[cur].second.second == "(") {
748.         handleFunc(modifiers, type, name, line);
749.     }
750.     else {
751.         handleVarDeclaration(modifiers, type, name, line);
752.     }
753. }
754. else if (lexRes[cur].second.first == LEX_TYPE::SEPARATOR && lexRes[cur].second.second == "}") {
755.     handleEndsWithBracket();
756. }
757. else if (lexRes[cur].second.first == LEX_TYPE::KEYWORD && lexRes[cur].second.second == "if") {
758.     handleIf();
759. }
760. else if (lexRes[cur].second.first == LEX_TYPE::KEYWORD && lexRes[cur].second.second == "else") {
761.     handleElse();
762. }
763. else if (lexRes[cur].second.first == LEX_TYPE::KEYWORD && lexRes[cur].second.second == "while") {
764.     handleWhile();
765. }
766. else if (lexRes[cur].second.first == LEX_TYPE::KEYWORD && lexRes[cur].second.second == "for") {
767.     handleFor();
768. }
769. else if (lexRes[cur].second.first == LEX_TYPE::KEYWORD && lexRes[cur].second.second == "switch") {
770.     handleSwitch();
771. }
772. else if (lexRes[cur].second.first == LEX_TYPE::KEYWORD && lexRes[cur].second.second == "case") {
773.     handleCase();
774. }
775. else if (lexRes[cur].second.first == LEX_TYPE::KEYWORD && lexRes[cur].second.second == "break") {
776.     handleBreak();
777. }
778. else if (lexRes[cur].second.first == LEX_TYPE::KEYWORD && lexRes[cur].second.second == "default") {
779.     handleDefault();
780. }
781. else if (lexRes[cur].second.first == LEX_TYPE::IDENTIFIER || lexRes[cur].second.first == LEX_TYPE::CONST || lexRes[cur].second.first == LEX_TYPE::OPERATOR || (lexRes[cur].second.first == LEX_TYPE::SEPARATOR && lexRes[cur].second.second == "(")) {
782.     handleAssign();
783. }
784. else if (lexRes[cur].second.first == LEX_TYPE::KEYWORD && lexRes[cur].second.second == "return") {
785.     handleReturn();
786. }
787. else {
788.     cout << "Others " << lexRes[cur].second.second << endl;
789. }
790. if (cur < lexRes.size() && lexRes[cur].second.first == LEX_TYPE::SEPARATOR && lexRes[cur].second.second == ";") {
791.     cur++;
792. }
793. }
794. /*****
795. * 函数名: printParseTree
796. * 作用: 打印语法解析树
797. * 参数: 无
798. * 返回值: 无
799. *****/
800. void GrammarAnalyser::printParseTree() {
801.     cout << "以下打印语法解析树: " << endl;
802.     root->getInfo(0);
803. }
804. /*****
805. * 函数名: extractType
806. * 作用: 从 lexRes 中解析当前指针指向的类型
807. * 参数: type 类型输出句柄[in/out]
808. * 返回值: 无
809. *****/
810. void GrammarAnalyser::extractType(string& type) {
811.     type += lexRes[cur++].second.second;
812.     if (hasTemplate(type) && lexRes[cur].second.first == LEX_TYPE::SEPARATOR && lexRes[cur].second.second == "<") {
813.         type += lexRes[cur++].second.second;
814.         int leftBra = 1;
815.         while (leftBra > 0 && lexRes[cur].second.second != ";") {
816.             if (lexRes[cur].second.first == LEX_TYPE::SEPARATOR && lexRes[cur].second.second == "<") {
817.                 leftBra++;
818.             }
819.             if (lexRes[cur].second.first == LEX_TYPE::SEPARATOR && lexRes[cur].second.second == ">") {
820.                 leftBra--;
821.             }
822.             type += lexRes[cur++].second.second;
823.         }
824.     }
825.     while (lexRes[cur].second.first == LEX_TYPE::OPERATOR && lexRes[cur].second.second == "=") {
826.         type += lexRes[cur++].second.second;
827.     }
828. }
829. /*****
830. * 函数名: extractType
831. * 作用: 从 lexRes 中解析当前指针指向的类型
832. * 参数: type 类型输出句柄[in/out]
833. * tmpResBin token 容器 [in]
834. * cursor token 指针 [in/out]
835. * 返回值: 无
836. *****/

```

```

837. void GrammarAnalyser::extractType(std::string& type, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>& tmpResBin, int &cursor)
838. {
839.     type += tmpResBin[cursor++].second.second;
840.     if (hasTemplate(type) && tmpResBin[cursor].second.first == LEX_TYPE::SEPERATOR && tmpResBin[cursor].second.second == "<") {
841.         type += tmpResBin[cursor++].second.second;
842.         int leftBra = 1;
843.         while (leftBra > 0 && tmpResBin[cursor].second.second != ";") {
844.             if (tmpResBin[cursor].second.first == LEX_TYPE::SEPERATOR && tmpResBin[cursor].second.second == "<") {
845.                 leftBra++;
846.             }
847.             if (tmpResBin[cursor].second.first == LEX_TYPE::SEPERATOR && tmpResBin[cursor].second.second == ">") {
848.                 leftBra--;
849.             }
850.             type += tmpResBin[cursor++].second.second;
851.         }
852.         while (tmpResBin[cursor].second.first == LEX_TYPE::OPERATOR && tmpResBin[cursor].second.second == "=") {
853.             type += tmpResBin[cursor++].second.second;
854.         }
855.     }
856.     /*****
857.     * 函数名:    extractArgTable
858.     * 作用:      从 lexRes 中解析函数定义时的参数列表
859.     * 参数:      argTable    参数列表输出句柄[in/out]
860.     * 返回值:    无
861.     *****/
862. void GrammarAnalyser::extractArgTable(std::vector<std::pair<std::string, std::string>>& argTable) {
863.     cur++;
864.     while (isType(lexRes[cur].second.first, lexRes[cur].second.second)) {
865.         string type = "";
866.         extractType(type);
867.         string name = lexRes[cur++].second.second;
868.         argTable.push_back(pair<std::string, std::string>(type, name));
869.         if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == ",") {
870.             cur++;
871.         }
872.     }
873.     cur++;
874. }
875. /*****
876. * 函数名:    extractJudgement
877. * 作用:      从 lexRes 中分离出分支语句判断表达式
878. * 参数:      tmpResBin    判断表达式输出句柄[in/out]
879. * 返回值:    无
880. *****/
881. void GrammarAnalyser::extractJudgement(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>& tmpResBin) {
882.     int leftBra = 1;
883.     cur++;
884.     while (leftBra > 0 && cur < lexRes.size()) {
885.         if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "(") {
886.             leftBra++;
887.         }
888.         if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == ")") {
889.             leftBra--;
890.         }
891.         tmpResBin.push_back(lexRes[cur++]);
892.     }
893.     if (tmpResBin.size() > 0) {
894.         tmpResBin.pop_back();
895.     }
896.     else {
897.         cout << "程序结尾有一个小括号无配对" << endl;
898.         abort();
899.     }
900. }
901. /*****
902. * 函数名:    split
903. * 作用:      从 lexRes 中按照目标 token 进行二重语句切割
904. * 参数:      tmpResBin    语句输出句柄[in/out]
905. *           commaedPos    内层切割的首元素索引输出句柄[in/out]
906. *           end            外层切割标记[in]
907. *           mid            内层切割标记[in]
908. * 返回值:    无
909. *****/
910. void GrammarAnalyser::split(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>& tmpResBin, std::vector<int>& commaedPos, std::string end, std::string mid) {
911.     commaedPos.push_back(0);
912.     int leftbra = 0;
913.     while (lexRes[cur].second.first != LEX_TYPE::SEPERATOR || lexRes[cur].second.second != end) {
914.         if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == "(") {
915.             leftbra++;
916.         }
917.         if (lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == ")") {
918.             leftbra--;
919.         }
920.         if (leftbra == 0 && lexRes[cur].second.first == LEX_TYPE::SEPERATOR && lexRes[cur].second.second == mid) {
921.             cur++;
922.             commaedPos.push_back(tmpResBin.size());
923.         }
924.         else {
925.             tmpResBin.push_back(lexRes[cur++]);
926.         }
927.     }
928.     if (tmpResBin.size() == 0) {
929.         commaedPos.pop_back();
930.     }
931. }
932. /*****
933. * 函数名:    split
934. * 作用:      在目标容器中按照目标 token 进行二重语句切割
935. * 参数:      src          目标 token 容器[in]
936. *           dst            语句输出句柄[in/out]
937. *           commaedPos    内层切割的首元素索引输出句柄[in/out]
938. *           end            外层切割标记[in]
939. *           mid            内层切割标记[in]
940. *           cursor        目标容器起始指针[in/out]

```

```

941. * 返回值: 无
942. *****/
943. void GrammarAnalyser::split(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> src, std::vector<std::pair<int, std::pair<LEX_TY
PE, std::string>>>& dst, std::vector<int>& commaedPos, std::string end, std::string mid, int &cursor) {
944.     commaedPos.push_back(0);
945.     int leftbra = 0;
946.     while (leftbra != 0 || src[cursor].second.first != LEX_TYPE::SEPERATOR || src[cursor].second.second != end) {
947.         if (src[cursor].second.first == LEX_TYPE::SEPERATOR && src[cursor].second.second == "(") {
948.             leftbra++;
949.         }
950.         if (src[cursor].second.first == LEX_TYPE::SEPERATOR && src[cursor].second.second == ")") {
951.             leftbra--;
952.         }
953.         if (leftbra == 0 && src[cursor].second.first == LEX_TYPE::SEPERATOR && src[cursor].second.second == mid) {
954.             cursor++;
955.             commaedPos.push_back(dst.size());
956.         }
957.         else {
958.             dst.push_back(src[cursor++]);
959.         }
960.     }
961.     if (dst.size() == 0) {
962.         commaedPos.pop_back();
963.     }
964. }
965. *****/
966. * 函数名: split
967. * 作用: 从 lexRes 中按照目标 token 进行单重语句切割
968. * 参数: tmpResBin 语句输出句柄[in/out]
969. * 返回值: 无
970. *****/
971. void GrammarAnalyser::split(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& tmpResBin, std::string end) {
972.     while (lexRes[cur].second.first != LEX_TYPE::SEPERATOR || lexRes[cur].second.second != end) {
973.         tmpResBin.push_back(lexRes[cur++]);
974.     }
975. }
976. *****/
977. * 函数名: judgeInvalidExpression
978. * 作用: 判断表达式是否合法, 若不合法, 调用中断
979. * 参数: bin: token 容器 [in]
980. * startPos: 起始索引位置 [in]
981. * endPos: 结束索引位置 [in]
982. * 返回值: 无
983. *****/
984. void judgeInvalidExpression(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& bin, int startPos, int endPos) {
985.     // [1] 检查是否为空解析
986.     if (startPos == endPos) {
987.         cout << "存在空子结点" << endl;
988.         abort();
989.     }
990.     // [2] 检查是否存在非法字符
991.     bool flag = false;
992.     for (int cursor = startPos; cursor < endPos; cursor++) {
993.         if (bin[cursor].second.first == LEX_TYPE::SEPERATOR && bin[cursor].second.second == ";") {
994.             flag = true;
995.         }
996.     }
997.     if (flag) {
998.         cout << "存在非法字符" << endl;
999.         abort();
1000.     }
1001.     // 检查是否存在名词性常量冗余 (运算符缺省)
1002.     if (endPos - startPos >= 2) {
1003.         for (int cursor = startPos; cursor < endPos; cursor++) {
1004.             if (bin[cursor].second.first == LEX_TYPE::SEPERATOR || bin[cursor].second.first == LEX_TYPE::OPERATOR) {
1005.                 flag = true;
1006.             }
1007.         }
1008.         if (flag == false) {
1009.             cout << "存在名词性常量冗余" << endl;
1010.             abort();
1011.         }
1012.     }
1013.     // 检查括号是否匹配
1014.     int leftBra = 0;
1015.     for (int cursor = startPos; cursor < endPos; cursor++) {
1016.         if (bin[cursor].second.first == LEX_TYPE::SEPERATOR && bin[cursor].second.second == "(") {
1017.             leftBra++;
1018.         }
1019.         if (bin[cursor].second.first == LEX_TYPE::SEPERATOR && bin[cursor].second.second == ")") {
1020.             leftBra--;
1021.         }
1022.     }
1023.     if (leftBra != 0) {
1024.         cout << "括号不匹配" << endl;
1025.         abort();
1026.     }
1027. }
1028. *****/
1029. * 函数名: searchForSpace
1030. * 作用: 查找某一行的缩进
1031. * 参数: spaceTable: 缩进表 [in]
1032. * line: 行号 [in]
1033. * 返回值: int
1034. *****/
1035. int searchForSpace(std::vector<std::pair<int, int>> spaceTable, int line) {
1036.     for (auto it: spaceTable) {
1037.         if (it.first == line) {
1038.             return it.second;
1039.         }
1040.     }
1041.     return -1;
1042. }
1043. *****/
1044.

```

```

1045. * 函数名: searchForLeastPriOp
1046. * 作用: 查找一行优先级最低的运算符索引
1047. * 参数: bin: token 容器 [in]
1048. * startPos: 起始索引位置 [in]
1049. * endPos: 结束索引位置 [in]
1050. * splitPos: 目标运算符所在索引 [in/out]
1051. * splitDir: 目标运算符方向 [in/out]
1052. * (M 代表双目, L 代表左边有表达式的单目, R 代表右边有表达式的单目)
1053. * 返回值: 无
1054. *****/
1055. void searchForLeastPriOp(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> bin, int startPos, int endPos, int &splitPos, std::string &splitDir) {
1056.     auto isNoun = [](int pos, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> bin)->bool {
1057.         return (bin[pos].second.first == LEX_TYPE::CONST) || (bin[pos].second.first == LEX_TYPE::IDENTIFIER) || (bin[pos].second.first == LEX_TYPE::KEYWORD);
1058.     };
1059.     auto isLeftBrack = [](int pos, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> bin)->bool {
1060.         return (bin[pos].second.first == LEX_TYPE::SEPARATOR) && (bin[pos].second.second == "(");
1061.     };
1062.     auto isRightBrack = [](int pos, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> bin)->bool {
1063.         return (bin[pos].second.first == LEX_TYPE::SEPARATOR) && (bin[pos].second.second == ")");
1064.     };
1065.     auto isRightSquareBrack = [](int pos, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> bin)->bool {
1066.         return (bin[pos].second.first == LEX_TYPE::SEPARATOR) && (bin[pos].second.second == "]");
1067.     };
1068.     auto isLeftSquareBrack = [](int pos, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> bin)->bool {
1069.         return (bin[pos].second.first == LEX_TYPE::SEPARATOR) && (bin[pos].second.second == "[");
1070.     };
1071.     auto isColon = [](int pos, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> bin)->bool {
1072.         return (bin[pos].second.first == LEX_TYPE::SEPARATOR) && (bin[pos].second.second == ":");
1073.     };
1074.     int pos = -1;
1075.     int maxPri = -1;
1076.     std::string dir = "";
1077.     int leftBra = 0;
1078.     for (int cursor = startPos; cursor < endPos; cursor++) {
1079.         if (leftBra==0 && (bin[cursor].second.first == LEX_TYPE::OPERATOR || isLeftSquareBrack(cursor, bin) || isColon(cursor, bin))) {
1080.             if ((cursor > startPos && (isNoun(cursor-1, bin) || isRightBrack(cursor-1, bin) || isRightSquareBrack(cursor-1, bin))) && (cursor < endPos-1 && (isNoun(cursor+1, bin) || isLeftBrack(cursor+1, bin)))) {
1081.                 if (OpTable[bin[cursor].second.second + "M"] > maxPri) {
1082.                     maxPri = OpTable[bin[cursor].second.second + "M"];
1083.                     pos = cursor;
1084.                     dir = "M";
1085.                 }
1086.             }
1087.             else if (cursor > startPos && (isNoun(cursor-1, bin) || isRightBrack(cursor-1, bin))) {
1088.                 if (OpTable[bin[cursor].second.second + "L"] > maxPri) {
1089.                     maxPri = OpTable[bin[cursor].second.second + "L"];
1090.                     pos = cursor;
1091.                     dir = "L";
1092.                 }
1093.             }
1094.             else {
1095.                 if (OpTable[bin[cursor].second.second + "R"] > maxPri) {
1096.                     maxPri = OpTable[bin[cursor].second.second + "R"];
1097.                     pos = cursor;
1098.                     dir = "R";
1099.                 }
1100.             }
1101.         }
1102.         if (isLeftBrack(cursor, bin) || isLeftSquareBrack(cursor, bin)) {
1103.             leftBra++;
1104.         }
1105.         if (isRightBrack(cursor, bin) || isRightSquareBrack(cursor, bin)) {
1106.             leftBra--;
1107.         }
1108.     }
1109.     splitPos = pos;
1110.     splitDir = dir;
1111. }
1112. *****/
1113. * 函数名: searchForCast
1114. * 作用: 查找表达式中存在的强制转型情况
1115. * 参数: bin token 容器 [in/out]
1116. * startPos 起始位置索引 [in]
1117. * endPos 结束位置索引 [in]
1118. * castPosStart 强制转型起始位置输出句柄 [in/out]
1119. * castPosEnd 强制转型结束位置输出句柄 [in/out]
1120. * CastType 强制转型类型输出句柄 [in/out]
1121. * 返回值: 无
1122. *****/
1123. void GrammarAnalyser::searchForCast(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> bin, int startPos, int endPos, int &castPosStart, int &castPosEnd, std::string &CastType) {
1124.     auto isLeftBrack = [](int pos, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> bin)->bool {
1125.         return (bin[pos].second.first == LEX_TYPE::SEPARATOR) && (bin[pos].second.second == "(");
1126.     };
1127.     auto isRightBrack = [](int pos, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> bin)->bool {
1128.         return (bin[pos].second.first == LEX_TYPE::SEPARATOR) && (bin[pos].second.second == ")");
1129.     };
1130.     auto isRightSquareBrack = [](int pos, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> bin)->bool {
1131.         return (bin[pos].second.first == LEX_TYPE::SEPARATOR) && (bin[pos].second.second == "]");
1132.     };
1133.     auto isLeftSquareBrack = [](int pos, std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> bin)->bool {
1134.         return (bin[pos].second.first == LEX_TYPE::SEPARATOR) && (bin[pos].second.second == "[");
1135.     };
1136.     int leftBra = 0;
1137.     for (int cursor = startPos; cursor < endPos; cursor++) {
1138.         if (leftBra==0 && isLeftBrack(cursor, bin)) {
1139.             castPosStart = cursor;
1140.             leftBra++;
1141.             cursor++;
1142.             string type = "";
1143.             while (bin[cursor].second.first == LEX_TYPE::KEYWORD && isModifier(bin[cursor].second.second)) {
1144.                 type += bin[cursor].second.second;
1145.                 cursor++;
1146.             }

```

```

1147.         if (isType(bin[cursor].second.first, bin[cursor].second.second)) {
1148.             extractType(type, bin, cursor);
1149.             if (isRightBranch(cursor, bin)) {
1150.                 CastPosEnd = (cursor++);
1151.                 CastType = type;
1152.                 return;
1153.             }
1154.         }
1155.     }
1156.     if (isLeftSquareBranch(cursor, bin)) {
1157.         leftBra++;
1158.     }
1159.     if (isRightBranch(cursor, bin) || isRightSquareBranch(cursor, bin)) {
1160.         leftBra--;
1161.     }
1162. }
1163. }
1164. /*****
1165.  * 函数名:  isModifier
1166.  * 作用:    判断是否为修饰符类关键词
1167.  * 参数:    word:    词语的内容[in]
1168.  * 返回值:  bool
1169.  *****/
1170. bool isModifier(string word) {
1171.     return (word == "inline") || (word == "public") || (word == "static") || (word == "protected") || (word == "private") || (word == "unsigned");
1172. }
1173. /*****
1174.  * 函数名:  isProtoType
1175.  * 作用:    判断是否是基本类型
1176.  * 参数:    word:    词语的内容    [in]
1177.  * 返回值:  bool
1178.  *****/
1179. bool isProtoType(string word) {
1180.     return (word == "void") || (word == "int") || (word == "char") || (word == "bool") || (word == "double") || (word == "float") || (word == "long") || (word == "auto") || (word == "unsigned");
1181. }
1182. /*****
1183.  * 函数名:  isClassName
1184.  * 作用:    判断是否是注册的类型名
1185.  * 参数:    word:    词语的内容    [in]
1186.  * 返回值:  bool
1187.  *****/
1188. bool isClassName(string word) {
1189.     for (auto it : existClassName) {
1190.         if (it == word) {
1191.             return true;
1192.         }
1193.     }
1194.     return false;
1195. }
1196. /*****
1197.  * 函数名:  isType
1198.  * 作用:    判断是否能够定义类型
1199.  * 参数:    lex:    词语的词法类型[in]
1200.  * 参数:    word:    词语的内容    [in]
1201.  * 返回值:  bool
1202.  *****/
1203. bool isType(LEX_TYPE lex, string word) {
1204.     return ((lex == LEX_TYPE::KEYWORD) && isProtoType(word)) || ((lex == LEX_TYPE::IDENTIFIER) && (isClassName(word)));
1205. }
1206. /*****
1207.  * 函数名:  hasTemplate
1208.  * 作用:    判断类型是否存在泛型（模板）
1209.  * 参数:    word:    词语的内容    [in]
1210.  * 返回值:  bool
1211.  *****/
1212. bool hasTemplate(string word) {
1213.     for (auto it : existTemplate) {
1214.         if (it == word) {
1215.             return true;
1216.         }
1217.     }
1218.     return false;
1219. }

```