

**课程设计报告**

**题目：基于高级语言源程序格式处理工具**

**课程名称：程序设计综合课程设计**

**专业班级： CS1908**

**学 号： U201990067**

**姓 名： 胡嘉慧**

**指导教师： 袁凌**

**报告日期： 2021年3月22日**

**计算机科学与技术学**

任 务 书

设计内容

抽象语法树(Abstract Syntax Tree ,AST)作为程序的一种中间表示形式,在[程序分析](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E5%88%86%E6%9E%90" \t "_blank)等诸多领域有广泛的应用.利用抽象语法树可以方便地实现多种[源程序](https://baike.baidu.com/item/%E6%BA%90%E7%A8%8B%E5%BA%8F" \t "_blank)处理工具,比如源程序浏览器、智能编辑器、[语言翻译器](https://baike.baidu.com/item/%E8%AF%AD%E8%A8%80%E7%BF%BB%E8%AF%91%E5%99%A8/6119005" \t "_blank)等。

在《高级语言源程序格式处理工具》这个题目中，首先需要采用形式化的方式，使用巴克斯（BNF）范式定义高级语言的词法规则（字符组成单词的规则）、语法规则（单词组成语句、程序等的规则）。再利用形式语言自动机的的原理，对源程序的文件进行词法分析，识别出所有单词；使用编译技术中的递归下降语法分析法，分析源程序的语法结构，并生成抽象语法树,最后可由抽象语法树生成格式化的源程序。

设计要求

要求具有如下功能：

**1. 语言定义**

选定C语言的一个子集，要求包含：

（1）基本数据类型的变量、常量，以及数组。不包含指针、结构，枚举等。

（2) 双目算术运算符（+-\*/%），关系运算符、逻辑与（&&）、逻辑或（||）、赋值运算符。不包含逗号运算符、位运算符、各种单目运算符等等。

（3）函数定义、声明与调用。

（4）表达式语句、复合语句、if语句的2种形式、while语句、for语句，return语句、break语句、continue语句、外部变量说明语句、局部变量说明语句。

（5）编译预处理（宏定义，文件包含）

（6）注释（块注释与行注释）

**2. 单词识别**

设计DFA的状态转换图（参见实验指导），实验时给出DFA，并解释如何在状态迁移中完成单词识别（每个单词都有一个种类编号和单词的字符串这2个特征值），最终生成单词识别（词法分析）子程序。

**3. 语法结构分析**

（1）外部变量的声明；

（2）函数声明与定义；

（3）局部变量的声明；

（4）语句及表达式；

（5）生成（1)-(4)（包含编译预处理和注释）的抽象语法树并显示。

**4. 按缩进编排生成源程序文件。**

参考文献

[1] 王生原，董渊，张素琴，吕映芝等. 编译原理（第3版）. 北京：清华大学出版社. 前4章

[2] 严蔚敏等.数据结构(C语言版).北京：清华大学出版社

目 录

[任 务 书 I](#_Toc67586059)

[设计内容 I](#_Toc67586060)

[设计要求 I](#_Toc67586061)

[参考文献 II](#_Toc67586062)

[目 录 III](#_Toc67586063)

[1 引言 5](#_Toc67586064)

[1.1 课题背景与意义 5](#_Toc67586065)

[1.1.1 课题研究背景 5](#_Toc67586066)

[1.1.2 课题研究意义 5](#_Toc67586067)

[1.2 国内外研究现状 5](#_Toc67586068)

[1.3 课程设计的主要研究工作 6](#_Toc67586069)

[2 系统需求分析与总体设计 7](#_Toc67586070)

[2.1 系统需求分析 7](#_Toc67586071)

[2.2 系统总体设计 7](#_Toc67586072)

[3 系统详细设计 9](#_Toc67586073)

[3.1 有关数据结构的定义 9](#_Toc67586074)

[3.1.1 词法分析 9](#_Toc67586075)

[3.1.2 语法树的结点类型 10](#_Toc67586076)

[3.1.3 抽象语法树 12](#_Toc67586077)

[3.1.4 变量链表 12](#_Toc67586078)

[3.1.5 数据之间的关系 13](#_Toc67586079)

[3.2 主要算法设计 13](#_Toc67586080)

[4 系统实现与测试 14](#_Toc67586081)

[4.1 系统实现 14](#_Toc67586082)

[4.1.1 词法分析 14](#_Toc67586083)

[4.1.2语法树的显示与语法分析 15](#_Toc67586084)

[4.1.3 程序的格式化处理 23](#_Toc67586085)

[4.2 系统测试 23](#_Toc67586086)

[5 总结与展望 41](#_Toc67586087)

[5.1 全文总结 41](#_Toc67586088)

[5.1 工作展望 41](#_Toc67586089)

[6 体会 42](#_Toc67586090)

1 引言

1.1 课题背景与意义

在计算机科学中，抽象语法树（abstract syntax tree或者缩写为AST），是将[源代码](https://baike.baidu.com/item/%E6%BA%90%E4%BB%A3%E7%A0%81)的语法结构的用树的形式表示，树上的每个结点都表示源程序代码中的一种语法成分。之所以说是“抽象”，是因为在抽象语法树中，忽略了源程序中语法成分的一些细节，突出了其主要语法特征。

1.1.1 课题研究背景

对于计算机科学与技术、我们已经学习了C语言程序设计、数据结构两门面向编程知识与技术的基础理论课，以及C语言程序设计实验、数据结构实验两门编程实践课程，不仅具有较为系统性的C语言、常用数据结构基本知识，而且具有初步的程序设计、数据抽象与建模、问题求解与算法设计的能力，奠定了进行复杂程序设计的知识基础。

1.1.2 课题研究意义

基于高级语言源程序格式处理工具这个课题，需要了解c语言程序单词的命名规则以及各种语法的判断，能够帮助自己更加深入的去了解c语言。

1.2 国内外研究现状

依托于高级语言和“巴克斯范式”（BNF），巴科斯范式以美国人巴科斯(Backus)和丹麦人诺尔(Naur)的名字命名的一种形式化的语法表示方法，用来描述语法的一种形式体系，是一种典型的元语言。又称巴科斯-诺尔形式(Backus-Naur form)。它不仅能严格地表示语法规则，而且所描述的语法是与上下文无关的。它具有语法简单，表示明确，便于语法分析和编译的特点。在各种文献中还存在巴科斯范式的一些变体，如[扩展巴科斯范式](https://zh.wikipedia.org/wiki/%E6%89%A9%E5%B1%95%E5%B7%B4%E7%A7%91%E6%96%AF%E8%8C%83%E5%BC%8F)EBNF或[扩充巴科斯范式](https://zh.wikipedia.org/wiki/%E6%89%A9%E5%85%85%E5%B7%B4%E7%A7%91%E6%96%AF%E8%8C%83%E5%BC%8F" \o "扩充巴科斯范式)ABNF。

1.3 课程设计的主要研究工作

主要的工作主要分为两部分，一是词法分析，二是语法分析。

**(1) 词法分析：**

词法分析需要识别出五类单词，标识符、关键字、常量、运算符和定界符，词法分析每识别出一个单词，就可返回单词的编码。词法分析的过程，就是在读取源程序的文本文件的过程中，识别出一个个的单词。在词法分析前，需要先给每一类单词定义一个类别码（用枚举常量形式），识别出一个单词后，即可得到该单词的类别码和单词自身值（对应的符号串）。实现词法分析器的相关技术是采用有穷自动机的原理。

**(2) 有穷状态机：**

对于有穷状态机的解释就是作用是描述对象在它的生命周期内所经历状态序列，以及如何响应来自外界的事件。有穷状态机首先包含一个有限状态的集合，还包含了从一个状态到另外一个状态的转换。有穷自动机看上去就像是一个有向图，其中状态是图的节点，而状态转换则是图的边。此外这些状态中还必须有一个初始状态和至少一个接受状态。

**(3) 语法分析：**

采用了递归下降子程序法，，而递归下降子程序是一种非常简洁的语法结构分析算法，通过对程序语句成分逐个分析，调用相应的子程序判断出语句所属的类别来作为AST树的子树，最后通过先序遍历将分析结果输出。

2 系统需求分析与总体设计

2.1 系统需求分析

用户有三大需求，如下:

1. **词法分析**

先把源文件中的字符逐个读取，再透过所写程序识别出五类单词，标识符、关键字、常量、运算符和定界符，词法分析每识别出一个单词，就可返回单词的编码。再输出单词类型和所对应单词供用户观看。

1. **语法树的显示与语法分析**

根据词法分析所得单词类型来进行语法分析，根据高级语言的语法规则分析程序的语法是否正确，如果正确，生成源程序的抽象语法树AST。再给予每个结点一个类型，再前序遍历所有结点，输出对应的语法树。

1. **格式化处理**

根据语法分析所生成的语法树来对源文件进行格式化的处理，使前序遍历来遍历语法树，再根据结点类型输出所对应的格式。

2.2 系统总体设计

系统分为三个模块，第一个模块是词法分析，第二个模块是语法树的显示与语法分析，第三个模块是格式化处理，故设计菜单界面(如图2-2所示)，菜单流程图如图2-1所示。

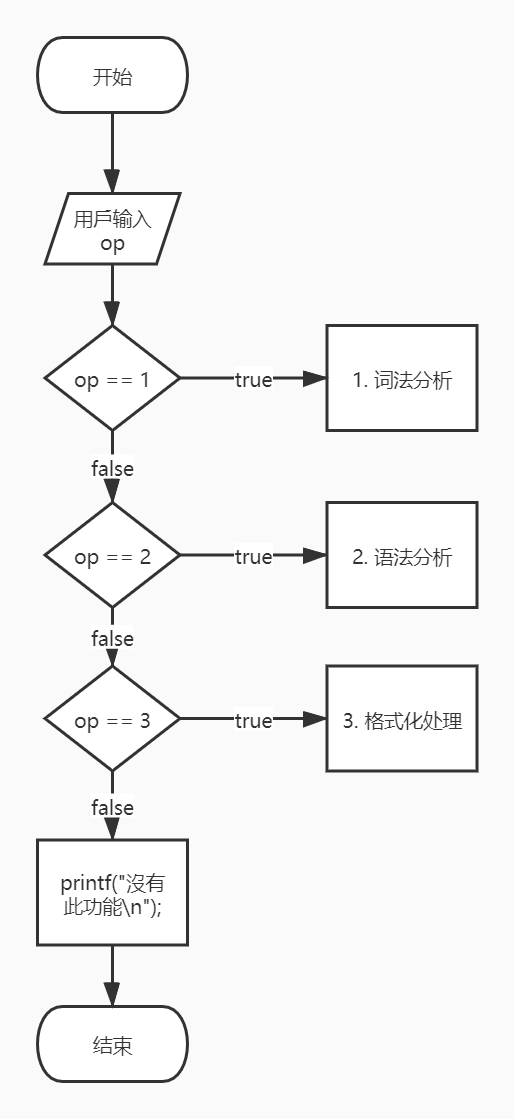


图2-1 菜单流程图

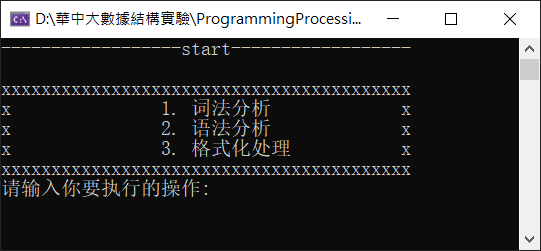


图2-2 菜单实际界面

3 系统详细设计

3.1 有关数据结构的定义

根据需求，词法分析每识别出一个单词，就可返回单词的编码。为唯一确定各单词的种类编码，需通过枚举类型定义各类单词的种类编号(如表3-1)。语法分析建立的语法树的每个结点也需要被定义种类编号(如表3-2)。

3.1.1 词法分析

通过枚举类型定义各类单词的种类编号如表3-1:

表3-1 单词类型

|  |  |
| --- | --- |
| 单词类型 | 种类编码 |
| 标识符 | IDENT |
| 关键字 | KEYWORD |
| 整型常量 | INT\_CONST |
| 浮点数常量 | FLOAT\_CONST |
| 字符常量 | CHAR\_CONST |
| 整型关键字 | INT |
| 浮点型关键字 | FLOAT |
| 字符型关键字 | CHAR |
| 有符号类型关键字 | SIGNED |
| 无符号类型关键字 | UNSIGNED |
| 无类型关键字 | VOID |
| 数组 | ARRAY |
| 关键字 | DO |
| 关键字 | WHILE |
| 关键字 | BREAK |
| 关键字 | SWITCH |
| 关键字 | CASE |
| 关键字 | DEFAULT |
| 关键字 | EXTURN |
| 关键字 | STATIC |
| 关键字 | CONST |
| 关键字 | TYPEDEF |
| 关键字 | FOR |
| 关键字 | IF |
| 关键字 | ELSE |
| 头文件引用 | INCLUDE |
| 宏定义 | DEFINE |
| 加号 | PLUS |
| 减号 | MINUS |
| 乘号 | MULTIPLE |
| 除号 | DIVIDE |
| 关键字 | CONTINUE |
| 关键字 | RETURN |
| 相等号 | EQ |
| 不相等号 | NEQ |
| 逻辑与 | AND |
| 逻辑或 | OR |
| 大于号 | GREATER |
| 大于等于号 | GREATER\_EQUAL |
| 小于号 | LESS |
| 小于等于号 | LESS\_EQUAL |
| 单行注释 | LINE\_COMMENT |
| 块注释 | BLOCK\_COMMENT |
| 賦值 | ASSIGN |
| 左括号 | LP |
| 右括号 | RP |
| 左大括号 | LB |
| 右大括号 | RB |
| 逗号 | COMMA |
| 分号 | SEMI |
| 井号 | POUND |
| 自增运算 | AUTO\_INCREASING |
| 自减运算 | AUTO\_DECREASING |
| 非法词汇 | ERROR\_TOKEN |

3.1.2 语法树的结点类型

通过枚举类型定义语法树结点的种类编号如表3-2:

表3-2 语法树的结点类型

|  |  |
| --- | --- |
| 结点类型 | 种类编码 |
| 外部定义 | extern\_define |
| 外部定义序列 | extern\_define\_sequence |
| 外部变量定义 | extern\_variable\_define |
| 变量 | variable |
| 变量类型 | variable\_type |
| 变量序列 | variable\_sequence |
| 函数定义 | function\_define |
| 函数名 | function\_name |
| 返回值类型 | return\_type |
| 函数形参序列 | formal\_parameter\_sequence |
| 参数类型 | formal\_parameter\_type |
| 函数形参 | formal\_parameter |
| 参数名 | formal\_parameter\_name |
| 复合语句 | complex\_statement |
| 语句序列 | statement\_sequence |
| 局部变量定义序列 | local\_variable\_define\_sequence |
| 局部变量定义 | local\_variable\_define |
| 语句 | statement |
| 表达式 | expression |
| 操作符 | operation |
| 表达式中的操作符 | expression\_ident |
| 操作数 | operator\_number |
| IF - ELSE子句 | IF\_ELSE\_node |
| IF子句 | IF\_node |
| IF内容类型 | IF\_content |
| ELSE结点 | ELSE\_node |
| WHILE子句 | WHILE\_node |
| WHILE条件 | WHILE\_condition |
| RETURN语句 | RETURN\_node |
| 没有RETURN值 | RETURN\_no\_node |
| FOR子句 | FOR\_node |
| 一个条件的FOR子句 | FOR\_node1 |
| 两个条件的FOR子句 | FOR\_node2 |
| 三个条件的FOR子句 | FOR\_node3 |
| FOR的条件一 | FOR\_condition1 |
| FOR的条件二 | FOR\_condition2 |
| "FOR的条件三 | FOR\_condition3 |
| FOR子句沒有条件 | FOR\_no\_condition\_node |
| CONTINUE语句 | CONTINUE\_node, |
| BREAK语句 | BREAK\_node |
| 数组名 | array\_name |
| 数组类型 | array\_type |
| 无视类型 | Ignore |
| 实参序列 | actual\_parameter\_sequence |
| 没有参数类型 | no\_formal\_parameter |
| 没有复合语句类型 | no\_complex\_statement |
| 没有实参序列类型 | no\_actual\_parameter\_sequence |
| 参数类型 | formal\_parameter\_type\_array |
| 参数名 | formal\_parameter\_name\_array |
| 函数形参 | formal\_parameter\_array |
| 错误类型 | ERROR\_TYPE |

3.1.3 抽象语法树

根据需求，我们要生成源程序的抽象语法树AST，此语法树有各种类型的结点，而且需要保存读入的数据和数据类型，基于这几点设计出较为合适的兄弟孩子树。语法树的结构定义如下:

1. **兄弟孩子语法树**

typedef struct ASTtree {

struct ASTtree\* sibling;

struct ASTtree\* children;

NodeType type;

NodeData data;

}ASTtree;

1. **语法树结点的数据**

typedef int DataType;

typedef struct NodeData {

DataType datatype; //结点的数据类型

char\* content; //结点的数据

}NodeData;

3.1.4 变量链表

根据巴克斯范式定义的语法规则，变量不能同名，故需要建立一个变量表来检查变量是否重定义，如果变量重定义了，则立即报错。此变量表也可用于检查是否使用了未定义的变量。变量结构链表的结构定义如下:

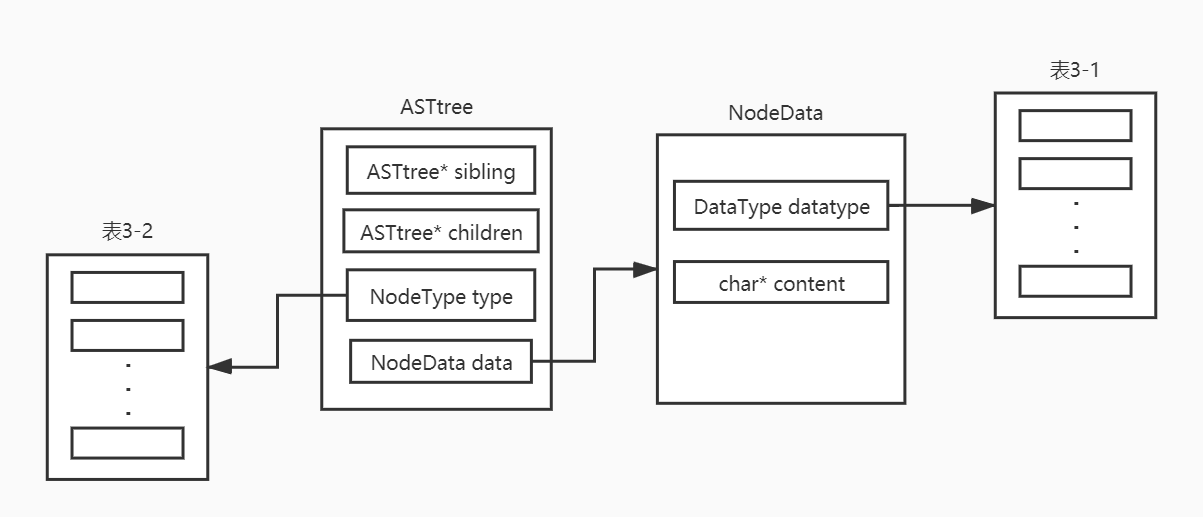
typedef struct VarList {

char\* var; //变量名

int size; //存放数组的数组大小，普通变量的size为0

DataType datatype; //变量类型

struct VarList\* next;

}Vars;

3.1.5 数据之间的关系

图3-1 数据之间的关系图

3.2 主要算法设计

系统分为三个模块，第一个模块是词法分析，第二个模块是语法树的显示与语法分析，第三个模块是格式化处理

**(1) 词法分析**

算法思想: 一共由两个cpp文件和两个头文件完成此模板，分别是gettoken.cpp、wordRecognization.cpp、gettoken.h和wordRecognization.h。在gettoken.cpp中用函数gettoken(FILE\* fp)从源文件中读取一个字符，如果字符是读取到的字符是换行符或制表符或空格，则忽略，如果是换行符时，纪录行数的变量加一，然后根据不同的字符进入到不同的IF子句，当IF字句结束时，表示成功的读取到了一个单词，返回单词的编码，单词自身值保存在全局变量token\_text中。再用wordRecognization.cpp的函数wordRecognization()调用函数gettoken(FILE\* fp)获得单词的编码，wordRecognization()根据单词的编码输出对应的单词类型和token\_text。

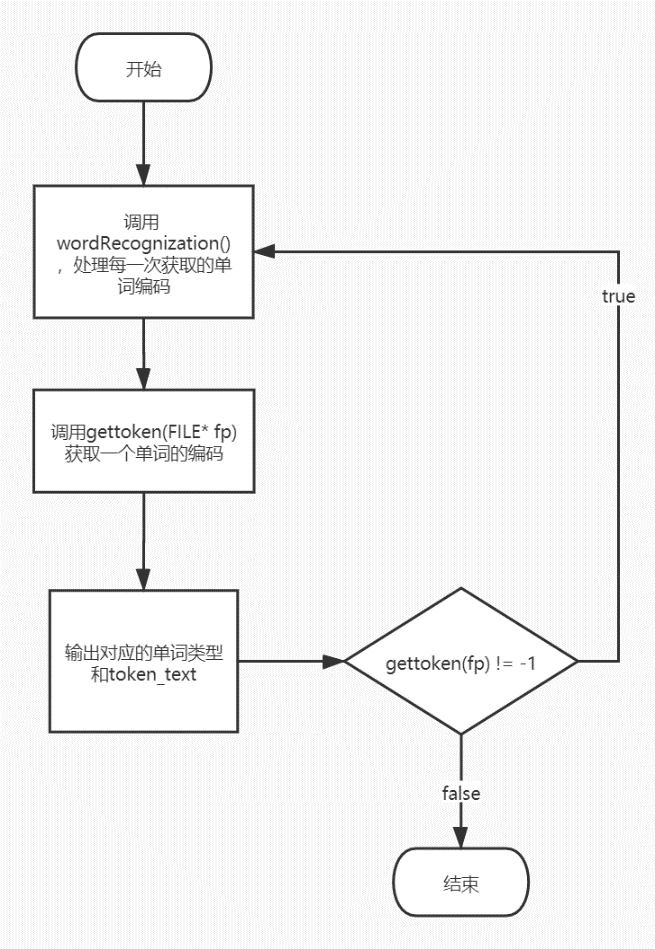


图3-2 词法分析的流程图

**(2) 语法分析与语法树的显示**

算法思想: 一共由一个cpp文件和一个头文件完成此模板，分别是syntaxAnalyse.cpp和syntaxAnalyse.h。语法分析过程中，需要逐个读取源程序中的单词，具体实现时，可以使用2种方式，一种是一次性的读取源程序文件的所有单词，得到一个单词的线性表，每个数据元素保存单词的种类码和自身值。单词的线性表即可采用顺序表，也可以用链表方式，这样语法分析需要单词时，直接从线性表中取单词；第二种方式是语法分析时，需要单词时，调用一次词法分析的函数，读取一个单词。设w为全局变量，存放当前读入的单词种类编码，token\_text保存单词的自身值。errors 表示错误标记，一旦有错，释放抽象语法树全部结点的空间。最后根据生成的语法树，显示语法树。

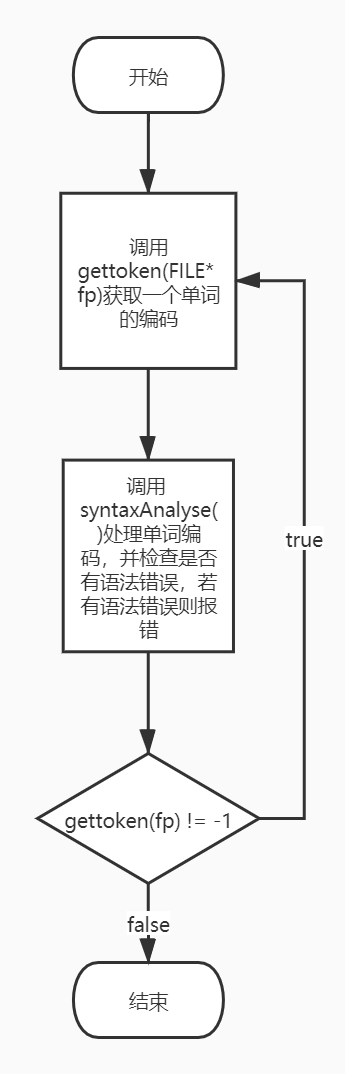


图3-3 语法分析与语法树的显示的流程图

1. **程序的格式化处理**

算法思想: 一共由一个cpp文件和一个头文件完成此模板，分别是outputFormat.cpp和outputFormat.h。由第二个模板所生成的抽象语法树来还原出源程序，并缩进编排重新生成源程序文件。

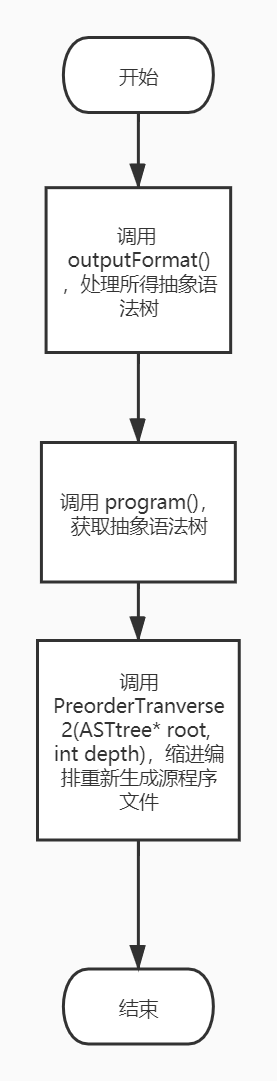


图3-4 程序的格式化处理流程图

4 系统实现与测试

4.1 系统实现

该系统实现的软件环境是Windows操作系统、Visual Studio 2019；硬件环境为：处理器：2.60GHz，内存：16GB。

4.1.1 词法分析

**(1) is16(char c);**

函数参数: char c(从源文件读取的字符)

算法思想:此函数用于判断字符是否是十六进制的字符。如果字符c是数字或是在a到f和A到F的范围内，则该字符是十六进制的字符，返回1。否则，返回0。

**(2) int check\_suffix(char c);**

函数参数: char c(从源文件读取的字符)

算法思想: 此函数用于判断是否是正确的后缀，如果字符c是f或F或u或l或L时，则该字符是正确的后缀，返回0。否则，返回1。

**(3) int gettoken(FILE\* fp);**

函数参数: FILE\* fp(源文件)

算法思想:从源文件中读取一个字符，如果字符是读取到的字符是换行符或制表符或空格，则忽略，如果是换行符时，纪录行数的变量加一，然后根据不同的字符进入到不同的IF子句，当IF字句结束时，表示成功的读取到了一个单词，返回单词的编码，单词自身值保存在全局变量token\_text中。

1. **int wordRecognization();**

函数参数:无

算法思想:此函数用于打印出词法分析的结果。首先先打开文件，如果不能读取文件里的内容，就立即报错和返回ERROR。如果读取成功，则调用gettoken()函数取得词法分析的编码，再根据编码打印出对应的单词类型。

4.1.2语法树的显示与语法分析

**(1) ASTtree\* program();**

函数参数:无

算法思想: 此函数用于生成一棵语法树，创建一个结点空间给根结点，根指针指向的是一个外部定义序列的结点。

**(2) ASTtree\* ExtDefList();**

函数参数:无

算法思想: 此函数用于处理一系列的外部定义。如果一开始调用GetTokenWithOutANNO()函数得到的INCLUDE或是DEFINE时，则忽略它们，继续调用GetTokenWithOutANNO()函数，因为宏定义和头文件引用都不参与语法分析。每个外部定义序列的结点，其第一个子树对应一个外部定义，第二棵子树对应后续的外部定义序列，直到没有外部定义序列，只有外部定义。

**(3) ASTtree\* ExtDef();**

函数参数:无

算法思想: 此函数用于完成一个外部定义的处理，调用此子程序时，已经读入了一个外部定义的第一个单词到w中。不管是外部变量定义，还是函数定义，第一个单词必须是类型关键字，第二个一定是标识符，只有读入第三个才可能区分，如果是小括号，就是函数的定义与声明，调用函数定义子程序FuncDef()，否则按外部变量的形式来处理，调用外部变量定义子程序ExtVarDef()。在分辨完第一个单词是类型关键字后，要把类型关键字保存到kindName，在分辨完第二个单词是标识符后，要把标识符保存到varname，还要调用check\_vars()函数检查变量链表中是否有此标识符，没有则把这个标识符加入到变量链表中，有则报错，返回NULL。该子程序处理完后，刚好处理到外部定义的最后一个符号，后续单词还没读入。

**(4) ASTtree\* ExtVarDef(char\* varname, char\* kindName);**

函数参数:char\* varname, char\* kindName

算法思想: 此函数用于完成一个外部变量定义的处理。分配空间建立一个指针p，且把kindName赋值给指针p的数据内容。如果此时的isarray是为1时，即此时是在定义一个数组，指针p的类型就是array\_type，否则是variable\_type。每个外部变量定义的结点，其第一个子树对应指针p，第二棵子树对应调用变量序列子程序VarList(char\* varname)。

**(5) ASTtree\* VarList(char\* varname);**

函数参数: char\* varname

算法思想: 此函数用于完成一个变量序列的处理。如果标识符后不是逗号或分号，则调用freeAST(ASTtree \*root)函数来释放所有结点，如果标识符后是逗号，则继续调用变量序列子程序VarList(char\* varname)，如果是分号则调用变量子程序VarDef(char\* varname)，每个变量序列的结点，其第一个子树对应一个变量，第二棵子树对应后续的变量序列，直到没有变量序列，只有变量。

**(6) ASTtree\* VarDef(char\* varname);**

函数参数: char\* varname

算法思想: 此函数用于完成一个变量的处理。分配空间建立一个指针p，且把varname赋值给指针p的数据内容。如果此时的isarray是为1时，即此时是在定义一个数组，指针p的类型就是array\_name，否则是variable。每个变量定义的结点，其第一个子树对应指针p，没有第二棵子树。

**(7) ASTtree\* FuncDef(char\* varname, char\* kindName);**

函数参数: char\* varname, char\* kindName

算法思想: 此函数用于完成一个函数定义的处理。调用此子程序时，函数返回值类型和函数名，正小括号的单词已经读入，函数名保存在varname中，函数类型说明符保存在kindName中，生成函数定义指针root，生成返回值类型结点，作为root的第一个孩子。处理参数部分到反小括号结束，调用形参序列子程序ForParaList()，得到参数部分的子树根指针，无参函数得到NULL，该子树作为root的第二棵子树。读入符号，如果分号，就是函数原型声明，函数体子树为空；正大括号，则调用复合语句子程序ComSent()，得到函数体子树根指针，其它符号表示有错。得到的函数体子树作为root的第三棵子树，最后返回root。

**(8) ASTtree\* ForParaList();**

函数参数:无

算法思想: 此函数用于完成一个形参序列的处理。如果函数第一个得到的w是右括号则表示后面该函数没有形参，故立即返回root，root的结点类型是no\_formal\_parameter。如果w是逗号，则调用GetTokenWithOutANNO()函数赋值给w。每个形参序列的结点，其第一个子树对应调用形参子程序ForParaDef()，第二棵子树对应调用形参序列子程序ForParaList()。

**(9) ASTtree\* ForParaDef();**

函数参数:无

算法思想: 此函数用于完成一个形参的处理。把形参的类型说明符保存到数组name中，并分配空间建立一个指针p，再把name用strcpy函数复制到指针p的数据内容中，指针p的结点类型是formal\_parameter\_type。把形参的标识符保存到数组name中，并分配空间建立一个指针q，再把name用strcpy函数复制到指针q的数据内容中，如果形参是数组时，还要保存数组的大小，指针q的结点类型是formal\_parameter\_name\_array，否则结点类型是formal\_parameter\_name。每个形参的结点，其第一个子树对应指针p，第二棵子树对应指针q。

**(10)ASTtree\* ActParaList(int x);**

函数参数:int x

算法思想: 此函数用于完成一个实参序列的处理。每个实参序列的结点，其第一个子树对应调用表达式子程序Expression(int endsym)，第二棵子树对应调用实参序列子程序ActParaList(int x)。 在调用子程序ActParaList(int x)前，要先用ftell函数纪录下fp的位置，如果表达式子程序的返回值是NULL时，就要用fseek函数读取回到ftell所纪录的位置。

**(11)ASTtree\* CompSent();**

函数参数:无

算法思想: 此函数用于完成一个复合语句的处理。每个复合语句的结点，其第一个子树对应调用局部变量定义序列子程序LocalVarDefSeq()，第二棵子树对应调用语句序列子程序ScentSeq()。若局部变量定义序列子程序的返回值是NULL，则无局部变量说明，若语句序列子程序的返回值是NULL，则无语句说明。当两棵子树中有一棵不为NULL时，根结点类型就是complex\_statement，否则为no\_complex\_statement。

**(12)ASTtree\* ScentSeq();**

函数参数:无

算法思想: 此函数用于完成一个语句序列的处理。初始化子树，根指针root指向NULL。调用处理一条语句的子程序Statement()，返回其子树根指针r1。如果r1为空，则表示语句序列已结束，返回NULL，否则生成语句序列的结点root，其第一棵子树对应指针r1，第二棵子树递归调用处理语句序列子程序ScentSeq()后的返回值，最后返回根指针root。

**(13)ASTtree\* LocalVarDefSeq();**

函数参数:无

算法思想: 此函数用于完成一个局部变量定义序列的处理。每个局部变量定义序列的结点，其第一棵子树对应调用局部变量定义子程序LocalVarDef()，第二棵子树递归调用处理局部变量定义序列子程LocalVarDefSeq()后的返回值，最后返回根指针root。在调用局部变量定义子程序前要先判断w是否是类型说明符，不是则直接返回NULL，此时表示局部变量定义为空。

**(14)ASTtree\* LocalVarDef();**

函数参数:无

算法思想: 此函数用于完成一个局部变量定义的处理。进入此函数前，w已是一个类型说明符，把类型说明符保存在数组\_type，再调用GetTokenWithOutANNO()函数获取一个新的w，检查此时的w是否为标识符或数组，不是则直接报错且返回NULL，是则保存标识符或数组的标识符到数组varname，且加入到变量链表中，之后把数组\_type赋值给第一棵子树的数据内容。每个局部变量定义的结点，其第一个子树保存了类型说明符的指针，第二棵子树对应调用变量序列子程序VarList(char\* varname)。

**(15)void freeAST(ASTtree\* root);**

函数参数: ASTtree\* root

算法思想:此函数用于把语法树的结点都释放。当根结点的数据内容不为NULL时就把它free掉，再把它指向NULL。当根结点的第一棵子树不为NULL时就递归调用函数freeAST(ASTtree\* root)释放第一棵子树的所有结点，再把它指向NULL。当根结点的第二棵子树不为NULL时就递归调用函数freeAST(ASTtree\* root)释放第二棵子树的所有结点，再把它指向NULL。最后才释放根结点，再把它指向NULL，使用了前序遍历递归调用函数freeAST(ASTtree\* root)。

**(16)ASTtree\* Statement();**

函数参数:无

算法思想: 此函数用于完成一个语句的处理。调用此子程序时，语句的第一个单词已经读入，处理一条语句时，根据这条语句的第一个单词，确定处理什么类型的语句。主要需要处理的语句如下:

①if-else语句和if语句

如遇到关键字if，若读入的下一个单词是左括号，则调用表达式子程序Expression(int endsym)得到IF的条件子树，其中endsym是右括号，否则报错并返回NULL。然后读入下一个单词，调用语句子程序Statement()得到IF子句的子树根指针，完成后，再读入下一个单词，如果是else，则调用处理一条语句的子程序，得到ELSE子句的子树根指针。生成IF-ELSE结点，下挂IF条件、IF子句、ELSE子句3棵子树，否则就是if语句，生成IF结点，下挂IF条件、IF子句2棵子树。

②while语句

如遇到关键字while，若读入的下一个单词是左括号，则调用表达式子程序Expression(int endsym)得到WHILE的条件子树根指针，其中endsym是右括号，否则报错并返回NULL。然后读入下一个单词，调用语句子程序Statement()得到WHILE子句的子树根指针。生成WHILE结点，下挂WHILE条件、WHILE子句2棵子树。

③for语句

如遇到关键字for，若读入的下一个单词是左括号，则调用表达式子程序Expression(int endsym)得到FOR的条件子树根指针，否则报错并返回NULL。由于for语句有一共有三个条件，而且皆可省略，所以读入的下一个单词后第一次调用Expression(int endsym)时得到的是for语句的第一个条件的子树，其中endsym是分号。然后再读入单词，再调用Expression(int endsym)时得到的是for语句的第二个条件的子树，其中endsym是分号。再次读入单词，调用Expression(int endsym)时得到的是for语句的第三个条件的子树，其中endsym是右括号。再由得出的子树是否为NULL来判定是一个条件的FOR子句、两个条件的FOR子句，还是三个条件的FOR子句来生成结点，下挂对应的条件子树。最后调用语句子程序Statement()得到FOR子句的子树根指针，再下挂在根树上。

④continue语句

如遇到关键字continue，若读入的下一个单词是分号，则生成CONTINUE结点，否则报错并返回NULL。

⑤break语句

如遇到关键字break，若读入的下一个单词是分号，则生成BREAK结点，否则报错并返回NULL。

⑥return语句

如遇到关键字return，先读入一个单词，然后调用语句子程序Statement()得到RETURN语句的子树指针，然后生成RETURN结点，下挂子树指针。

此子程序调用结束时，会读入下一条语句的第一个单词到w中，以便后续处理。

**(17)ASTtree\* Expression(int endsym);**

函数参数: int endsym(表达式的结束符)

算法思想: 此函数用于完成一个表逹式的处理。首先先用一个二维数组定义运算符的优先级，1代表大于，0代表等于，-1代表小于。然后每当遇到一个操作数，生成一个结点，将结点指针进操作数栈，每当需要处理一个运算符，根据优先级的不同，进入不同的case中处理，最后生成运算符的结点，把操作数的结点作为该结点的孩子。如果遇到结束符，结束符就会被替换成井号，然后就能退出循环，如果此时操作数栈只有一个结点指针且操作当运算符栈栈顶是起止符号且没有错误时，返回表达式语法树的根结点指针，否则报错并返回NULL。

**(18)void printType(NodeType type);**

函数参数: NodeType type

算法思想:此函数用于打印出语法树的结点类型。根据传入参数的类型用switch判断并打印出对应的中文。

**(19)void PreorderTranverse(ASTtree\* root, int depth);**

函数参数: ASTtree\* root, int depth

算法思想:前序遍历生成的语法树，根据种类编码来输出相对应的结点类型，还要根据对种类编码和树的深度不同来输出一定数量的空格，达到控制输出格式的效果。

**(20)int syntaxAnalyse();**

函数参数:无

算法思想: 首先先打开文件，如果不能读取文件里的内容，就立即报错和返回ERROR。如果读取成功，则调用program()函数赋值给语法树的头结点，然后再调用PreorderTranverse()函数前序遍历输出语法树。

**(21)int GetTokenWithOutANNO();**

函数参数:无

算法思想:此函数用于忽略对行注释和块注释的语法分析，因为行注释和块注释是不需要做语法分析的。如果gettoken(fp)所得是行注释或块注释的编码，则再令w = gettoken(fp)一次。

**(22)int add\_vars(char\* varname, int cap, DataType kind);**

函数参数: char\* varname, int cap, DataType kind

算法思想:此函数用于把新定义的变量加入到变量链表中。用varname、cap和kind的值初始化结点p，再使用头插法把结点p插入到varRoot。如果成功插入就返回0，否则返回-1。

**(23)int check\_vars(char\* varname);**

函数参数: char\* varname

算法思想: 此函数用于检查varname是否已经在变量链表中。遍历变量链表，如果匹配到和varname一样的变量名时报错并返回-1，否则返回0。

**(24)int had\_return(ASTtree\* root,DataType kind);**

函数参数: ASTtree\* root,DataType kind

算法思想:此函数用于处理函数的返回值是否确。首先定义了变量isreturn，当isreturn为1和3时表示有返回值，isreturn为2表示函数没有返回值。定义了变量countreturn纪录语法树中return的个数。定义了变量countvarreturn纪录返回的变量是否已经被定义。定义了变量hadreturn纪录返回值是否与函数一致的次数。 使用前序遍历来遍历语法树，当遍历到结点的类型是RETURN\_node时，countreturn加一，然后遍历变量链表，当遍历到有相同的变量名时，countvarreturn加一，如果变量名的类型和返回值的类型不一致时，hadreturn加一。

函数的返回值是否确一共分为四种情况:

①函数需要返回值，但没有返回值。

②返回值类型不匹配

③函数不需要返回值，但却有返回值。

④返回值的变量并没有被定义

情况一即countvarreturn为零且isreturn为3时或countvarreturn大于零且isreturn为1且hadreturn大于零时，返回-1。

情况二即hadreturn大于零，返回-2。

情况三即isreturn不等于3或 isreturn等于1并w是结束符，报错。

情况四即countvarreturn等于零，返回-3。

4.1.3 程序的格式化处理

**int outputFormat();**

函数参数:无

算法思想:打开源文件，如果可以读入，则把头文件引用和宏定义先打印出，然后调用子程序program()获取已生成的语法树，如果program()返回的值是NULL时，则立即报错且exit(-1)，否则就再调用PreorderTranverse2(ASTtree\* root, int depth)函数来前序遍历语法树。否则就报错且返回-1。

**void PreorderTranverse2(ASTtree\* root, int depth);**

函数参数: ASTtree\* root, int depth(语法树的根和语法的深度)

算法思想: 此函数用于打印出生成风格统一的格式化缩进编排的源程序文件。使用先序遍历遍功语法树，根据根指针和其子树的种类编码来输出格式和在数据内容里的内容。

4.2 系统测试

4.2.1 词法分析

**(1) 识别单词**

一共能识别47个单词种类。

表4-1 测试用例与结果

|  |  |  |  |
| --- | --- | --- | --- |
| 用例 | 描述 | 预期结果 | 运行结果 |
| (1) | 识别单词 | 图4-2-1识别单词 | 图4-2-1识别单词 |

**(2) 词法异常处理**

表4-2 测试用例与结果

|  |  |  |  |
| --- | --- | --- | --- |
| 用例 | 描述 | 预期结果 | 运行结果 |
| (1) | 不能识别的符号 | 图4-2-2不能识别的符号 | 图4-2-2不能识别的符号 |
| (2) | 错误后缀 | 图4-2-3 错误后缀 | 图4-2-3 错误后缀 |

4.2.2 语法树的显示与语法分析

**(1) 语法分析**

一共可以分析19种语句。

表4-3 测试用例与结果

|  |  |  |  |
| --- | --- | --- | --- |
| 用例 | 描述 | 预期结果 | 运行结果 |
| (1) | 表达式语句 | 图4-2-4 表达式语句 | 图4-2-4 表达式语句 |
| (2) | if语句 | 图4-2-5 if语句 | 图4-2-5 if语句 |
| (3) | if else语句 | 图4-2-6 if else语句 | 图4-2-6 if else语句 |
| （4） | [while](https://www.educoder.net/tasks/o4rahntsuxz9)语句 | 图4-2-7 [while](https://www.educoder.net/tasks/o4rahntsuxz9)语句 | 图4-2-7 [while](https://www.educoder.net/tasks/o4rahntsuxz9)语句 |
| (5) | 一个条件的for语句 | 图4-2-8 一个条件的for语句 | 图4-2-8 一个条件的for语句 |
| (6) | 二个条件的for语句 | 图4-2-9 二个条件的for语句 | 图4-2-9 二个条件的for语句 |
| (7) | 三个条件的for语句 | 图4-2-10 三个条件的for语句 | 图4-2-10 三个条件的for语句 |
| (8) | 无条件的for语句 | 图4-2-11 无条件的for语句 | 图4-2-11 无条件的for语句 |
| (9) | return语句 | 图4-2-12 return语句 | 图4-2-12 return语句 |
| (10) | break语句 | 图4-2-13 break语句 | 图4-2-13 break语句 |
| (11) | continue语句 | 图4-2-14 continue语句 | 图4-2-14 continue语句 |
| (12) | 复合语句 | 图4-2-15 复合语句 | 图4-2-15 复合语句 |
| (13) | 函数定义 | 图4-2-16 函数定义 | 图4-2-16 函数定义 |
| (14) | 函数声明 | 图4-2-17 函数声明 | 图4-2-17 函数声明 |
| (15) | 函数调用 | 图4-2-18 函数调用 | 图4-2-18 函数调用 |
| (16) | if语句嵌套 | 图4-2-19 if语句嵌套 | 图4-2-19 if语句嵌套 |  |
| (17) | 循环语句嵌套 | 图4-2-20 循环语句嵌套 | 图4-2-20 循环语句嵌套 |
| (18) | 外部变量说明语句 | 图4-2-21 外部变量说明语句 | 图4-2-21 外部变量说明语句 |
| (19) | 局部变量说明语句 | 图4-2-22 局部变量说明语句 | 图4-2-22 局部变量说明语句 |

**(2) 语法异常处理**

表4-4 测试用例与结果

|  |  |  |  |
| --- | --- | --- | --- |
| 用例 | 描述 | 预期结果 | 运行结果 |
| (1) | 使用了未定义变量 | 图4-2-23 使用了未定义变量 | 图4-2-23 使用了未定义变量 |
| (2) | 重定义变量名 | 图4-2-24 重定义变量名 | 图4-2-24 重定义变量名 |
| (3) | 括号匹配错误，缺少左括号 | 图4-2-25 括号匹配错误 | 图4-2-25 括号匹配错误 |
| （4） | 表达式错误 | 图4-2-26 表达式错误 | 图4-2-26 表达式错误 |
| (5) | 函数缺少返回值 | 图4-2-27 函数缺少返回值 | 图4-2-27 函数缺少返回值 |
| (6) | void函数不应该有返回值 | 图4-2-28 void函数不应该有返回值 | 图4-2-28 void函数不应该有返回值 |

4.2.3 程序的格式化处理

表4-5 测试用例与结果

|  |  |  |  |
| --- | --- | --- | --- |
| 用例 | 描述 | 预期结果 | 运行结果 |
| (1) | 程序的格式化处理 | 图4-2-29 程序的格式化处理 | 图4-2-29 程序的格式化处理 |

5 总结与展望

5.1 全文总结

本实验学习了巴克斯（BNF）范式定义高级语言的词法规则和抽象语法树的基本知识，再结合大一时所学的c语言课程来完成此次实验。

对基于高级语言源程序格式处理工具研究过程中所做的主要工作如下：

1. 词法分析，识别出所有按词法规则定义的单词。
2. 语法分析，根据定义的语法规则，分析单词序列是否满足语法规则，同时生成抽象语法树。
3. 程序的格式化处理，通过对抽象语法树的遍历，生成风格统一的格式化缩进编排的源程序文件。

5.1 工作展望

本程序能识别基本的词法和语法分析，但还需要功能上的完善。在词法识别上还缺少一些特别的关键字的识别，例如union和goto。在语法分析上，程序报错时，提示语句还不够清晣。还有此程序虽然能识别标识符是否是被定义后才使用，但因为目前程序并没有分析定义域的功能，所以并不能分辨标识符是全局变量还是局部变量。

下面列出了几个将来的工作重点和研究方向：

（1）能把其余的关键字都能成功识别。

（2）可以尝试写一个专门用来报错的函数，减少重覆性的代码，并设计一个规范来输出错误。

（3）可以尝试分析定义域，这样才能让在函数内定义的变量，不会在函数外使用。

（4）能分辨所include的库和文件是否存在。不存在则报错。

（5）實現指針的功能。

6 体会

此次实验是我做得最久的一次实验，也是学到最多的一次实验，在此次实验中，我实际运用到了之前在数据结构课堂学到的知识，例如链表和二叉树，重新巩固了这方面的知识。通过这次实验，我对c语言的了解加深了，并了解到做一个简单编译器的基本步骤和原理，了解过后，我还觉得挺有趣，有时间的话我一定会选修编译原理来深入了解学习。

在此实验中，我还尝试解决了多种多样的bug,例如读取权限错误，是因为左子树和右子树没有初始化，虽然很简单，但却是很容易会忘记的细节。除了编译器所报的bug,更多是功能上的bug，我每次都是解决掉一个bug后，就发现还有其他的bug，然后又要列一个表格来提醒自己有哪些bug还没弄，虽然整个过程很累，但每当解决掉一个bug时却很有成就感。有时自己能想一些不一样的解决办法时，也很有成就感，例如表达式子程序的左括号和右括号，我并没有用传统的方式来进栈和出栈左右括号来检查是否可以配对，我是直接递归一个表达式子程序，其结束符是右括号，正好可以利用表达式子程序中找不到结束符就会报错的功能来解决括号的配对问题。

参考文献

[1] 王生原，董渊，张素琴，吕映芝等. 编译原理（第3版）. 北京：清华大学出版社. 前4章

[2] 严蔚敏等.数据结构(C语言版).北京：清华大学出版社

附录

1. main.cpp 主函数

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "gettoken.h"

#include "wordRecognization.h"

#include "syntaxAnalyse.h"

#include "outputFormat.h"

FILE\* fp;

Vars\* varRoot = NULL, \* varTail = NULL;

int op;

int main(int argc, char\*\* argv) {

varRoot = (Vars\*)malloc(sizeof(Vars));

varTail = varRoot;

printf("%s", "------------------start------------------\n\n");

for (int i = 0; i < 41; i++) {

printf("x");

}

printf("\n");

printf("x\t\t1. 词法分析\t\tx\n");

printf("x\t\t2. 语法分析\t\tx\n");

printf("x\t\t3. 格式化处理\t\tx\n");

for (int i = 0; i < 41; i++) {

printf("x");

}

printf("\n");

printf("请输入你要执行的操作: ");

scanf("%d", &op);

if (op == 1) {

wordRecognization();

}

else if (op == 2) {

syntaxAnalyse();

}

else if (op == 3) {

outputFormat();

}

else {

printf("沒有此功能\n");

}

printf("%s", "\n------------------over-------------------\n");

free(varRoot);

varRoot = NULL;

return 0;

}

1. gettoken.h 头文件

#pragma once

#include <map>

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#include <string>

#define ERROR -1;

namespace tk {

enum token\_kind {

MULTIPLE,

DIVIDE,

PLUS,

MINUS,

LESS\_EQUAL,

GREATER\_EQUAL, //大於等於>=

LESS,

GREATER, // 大於

EQ,

NEQ,

AND,

OR,

ASSIGN, //賦值

LP, //左括号

RP, //右括号

POUND, //起止号

AUTO\_INCREASING,//自增

AUTO\_DECREASING,

ERROR\_TOKEN,

KEYWORD,//关键字

IDENT, //标识符

INT\_CONST, // 整数常量

FLOAT\_CONST,// 浮点数常量

CHAR\_CONST,// 字符常量

INT,

FLOAT,

CHAR,

SIGNED,

UNSIGNED,

VOID,

COMMA,//逗号

SEMI,//分号

LB,//左大括号

RB,//右大括号

ARRAY,

LINE\_COMMENT,//注释

BLOCK\_COMMENT,/\*\*/

DO,

WHILE,

BREAK,

SWITCH,

CASE,

DEFAULT,

EXTURN,

STATIC,

CONST,

TYPEDEF,

FOR,

IF,

ELSE,

INCLUDE,

DEFINE,

CONTINUE,

RETURN,

};

//double d = 60;//double\_const

static std::map<std::string, token\_kind> keywordMap = {

{ "char", token\_kind::CHAR}, {"float",token\_kind::FLOAT},{"int", token\_kind::INT},

{"signed",token\_kind::SIGNED},{"unsigned",token\_kind::UNSIGNED}, {"void",token\_kind::VOID},{"for",token\_kind::FOR},{"do",token\_kind::DO},{"while",token\_kind::WHILE},

{"break", token\_kind::BREAK},{"continue",token\_kind::CONTINUE},{"if",token\_kind::IF},{"else",token\_kind::ELSE},{"switch",token\_kind::SWITCH},{"case",token\_kind::CASE},

{"default",token\_kind::DEFAULT},{"return",token\_kind::RETURN},{"extern",token\_kind::EXTURN},{"static",token\_kind::STATIC},{"const",token\_kind::CONST},{"typedef",token\_kind::TYPEDEF}

};

static char keyword[21][10] = { "char","float","int","signed","unsigned","void",

"for","do","while","break","continue","if","else","switch","case","default","return",

"extern","static","const","typedef"

};

};

int gettoken(FILE\* fp);

int is16(char c);

int check\_suffix(char c);

1. gettoken.cpp源文件

#include "gettoken.h"

using namespace std;

using namespace tk;

char token\_text[30];

char array\_capacity[10]; //存放數組容量

int row\_count = 1; //行数计数器

int array\_capacity\_int = 0;

int is16(char c) {

if ((isdigit(c)) || (c >= 'a' && c <= 'f') || (c >= 'A' && c <= 'F')) {

return 1;

}

else {

return 0;

}

};

int gettoken(FILE\* fp) {

char c = '0';

int i = 0;

int k = 0;

token\_text[i] = '\0';

c = fgetc(fp);

while (c == ' ' || c == '\t' || c == '\n') {

if (c == '\n') {

row\_count++;

}

c = fgetc(fp);

}

const int ident\_max\_len = 200;

// 數組

// 是關鍵字?是標識符?

// 1 0 - 无限（min(string.len-1, 255）)

// （字母|下划线）（字母|数字|下划线）\*

// string == 關鍵字? => string 比较相等时返回keyword ，否则返回识别符。

if (c == '@' || c == '&') {

return ERROR\_TOKEN;

}

if (isalpha(c) || c == '\_') {

i = 0;

do {

token\_text[i++] = c;

c = fgetc(fp);

} while (isalnum(c) || c == '\_');

token\_text[i] = '\0';

for (int j = 0; j < 21; j++) {

if (strcmp(token\_text, keyword[j]) == 0) {

ungetc(c, fp);

return keywordMap[token\_text];

}

}

int limit = 0;

if (c == '[') {

do {

token\_text[i++] = c;

if (limit > 0) {

array\_capacity[k++] = c;

}

c = fgetc(fp);

limit++;

} while (isdigit(c) && limit <= 10);

array\_capacity[k] = '\0';

array\_capacity\_int = atoi(array\_capacity);

if (c == ']') {

token\_text[i++] = c;

token\_text[i] = '\0';

return ARRAY;

}

else {

return ERROR\_TOKEN;

}

}

else {

ungetc(c, fp);

token\_text[i] = '\0';

return IDENT;

}

}

//float a = 1U.75;

//整數常量、浮點數常量

if (isdigit(c)) {

i = 0;

do {

token\_text[i] = c;

i++;

c = fgetc(fp);

} while (isdigit(c));

// 跟着不是. u l 就是有可能是整数

if (c != '.' && c != 'u' && c != 'l') {

// + - \* / % ( ) 5(5\*4)

if (c != ' ' && c != ';' && c != '%' && c != '(' && c != ')' && c != '+' && c != '-' && c != '\*' && c != '/') {

return ERROR\_TOKEN;

}

ungetc(c, fp);

token\_text[i] = '\0';

return INT\_CONST;

}

else if (c == '.') {//10.0f

do {

token\_text[i++] = c;

c = fgetc(fp);

} while (isdigit(c));

if (c == 'f') {

token\_text[i] = c;

i++;

token\_text[i] = '\0';

return FLOAT\_CONST;

}else if (c == ' ' || c == ';' || c == '%' || c == '(' || c == ')' || c == '+' || c == '-' || c == '\*' || c == '/' || check\_suffix(c)) {

return ERROR\_TOKEN;

}

token\_text[i] = '\0';

return FLOAT\_CONST;

}

else if (c == 'u') {

token\_text[i] = c;

i++;

c = fgetc(fp);

if (c == 'l' || c == 'L') {

token\_text[i] = c;

i++;

c = fgetc(fp);

if (c == 'l' || c == 'L') {

token\_text[i] = c;

i++;

token\_text[i] = '\0';

return INT\_CONST;

}

else {

ungetc(c, fp);

token\_text[i] = '\0';

return INT\_CONST;

}

}else if (c == ' ' || c == ';' || c == '%' || c == '(' || c == ')' || c == '+' || c == '-' || c == '\*' || c == '/' || check\_suffix(c)) {

return ERROR\_TOKEN;

}

else {

ungetc(c, fp);

token\_text[i] = '\0';

return INT\_CONST;

}

}

else if (c == 'l' || c == 'L') {

token\_text[i] = c;

i++;

c = fgetc(fp);

if (c == 'l' || c == 'L') {

token\_text[i] = c;

i++;

token\_text[i] = '\0';

return INT\_CONST;

}

else if (c == 'u') {

token\_text[i++] = c;

token\_text[i] = '\0';

return INT\_CONST;

}

else if (c == ' ' || c == ';' || c == '%' || c == '(' || c == ')' || c == '+' || c == '-' || c == '\*' || c == '/' || check\_suffix(c)) {

return ERROR\_TOKEN;

}

else {

ungetc(c, fp);

token\_text[i] = '\0';

return INT\_CONST;

}

}

else {

return ERROR\_TOKEN;

}

}

//0123、0x123

if (c == '0') {

c = fgetc(fp);

if (c >= '0' && c <= '7') {

token\_text[i++] = c;

do {

token\_text[i++] = c;

c = fgetc(fp);

} while (c >= '0' && c <= '7');

ungetc(c, fp);

token\_text[i] = '\0';

return INT\_CONST;

}

else if (c == 'x') {

token\_text[i++] = c;

do {

token\_text[i++] = c;

c = fgetc(fp);

} while (is16(c));

ungetc(c, fp);

token\_text[i] = '\0';

return INT\_CONST;

}

else {

return ERROR\_TOKEN;

}

}

//.開頭的浮點數

if (c == '.') {

i = 0;

do {

token\_text[i++] = c;

c = fgetc(fp);

} while (isdigit(c));

if (c == 'f') {

token\_text[i++] = c;

token\_text[i] = '\0';

return FLOAT\_CONST;

}

else {

ungetc(c, fp);

token\_text[i] = '\0';

return FLOAT\_CONST;

}

}

//宏定義、頭文件

if (c == '#') {

i = 0;

token\_text[i++] = c;

c = fgetc(fp);

if (isalpha(c)) {

do {

token\_text[i++] = c;

c = fgetc(fp);

} while (isalpha(c));

ungetc(c, fp);

token\_text[i] = '\0';

// #include <xxx> / #include ""

if (strcmp(token\_text, "#include") == 0) {

int limit = 0;

while ((c = fgetc(fp)) == ' ');

if (c == '<') {

// 判断是否等于 ‘<’ ‘"’

token\_text[i++] = c;

if (isalpha(c = fgetc(fp))) {

do {

token\_text[i++] = c;

c = fgetc(fp);

limit++;

} while (c != '>' && limit <= 30);

if (c == '>') {

token\_text[i++] = c;

c = fgetc(fp);

if (c == '\n') {

token\_text[i] = '\0';

return INCLUDE;

}

else {

return ERROR\_TOKEN;

}

}

else {

ungetc(c, fp);

return ERROR\_TOKEN;

}

}

else {

return ERROR\_TOKEN;

}

}

else if (c == '"') {

token\_text[i++] = c;

if (isalpha(c = fgetc(fp))) {

do {

token\_text[i++] = c;

c = fgetc(fp);

limit++;

} while (c != '"' && limit <= 30);

if (c == '"')

{

token\_text[i++] = c;

c = fgetc(fp);

if (c == '\n') {

token\_text[i] = '\0';

return INCLUDE;

}

else {

return ERROR\_TOKEN;

}

}

else {

ungetc(c, fp);

return ERROR\_TOKEN;

}

}

else {

return ERROR\_TOKEN;

}

}

else {

return ERROR\_TOKEN;

}

}

else if (strcmp(token\_text, "#define") == 0) {

if ((c = fgetc(fp)) == ' ') {

token\_text[i++] = c;

if (isalpha(c = fgetc(fp))) {

do {

token\_text[i] = c;

i++;

c = fgetc(fp);

} while (c != '\n');

ungetc(c, fp);

token\_text[i] = '\0';

return DEFINE;

}

else {

return ERROR\_TOKEN;

}

}

else {

return ERROR\_TOKEN;

}

}

}

else {

token\_text[i] = '\0';

return POUND;

}

}

//注釋

if (c == '/') {

i = 0;

token\_text[i++] = c;

c = fgetc(fp);

if (c == '/') {

do {

token\_text[i++] = c;

c = fgetc(fp);

} while (c != '\n');

ungetc(c, fp);

token\_text[i] = '\0';

return LINE\_COMMENT;

}

else if (c == '\*') {

token\_text[i++] = c;

c = fgetc(fp);

while (c != '\*') {

token\_text[i++] = c;

c = fgetc(fp);

if (c == '\*') {

token\_text[i++] = c;

c = fgetc(fp);

if (c == '/') {

token\_text[i++] = c;

token\_text[i] = '\0';

return BLOCK\_COMMENT;

}

else continue;

}

}

}

else {

return ERROR\_TOKEN;

}

}

//字符常量、轉義字符

if (c == '\'') {

i = 0;

token\_text[i++] = c;

c = fgetc(fp);

if (c != '\'' && c != '\\' && c != 'x') { // c 不等于 \ / '

token\_text[i++] = c;

c = fgetc(fp);

if (c == '\'') {

token\_text[i++] = c;

token\_text[i] = '\0';

return CHAR\_CONST;

}

else {

return ERROR\_TOKEN;

}

}

// '\t' "\t" '\\' 't' '\t' "\t"

else if (c == '\\') {

token\_text[i++] = c;

c = fgetc(fp);

if (c == 'n' || c == 't') {

token\_text[i++] = c;

c = fgetc(fp);

if (c == '\'') {

token\_text[i++] = c;

token\_text[i] = '\0';

return CHAR\_CONST;

}

else {

return ERROR\_TOKEN;

}

}

else if (c >= '0' && c <= '7') { //8進制轉義字符

int limit = 1;

do {

token\_text[i++] = c;

c = fgetc(fp);

limit++;

} while (c >= '0' && c <= '7' && limit <= 3);

/\*if (c >= '8' && c <= '9') printf("\t大哥这是8進制啊\n");

else \*/

if (c == '\'') {

token\_text[i++] = c;

token\_text[i] = '\0';

return CHAR\_CONST;

}

else {

return ERROR\_TOKEN;

}

}

else if (c == 'x') { //16進制轉義字符

int limit = 0;

do {

token\_text[i++] = c;

c = fgetc(fp);

limit++;

} while (is16(c) && limit <= 2);

if (c == '\'') {

token\_text[i++] = c;

token\_text[i] = '\0';

return CHAR\_CONST;

}

else {

return ERROR\_TOKEN;

}

}

else {

return ERROR\_TOKEN;

}

}

else {

return ERROR\_TOKEN;

}

}

//字符串常量

if (c == '"') {

i = 0;

do {

token\_text[i++] = c;

c = fgetc(fp);

} while (c != '"');

if (c == '"') {

token\_text[i++] = c;

token\_text[i] = '\0';

return CHAR\_CONST;

}

else {

return ERROR\_TOKEN;

}

}

char str[10] = "abc";

// ==, != ,+,++,-,\*,/,&&,||,(,),>,>=,<,<=, , ,;

switch (c) {

case'=':

i = 0;

token\_text[i++] = c;

c = fgetc(fp);

if (c == '=') {

token\_text[i++] = c;

token\_text[i] = '\0';

return EQ;

}

else {

ungetc(c, fp);

token\_text[i] = '\0';

return ASSIGN;

}

case'!':

i = 0;

token\_text[i++] = c;

c = fgetc(fp);

if (c == '=') {

token\_text[i++] = c;

token\_text[i] = '\0';

return NEQ;

}

else {

return ERROR\_TOKEN;

}

case'+':

i = 0;

token\_text[i++] = c;

c = fgetc(fp);

if (c == '+') {

token\_text[i++] = c;

token\_text[i] = '\0';

return AUTO\_INCREASING;

}

ungetc(c, fp);

token\_text[i] = '\0';

return PLUS;

case'-':

i = 0;

token\_text[i++] = c;

c = fgetc(fp);

if (c == '-') {

token\_text[i++] = c;

token\_text[i] = '\0';

return AUTO\_DECREASING;

}

ungetc(c, fp);

token\_text[i] = '\0';

return MINUS;

case'\*':

i = 0;

token\_text[i++] = c;

token\_text[i] = '\0';

return MULTIPLE;

case'/':

i = 0;

token\_text[i++] = c;

token\_text[i] = '\0';

return DIVIDE;

case'&':

i = 0;

token\_text[i++] = c;

c = fgetc(fp);

if (c == '&') {

token\_text[i++] = c;

token\_text[i] = '\0';

return AND;

}

else return ERROR\_TOKEN; //沒有位運算

case'|':

i = 0;

token\_text[i++] = c;

c = fgetc(fp);

if (c == '|') {

token\_text[i++] = c;

token\_text[i] = '\0';

return OR;

}

else return ERROR\_TOKEN; //沒有位運算

case'(':

i = 0;

token\_text[i++] = c;

token\_text[i] = '\0';

return LP;

case')':

i = 0;

token\_text[i++] = c;

token\_text[i] = '\0';

return RP;

case'>':

i = 0;

token\_text[i++] = c;

c = fgetc(fp);

if (c == '=') {

token\_text[i++] = c;

token\_text[i] = '\0';

return GREATER\_EQUAL;

}

else {

ungetc(c, fp);

token\_text[i] = '\0';

return GREATER;

}

case'<':

i = 0;

token\_text[i++] = c;

c = fgetc(fp);

if (c == '=') {

token\_text[i++] = c;

token\_text[i] = '\0';

return LESS\_EQUAL;

}

else {

ungetc(c, fp);

token\_text[i] = '\0';

return LESS;

}

case',':

i = 0;

token\_text[i++] = c;

token\_text[i] = '\0';

return COMMA;

case';':

i = 0;

token\_text[i++] = c;

token\_text[i] = '\0';

return SEMI;

case'{':

i = 0;

token\_text[i++] = c;

token\_text[i] = '\0';

return LB;

case'}':

i = 0;

token\_text[i++] = c;

token\_text[i] = '\0';

return RB;

default:

if (feof(fp)) {

return EOF;

}

else {

return ERROR\_TOKEN;

}

}

}

int check\_suffix(char c) {

if (c == 'f' || c == 'F' || c == 'u' || c == 'l' || c == 'L') {

return 0;

}

return 1;

}

1. wordRecognization.h 头文件

#pragma once

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#include <string>

int wordRecognization();

1. wordRecognization.cpp源文件

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "wordRecognization.h"

#include "gettoken.h"

using namespace tk;

extern char token\_text[30];

extern int row\_count;

extern FILE\* fp;

int wordRecognization() {

if ((fp = fopen("test1.txt", "r")) == NULL) {

printf("\t 打开文件失败!\n");

return ERROR;

}

else {

printf("\t 打开文件成功!\n");

}

printf("%-15s","\t单词类别");

printf( "%s","单词值\n");

int gettoken\_result = -1;

do {

gettoken\_result = gettoken(fp);

if (gettoken\_result != ERROR\_TOKEN) {

switch (gettoken\_result) {

case IDENT:

printf("%-15s", "\t标识符");

break;

case KEYWORD:

printf("%-15s", "\t关键字");

break;

case INT\_CONST:

printf("%-15s", "\t整型常量");

break;

case FLOAT\_CONST:

printf("%-15s", "\t浮点数常量");

break;

case CHAR\_CONST:

printf("%-15s", "\t字符常量");

break;

case INT:

printf("%-15s", "\t整型关键字");

break;

case FLOAT:

printf("%-15s", "\t浮点型关键字");

break;

case CHAR:

printf("%-15s", "\t字符型关键字");

break;

case SIGNED:

printf("%-15s", "\t有符号类型关键字");

break;

case UNSIGNED:

printf("%-15s", "\t无符号类型关键字");

break;

case VOID:

printf("%-15s", "\t无类型关键字");

break;

case ARRAY:

printf("%-15s", "\t数组");

break;

case DO:

printf("%-15s", "\t关键字");

break;

case WHILE:

printf("%-15s", "\t关键字");

break;

case BREAK:

printf("%-15s", "\t关键字");

break;

case SWITCH:

printf("%-15s", "\t关键字");

break;

case CASE:

printf("%-15s", "\t关键字");

break;

case DEFAULT:

printf("%-15s", "\t关键字");

break;

case EXTURN:

printf("%-15s", "\t关键字");

break;

case STATIC:

printf("%-15s", "\t关键字");

break;

case CONST:

printf("%-15s", "\t关键字");

break;

case TYPEDEF:

printf("%-15s", "\t关键字");

break;

case FOR:

printf("%-15s", "\t关键字");

break;

case IF:

printf("%-15s", "\t关键字");

break;

case ELSE:

printf("%-15s", "\t关键字");

break;

case INCLUDE:

printf("%-15s", "\t头文件引用");

break;

case DEFINE:

printf("%-15s", "\t宏定义");

break;

case PLUS:

printf("%-15s", "\t加号");

break;

case MINUS:

printf("%-15s", "\t减号");

break;

case MULTIPLE:

printf("%-15s", "\t乘号");

break;

case DIVIDE:

printf("%-15s", "\t除号");

break;

case CONTINUE:

printf("%-15s", "\t关键字");

break;

case RETURN:

printf("%-15s", "\t关键字");

break;

case EQ:

printf("%-15s", "\t相等号");

break;

case NEQ:

printf("%-15s", "\t不相等号");

break;

case AND:

printf("%-15s", "\t逻辑与");

break;

case OR:

printf("%-15s", "\t逻辑或");

break;

case GREATER:

printf("%-15s", "\t大于号");

break;

case GREATER\_EQUAL:

printf("%-15s", "\t大于等于号");

break;

case LESS:

printf("%-15s", "\t小于号");

break;

case LESS\_EQUAL:

printf("%-15s", "\t小于等于号");

break;

case LINE\_COMMENT:

printf("%-15s", "\t单行注释");

break;

case BLOCK\_COMMENT:

printf("%-15s", "\t块注释");

break;

case ASSIGN:

printf("%-15s", "\t賦值");

break;

case LP:

printf("%-15s", "\t左括号");

break;

case RP:

printf("%-15s", "\t右括号");

break;

case LB:

printf("%-15s", "\t左大括号");

break;

case RB:

printf("%-15s", "\t右大括号");

break;

case COMMA:

printf("%-15s", "\t逗号");

break;

case SEMI:

printf("%-15s", "\t分号");

break;

case POUND:

printf("%-15s", "\t井号");

break;

case AUTO\_INCREASING:

printf("%-15s", "\t自增运算");

break;

case AUTO\_DECREASING:

printf("%-15s", "\t自减运算");

break;

}

printf("%s\n", token\_text);

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t非法词汇!\n");

break;

}

} while (gettoken\_result != -1);

return 0;

}

1. syntaxAnalyse.h 头文件

#pragma once

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#include <string>

#include <stack>

typedef int DataType;

typedef struct NodeData {

DataType datatype;

char\* content;

}NodeData;

typedef struct VarList {

char\* var;

int size;

DataType datatype;

struct VarList\* next;

}Vars;

enum class Operator {

MULTIPLE,

DIVIDE,

PLUS,

MINUS,

LESS\_EQUAL,

GREATER\_EQUAL, //大於等於>=

LESS,

GREATER, // 大於

EQ,

NEQ,

AND,

OR,

ASSIGN, //賦值

LP, //左括号

RP, //右括号

POUND, //起止号

AUTO\_INCREASING,//自增

AUTO\_DECREASING,

};

typedef enum NodeType {

ERROR\_TYPE,

extern\_define\_sequence, //外部定义序列

extern\_define, //外部定义

extern\_variable\_define, //外部变量定义

variable, //变量

variable\_type, //变量類型

variable\_sequence, //变量序列

function\_define, //函数定义

function\_name, //函数名称

return\_type, //返回值类型

formal\_parameter\_sequence, //形参序列

formal\_parameter\_type, //参数类型

formal\_parameter, //形参

formal\_parameter\_name,

complex\_statement, //复合语句

statement\_sequence, //语句序列

local\_variable\_define\_sequence, //局部变量定义序列

local\_variable\_define, //局部变量定义

statement, //語句

expression, //表达式

operation, //操作符

expression\_ident, //表达式中的操作符

operator\_number, //操作数

IF\_ELSE\_node, //IF - ELSE结点

IF\_node, //IF结点

IF\_content,

ELSE\_node,

WHILE\_node, //WHILE结点

WHILE\_condition,

RETURN\_node, //RETURN结点

RETURN\_no\_node,

SWITCH\_node, //SWITCH结点

FOR\_node,

FOR\_node1,

FOR\_node2,

FOR\_node3,

FOR\_condition1,

FOR\_condition2,

FOR\_condition3,

FOR\_no\_condition\_node,

CONTINUE\_node,

BREAK\_node,

array\_name, //数组名

array\_type, //数组类型

Ignore,

actual\_parameter\_sequence, //實參序列

no\_formal\_parameter,

no\_complex\_statement,

no\_actual\_parameter\_sequence,

formal\_parameter\_type\_array,

formal\_parameter\_name\_array,

formal\_parameter\_array,

}NodeType;

typedef struct ASTtree {

struct ASTtree\* sibling;

struct ASTtree\* children;

NodeType type;

NodeData data;

}ASTtree;

ASTtree\* program();

ASTtree\* ExtDefList();

ASTtree\* ExtDef();

ASTtree\* ExtVarDef(char\* varname, char\* kindName);

ASTtree\* VarList(char\* varname);

ASTtree\* VarDef(char\* varname);

ASTtree\* FuncDef(char\* varname, char\* kindName);

ASTtree\* ForParaList();

ASTtree\* ForParaDef();

ASTtree\* ActParaList(int x);

ASTtree\* CompSent();

ASTtree\* ScentSeq();

ASTtree\* LocalVarDef();

ASTtree\* LocalVarDefSeq();

void freeAST(ASTtree\* root);

ASTtree\* Statement();

ASTtree\* Expression(int endsym);

void printType(NodeType type);

void PreorderTranverse(ASTtree\* root, int depth);

int syntaxAnalyse();

int GetTokenWithOutANNO();

int add\_vars( char\* varname, int cap, DataType kind);

int check\_vars(char\* varname);

int had\_return(ASTtree\* root,DataType kind);

1. syntaxAnalyse.cpp 源文件

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "gettoken.h"

#include "syntaxAnalyse.h"

using namespace tk;

extern FILE\* fp;

extern char token\_text[30];

extern char array\_capacity[10]; //存放數組容量

extern int array\_capacity\_int;

extern int row\_count; //行数计数器

int w, inRecycle = -1, countCondition = 0, countvarname = 0, isarray = 0, countScentSeq = 0, isreturn = 0, countreturn = 0, countvarreturn = 0

, hadreturn = 0, returnkind = 0, inFor = -1, inReturn = 0;

extern Vars\* varRoot, \* varTail;

char copy[100];

int syntaxAnalyse() { //程序语法正确，返回的语法树根结点指针，可遍历显示

if ((fp = fopen("test2.txt", "r")) == NULL) {

printf("\t 打开文件失败!\n");

return -1;

}

else {

row\_count = 0;

printf("\t 打开文件成功!\n");

}

ASTtree \*root = program();

PreorderTranverse(root, 0);

return 0;

}

void freeAST(ASTtree\* root) {

if (root->data.content) {

free(root->data.content);

root->data.content = NULL;

}

if (root->children) {

freeAST(root->children);

root->children = NULL;

}

if (root->sibling) {

freeAST(root->sibling);

root->sibling = NULL;

}

free(root);

root = NULL;

}

ASTtree\* program() {

ASTtree\* root = (ASTtree\*)malloc(sizeof(ASTtree));

root = ExtDefList();

if (root) {

return root;

}

else{

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

}

int GetTokenWithOutANNO(){

w = gettoken(fp);

while (w == (int)token\_kind::LINE\_COMMENT || w == (int)token\_kind::BLOCK\_COMMENT){

w = gettoken(fp);

}

return w;

}

//<外部定义序列>：：=<外部定义> <外部定义序列> | <外部定义>

ASTtree\* ExtDefList() { //处理外部定义序列，正确时，返回子树根结点指针，否则错误处理

w = GetTokenWithOutANNO();

if (w == EOF) return NULL;

while (w == (int)token\_kind::INCLUDE || w == (int)token\_kind::DEFINE) {

strcat(copy, token\_text);

strcat(copy, "\n");

w = GetTokenWithOutANNO();

}

ASTtree\* root = (ASTtree\*)malloc(sizeof(ASTtree)); //生成一个外部定义序列结点root

if (root) {

root->children = ExtDef(); //处理一个外部定义，得到一棵子树，作为root的第一棵子树

root->type = NodeType::extern\_define\_sequence;

root->data.content = NULL;

root->sibling = ExtDefList(); //得到的子树，作为root的第二棵子树

return root;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

}

//<外部定义>：： = <外部变量定义> | <函数定义>

int kind = -1;

ASTtree\* ExtDef() { //处理外部定义，正确时，返回子树根结点指针，否则返回NULL

char\* kindName = (char\*)malloc(sizeof(char) \* 30);

if (w == (int)token\_kind::INT || w == (int)token\_kind::FLOAT || w == (int)token\_kind::CHAR || w == (int)token\_kind::VOID) {

kind = w; //保存类型说明符

if (kindName) {

\*(kindName) = '\0';

strcpy(kindName, token\_text);

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t申请内存失败\n");

return NULL;

}

}

else if (w == (int)token\_kind::BREAK) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:break应出现在循环中\n");

return NULL;

}

else if (w == (int)token\_kind::CONTINUE) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:continue应出现在循环中\n");

return NULL;

}

else {//报错并返回NULL

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t外部定义出现错误:类型关键字错误\n");

return NULL;

}

w = GetTokenWithOutANNO();

if (w != (int)token\_kind::IDENT && w!= (int)token\_kind::ARRAY) { //报错并返回NULL //数组

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t外部定义出现错误:标识符错误\n");

return NULL;

}

else {

char\* varname = (char\*)malloc(sizeof(char) \* 30);

if (varname) {

\*(varname) = '\0';

if (w == (int)token\_kind::IDENT) {

strcpy(varname, token\_text); // 保存第一个变量名或函数名到tokenText0

if (check\_vars(varname) == -1) {

return NULL;

}

else {

if (add\_vars(varname, 0,kind) == 0) {

countvarname++;

}

}

}

else {

isarray = 1;

for (int i = 0; i < strlen(token\_text); i++) {

char f = token\_text[i];

if (f == '[') {

strncpy(varname, token\_text, i); // 保存第一个变量名或函数名到tokenText0 a[10]

varname[i] = '\0';

break;

}

}

if (check\_vars(varname) == -1) {

return NULL;

}

else {

if (add\_vars(varname, array\_capacity\_int,kind) == 0) {

countvarname++;

}

}

}

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t申请内存失败\n");

return NULL;

}

w = GetTokenWithOutANNO();

ASTtree\* p;

if (w != (int)token\_kind::LP) p = ExtVarDef(varname,kindName); //调用外部变量定义子程序

else {

p = FuncDef(varname,kindName); //调用函数定义子程序

}

free(varname);

free(kindName);

return p; // 如果返回值p非空，表示成功完成一个外部定义的处理，返回p

}

}

// <外部变量定义>：：=<类型说明符> <变量序列> ；

ASTtree\* ExtVarDef(char \*varname,char \*kindName) {

ASTtree\* root = (ASTtree\*)malloc(sizeof(ASTtree));//生成外部变量定义结点

if (root) {

root->children = NULL;

root->sibling = NULL;

root->data.content = NULL;

root->type = NodeType::extern\_variable\_define;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

if (kind != -1) {

ASTtree\* p = (ASTtree\*)malloc(sizeof(ASTtree));//根据已读入的外部变量的类型，生成外部变量类型结点

if (p) {

p->children = NULL;

p->sibling = NULL;

char\* name = (char\*)malloc(sizeof(char) \* 30);

if (name) {

\*(name) = '\0';

strcpy(name, kindName);

}

p->data.content = name;

if (isarray == 1) {

p->type = NodeType::array\_type;

}

else {

p->type = NodeType::variable\_type;

}

root->children = p; //作为root的第一个孩子

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:外部变量类型节点的指针为空\n");

return NULL;

}

root->sibling = VarList(varname);

return root;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:错误定义或未指定外部变量的类型\n");

return NULL;

}

}

//<变量序列>：： = <变量>,<变量序列> | <变量>;

ASTtree\* VarList(char\* varname) {

ASTtree\* root = (ASTtree\*)malloc(sizeof(ASTtree));//生成`变量序列结点`

if (root) {

root->children = NULL;

root->sibling = NULL;

root->data.content = NULL;

root->type = NodeType::variable\_sequence;

root->children = VarDef(varname);

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

if (w == (int)token\_kind::IDENT) {

w = GetTokenWithOutANNO();

}

if (w != (int)token\_kind::COMMA && w != (int)token\_kind::SEMI) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:变量定义错误,可能在逗号或分号附近发生语句错误\n");

freeAST(root); //报错，释放root为根的全部结点，返回空指针

return NULL;

}

if (w == (int)token\_kind::SEMI) {

//w = GetTokenWithOutANNO();

return root;

//返回根结点root；

}

else if (w == (int)token\_kind::COMMA) {

w = GetTokenWithOutANNO(); //w == , eg. int i,j;

if (w != (int)token\_kind::IDENT) {

//报错，释放root为根的全部结点，返回空指针

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t外部定义出现错误:变量定义错误\n");

freeAST(root);

return NULL;

}

root->sibling = VarList(token\_text);

}

return root;

}

//<变量>

ASTtree\* VarDef(char\* varname) {

ASTtree\* p = (ASTtree\*)malloc(sizeof(ASTtree));//生成`变量结点`

if (p) {

p->children = NULL;

p->sibling = NULL;

char\* name = (char\*)malloc(sizeof(char) \* 30);

if (name) {

name[0] = '\0';

strcpy(name, varname); //由保存在varname的第一个变量名生成一个`变量名结点`

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t申请内存失败\n");

return NULL;

}

p->data.datatype = kind;

p->data.content = name;

if (isarray == 1) {

p->type = NodeType::array\_name;

isarray = 0;

}

else {

p->type = NodeType::variable;

}

return p;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

}

//<形式参数序列>：：=<形式参数> ， <形式参数序列> | <形式参数>

ASTtree\* ForParaList() { //处理参数部分到反小括号结束，调用形参子程序，得到参数部分的子树根指针，

w = GetTokenWithOutANNO();

if (w == (int)token\_kind::RP) {

ASTtree\* root = (ASTtree\*)malloc(sizeof(ASTtree));

if (root) {

root->children = NULL;

root->sibling = NULL;

root->data.content = NULL;

root->type = NodeType::no\_formal\_parameter;

return root;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

return NULL;

}

else if (w == (int)token\_kind::COMMA) {

w = GetTokenWithOutANNO();

}

// 如果后面跟着的是<形式参数>

ASTtree\* root = (ASTtree\*)malloc(sizeof(ASTtree));//生成`形式参数序列结点`

if (root) {

root->children = NULL;

root->sibling = NULL;

root->data.content = NULL;

root->type = NodeType::formal\_parameter\_sequence;

root->children = ForParaDef();

root->sibling = ForParaList();

return root;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

}

//<形式参数>：： = <类型说明符> 标识符

ASTtree\* ForParaDef() {

if (w != (int)token\_kind::INT && w != (int)token\_kind::FLOAT && w != (int)token\_kind::CHAR && w != (int)token\_kind::VOID) {//无参函数得到NULL，该子树作为root的第二棵子树；

printf("\t参数错误\n");

return NULL;

}

kind = w;

ASTtree\* p = (ASTtree\*)malloc(sizeof(ASTtree));

if (p) {

p->data.content = NULL;

p->children = NULL;

p->sibling = NULL;

char\* name = (char\*)malloc(sizeof(char) \* 30);

if (name) {

\*(name) = '\0';

strcpy(name, token\_text);

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t申请内存失败\n");

return NULL;

}

p->data.content = name;

p->type = NodeType::formal\_parameter\_type;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t申请内存失败\n");

return NULL;

}

w = GetTokenWithOutANNO();

if(w == (int)token\_kind::IDENT){

char\* name = (char\*)malloc(sizeof(char)\*30);

if (name) {

\*(name) = '\0';

strcpy(name, token\_text);

add\_vars(name, 0,kind );

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t申请内存失败\n");

return NULL;

}

ASTtree\* q = (ASTtree\*)malloc(sizeof(ASTtree));

if (q) {

q->data.content = name;

q->type = NodeType::formal\_parameter\_name;

q->data.datatype = kind;

q->children = NULL;

q->sibling = NULL;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

ASTtree\* root = (ASTtree\*)malloc(sizeof(ASTtree));

if (root) {

root->data.content = NULL;

root->children = p;

root->sibling = q;

root->type = NodeType::formal\_parameter;

return root;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

}

else if (w == (int)token\_kind::ARRAY) {

char\* varname = (char\*)malloc(sizeof(char) \* 30);

if (varname) {

\*(varname) = '\0';

for (int i = 0; i < strlen(token\_text); i++) {

char f = token\_text[i];

if (f == '[') {

strncpy(varname, token\_text, i); // 保存第一个变量名或函数名到varname

varname[i] = '\0';

break;

}

}

if (check\_vars(varname) == -1) {

return NULL;

}

else {

if (add\_vars(varname, array\_capacity\_int, kind) == 0) {

countvarname++;

}

}

p->type = NodeType::formal\_parameter\_type\_array;

ASTtree\* q = (ASTtree\*)malloc(sizeof(ASTtree));

if (q) {

q->data.content = varname;

q->type = NodeType::formal\_parameter\_name\_array;

q->data.datatype = kind;

q->children = NULL;

q->sibling = NULL;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

ASTtree\* root = (ASTtree\*)malloc(sizeof(ASTtree));

if (root) {

root->data.content = NULL;

root->children = p;

root->sibling = q;

root->type = NodeType::formal\_parameter\_array;

return root;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:识别符和关键字相同\n");

return NULL;

}

}

//<实参序列>：：=<表达式> <实参序列> | <空>

ASTtree\* ActParaList(int x) {

ASTtree\* root = (ASTtree\*)malloc(sizeof(ASTtree));

if (root) {

root->children = NULL;

root->sibling = NULL;

root->data.content = NULL;

root->type = actual\_parameter\_sequence;

if (w == (int)token\_kind::RP) {

root->type = no\_actual\_parameter\_sequence;

return root;

}

ASTtree\* t = Expression((int)token\_kind::COMMA);

if (t) {

x = ftell(fp);

w = GetTokenWithOutANNO();

root->children = t;

root->sibling = ActParaList(x);

if (root->sibling) {

if (root->sibling->type == Ignore) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:实参不完整\n");

exit(-1);

}

}

}

else {

fseek(fp, x, SEEK\_SET);

w = GetTokenWithOutANNO();

ASTtree\* t = Expression((int)token\_kind::RP);

root->children = t;

}

return root;

}

return NULL;

}

//<复合语句>：：={ <局部变量定义序列> <语句序列> }

//（复合语句）子程序

ASTtree\* CompSent() { //正大括号，则调用函数体（复合语句）子程序，得到函数体子树根指针，其它符号表示有错。

ASTtree\* root = (ASTtree\*)malloc(sizeof(ASTtree));

if (root) {

root->children = NULL; //root = 生成复合语句结点；注意其中局部变量说明和语句序列都可以为空

root->sibling = NULL;

root->data.content = NULL;

root->type = NodeType::Ignore;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

root->children = LocalVarDefSeq();//调用处理局部变量说明序列子程序

root->sibling = ScentSeq();//调用处理'语句序列子程序'，返回子树根结点指针，作为root的第2个孩子

if (w != (int)token\_kind::RB ){

freeAST(root); //返回空指针，报错并释放结点

return NULL;

}

if (root->children || root->sibling) {

root->type = NodeType::complex\_statement;

}

else {

root->type = NodeType::no\_complex\_statement;

}

return root; // 返回复合语句的子树根指针。

}

//<函数定义>：： = <类型说明符> <函数名>（<形式参数序列>）<复合语句>

ASTtree\* FuncDef(char\* varname,char \*kindName) {

ASTtree\* root = (ASTtree\*)malloc(sizeof(ASTtree)); //生成函数定义结点root；

if (root) {

root->children = NULL;

root->sibling = NULL;

root->data.content = NULL;

root->data.content = NULL;

root->type = NodeType::function\_define;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

ASTtree\* p = (ASTtree\*)malloc(sizeof(ASTtree)); //生成返回值类型结点，作为root的第一个孩子

if (p) {

p->children = NULL;

p->sibling = NULL;

char\* name = (char\*)malloc(sizeof(char) \* 30);

if (name) {

\*(name) = '\0';

strcpy(name, kindName);

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t申请内存失败\n");

return NULL;

}

p->data.content = name;

p->type = NodeType::return\_type;

if (kind != (int)token\_kind::VOID) {

isreturn = 1;

}

else {

isreturn = 2;

}

p->data.datatype = kind;

returnkind = kind;

root->children = p;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

ASTtree\* q = (ASTtree\*)malloc(sizeof(ASTtree)); //生成函數名类型结点，作为root的第二个孩子

if (q) {

q->children = NULL;

q->sibling = NULL;

char\* name = (char\*)malloc(sizeof(char) \* 30);

if (name) {

name[0] = '\0';

strcpy(name, varname);

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t申请内存失败\n");

return NULL;

}

q->data.content = name;

q->type = NodeType::function\_name;

q->data.datatype = kind;

q->children = ForParaList();

}

root->sibling = q;

if (!root->sibling) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t函数定义出现错误\n");

return NULL;

}

else {

w = GetTokenWithOutANNO();

if (w == (int)token\_kind::SEMI) { //读入符号，如果分号，就是函数原型声明，函数体子树为空；

root->sibling->sibling = NULL;

}

else if (w == (int)token\_kind::LB) { //得到的函数体子树作为root的第三棵子树

ASTtree\* t = (ASTtree\*)malloc(sizeof(ASTtree));

t = CompSent();

root->sibling->sibling = t;

//如果函数要返回值，但没有返回值

countreturn = 0;

countvarreturn = 0;

hadreturn = 0;

if (isreturn == 1 || isreturn == 2 || isreturn == 3) {

switch (had\_return(t, returnkind)) {

case -1:

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t函数缺少返回值\n");

return NULL;

break;

case -2:

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t返回值类型错误，应为%s类型\n", kindName);

return NULL;

break;

case -3:

if (isreturn != 2) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t没有此返回值\n");

return NULL;

break;

}

}

isreturn = 0;

}

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t函数定义出现错误\n");

return NULL;

}

return root; // 返回root；

}

}

//<语句序列>：：=<语句><语句序列> | <空>

ASTtree\* ScentSeq() {

//要考虑语句序列为空的情况

ASTtree\* r1 = (ASTtree\*)malloc(sizeof(ASTtree));

r1 = Statement(); //调用处理一条语句的子程序；返回其子树根指针r1；

if (r1 == NULL) {

//否则表示语句序列已结束

return NULL; //返回NULL

}

else { //初始化子树，根指针root = NULL；

ASTtree\* root = (ASTtree\*)malloc(sizeof(ASTtree)); // 生成语句序列的结点root

if (root) {

root->children = NULL;

root->sibling = NULL;

root->data.content = NULL;

root->type = NodeType::statement\_sequence;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

root->children = r1; //root->第1孩子 = r1;

w = GetTokenWithOutANNO();

root->sibling = ScentSeq(); //root->第2孩子 = 递归调用处理语句序列子程序后的返回值；

return root; //返回root；

}

}

//<局部变量定义序列>：： = <局部变量定义> <局部变量定义序列> | <空>

ASTtree\* LocalVarDefSeq() {

ASTtree\* root = (ASTtree\*)malloc(sizeof(ASTtree));

if (root) {

w = GetTokenWithOutANNO();

if (w == (int)token\_kind::INT || w == (int)token\_kind::FLOAT || w == (int)token\_kind::CHAR || w == (int)token\_kind::VOID) {

root->children = LocalVarDef();

root->type = NodeType::local\_variable\_define\_sequence;

root->data.content = NULL;

root->sibling = LocalVarDefSeq();

return root;

}

else {

return NULL; //局部变量定义为空

}

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

}

//<局部变量定义>：： = <类型说明符> <变量序列>

ASTtree \*LocalVarDef(){

char\* \_type = (char\*)malloc(sizeof(char) \* 30);

if(\_type) \*(\_type) = '\0';

else {

return NULL;

}

kind = w;

ASTtree\* p = (ASTtree\*)malloc(sizeof(ASTtree)); // 得到返回的子树根结点作为root的第一个孩子

if (p) {

p->children = NULL;

p->sibling = NULL;

p->data.content = NULL;

p->type = NodeType::local\_variable\_define;

p->data.datatype = w;

strcpy(\_type,token\_text);

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

w = GetTokenWithOutANNO();

if (w != (int)token\_kind::IDENT && w != (int)token\_kind::ARRAY) { //报错并返回NULL

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t局部变量定义出现错误:标识符错误\n");

return NULL;

}

else {

char\* varname = (char\*)malloc(sizeof(char) \* 30);

if (varname) {

\*(varname) = '\0';

if (w == (int)token\_kind::IDENT) {

strcpy(varname, token\_text); // 保存第一个变量名或函数名到tokenText0

if (check\_vars(varname) == -1) {

return NULL;

}

else {

if (add\_vars(varname, 0, kind) == 0) {

countvarname++;

}

}

}

else {

isarray = 1;

for (int i = 0; i < strlen(token\_text); i++) {

char f = token\_text[i];

if (f == '[') {

strncpy(varname, token\_text, i); // 保存第一个变量名或函数名到tokenText0 a[10]

varname[i] = '\0';

break;

}

}

if (check\_vars(varname) == -1) {

return NULL;

}

else {

if (add\_vars(varname, array\_capacity\_int, kind) == 0) {

countvarname++;

}

}

}

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t申请内存失败\n");

return NULL;

}

// int i=0;

ASTtree\* q = (ASTtree\*)malloc(sizeof(ASTtree));

if (q) {

q->children = NULL;

q->sibling = NULL;

q->data.content = \_type;

if (isarray == 1) {

q->type = NodeType::array\_type;

}

else {

q->type = NodeType::variable\_type;

}

q->data.datatype = w;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

p->children = q;

p->sibling = VarList(token\_text);

return p;

}

}

/\*<语句>::= <表达式>； | return <表达式>

| if （<表达式>）<语句>

| if （<表达式>）<语句> else <语句>\*/

ASTtree \*Statement(){

ASTtree\* root = (ASTtree\*)malloc(sizeof(ASTtree));

if (root) {

root->children = NULL;

root->sibling = NULL;

root->data.content = NULL;

}

ASTtree \*p = NULL,\*q = NULL,\*t = NULL,\*p2 = NULL,\*q2 = NULL,\*t2 = NULL,\*f = NULL,\*root2 = NULL;

int seek\_index = 0;

switch (w){

case (int)token\_kind::IF://分析条件语句

w = GetTokenWithOutANNO();

if (w != (int)token\_kind::LP){

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\tif语句出现错误\n");

return NULL;//报错并返回空

}

w = GetTokenWithOutANNO();

p = Expression((int)token\_kind::RP); //调用处理表达式的子程序（结束符号为反小括号），正确时得到条件表达式子树结点指针

w = GetTokenWithOutANNO();

q = Statement(); //调用处理一条语句的子程序，得到IF子句的子树根指针。

seek\_index = ftell(fp);

w = GetTokenWithOutANNO();

if (w == (int)token\_kind::ELSE){

w = GetTokenWithOutANNO();

t = Statement(); //调用处理一条语句的子程序，得到ELSE子句的子树根指针。

// 生成IF - ELSE结点，下挂条件、IF子句、ELSE子句3棵子树

if (root) {

root->children = p; //IF条件

root->sibling = q; //IF子句

root->sibling->type = IF\_content;

root->sibling->sibling = t;

root->sibling->sibling->type = ELSE\_node;

root->data.content = NULL;

root->type = NodeType::IF\_ELSE\_node;

return root;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

}

else {//生成IF结点，下挂条件、IF子句2棵子树

fseek(fp, seek\_index, SEEK\_SET);

if (root) {

root->children = p;

root->sibling = q;

root->data.content = NULL;

root->type = NodeType::IF\_node;

return root;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

}

break;

case (int)token\_kind::LB:

if (root) {

root = CompSent();//调用处理复合语句子程序,返回得到的子树根指针

return root;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

break;

case (int)token\_kind::WHILE:

inRecycle = 1;

inFor = 0;

w = GetTokenWithOutANNO();

if (w != (int)token\_kind::LP) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\tif语句出现错误\n");

return NULL;//报错并返回空

}

w = GetTokenWithOutANNO();

p = Expression((int)token\_kind::RP);

if (p) {

p2 = (ASTtree\*)malloc(sizeof(ASTtree));

if (p2) {

p2->children = p;

p2->sibling = NULL;

p2->data.content = NULL;//调用处理表达式的子程序（结束符号为反小括号），正确时得到条件表达式子树结点指针

p2->type = NodeType::WHILE\_condition;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

w = GetTokenWithOutANNO();

q = Statement(); //调用处理一条语句的子程序，得到WHILE子句的子树根指针。處理{ }中的東西

inRecycle = -1;

inFor = -1;

if (root) {// 生成WHILE结点，下挂条件、WHILE子句 2棵子树

root->children = p2;

root->sibling = q;

root->data.content = NULL;

root->type = NodeType::WHILE\_node;

return root;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

}

else {

return NULL;

}

break;

case (int)token\_kind::IDENT:

case (int)token\_kind::INT\_CONST:

case (int)token\_kind::FLOAT\_CONST:

case (int)token\_kind::CHAR\_CONST:

p = Expression((int)token\_kind::SEMI); // 调用表达式处理子程序（结束符号为分号）；

if (root) {

root->data.content = NULL;

root->children = p;

root->sibling = NULL;

root->type = NodeType::statement;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

return root;

break;

case (int)token\_kind::LP: //各种表达式语句，含赋值等，形式为表达式以分号结束 ????

return Expression((int)token\_kind::SEMI);

case (int)token\_kind::RB: //语句序列结束符号，

return NULL;

break;

case (int)token\_kind::FOR:

inRecycle = 1;

inFor = 1;

w = GetTokenWithOutANNO();

if (w != (int)token\_kind::LP) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\tif语句出现错误\n");

return NULL;//报错并返回空

}

w = GetTokenWithOutANNO();

p = Expression((int)token\_kind::SEMI);

w = GetTokenWithOutANNO();

q = Expression((int)token\_kind::SEMI);

w = GetTokenWithOutANNO();

t = Expression((int)token\_kind::RP);

countCondition = 0;

root2 = (ASTtree\*)malloc(sizeof(ASTtree)); //FOR的條件結點

if (root2) {

root2->data.content = NULL;

p2 = (ASTtree\*)malloc(sizeof(ASTtree));

if (p2 && p) {

p2->children = p;

p2->sibling = NULL;

p2->data.content = NULL;

if (p->type == NodeType::Ignore) {

p2->type = NodeType::Ignore;

}

else {

p2->type = NodeType::FOR\_condition1;

countCondition++;

}

root2->children = p2;

}

else {

free(p2);

p2 = NULL;

exit(-1);

}

q2 = (ASTtree\*)malloc(sizeof(ASTtree));

if (q2 && q) {

q2->children = q;

q2->sibling = NULL;

q2->data.content = NULL;

if (q->type == NodeType::Ignore) {

q2->type = NodeType::Ignore;

}

else {

q2->type = NodeType::FOR\_condition2;

countCondition++;

}

root2->sibling = q2;

}

else {

free(q2);

q2 = NULL;

exit(-1);

}

t2 = (ASTtree\*)malloc(sizeof(ASTtree));

if (t2 && t) {

t2->children = t;

t2->sibling = NULL;

t2->data.content = NULL;

if (t->type == NodeType::Ignore) {

t2->type = NodeType::Ignore;

}

else {

t2->type = NodeType::FOR\_condition3;

countCondition++;

}

root2->sibling->sibling = t2;

}

else{

free(t2);

t2 = NULL;

exit(-1);

}

if (countCondition == 1) {

root2->type = NodeType::FOR\_node1;

}

else if (countCondition == 2) {

root2->type = NodeType::FOR\_node2;

}

else if(countCondition == 3) {

root2->type = NodeType::FOR\_node3;

}

else if (countCondition == 0) {

root2->type = NodeType::FOR\_no\_condition\_node;

}

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

w = GetTokenWithOutANNO();

f = Statement();

if (f) {

if (root) {

root->children = root2;

root->sibling = f;

root->data.content = NULL;

root->type = FOR\_node;

inRecycle = 0;

inFor = -1;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

}

return root;

break;

case (int)token\_kind::CONTINUE:

if (inRecycle == 1) {

w = GetTokenWithOutANNO();

if (w != (int)token\_kind::SEMI) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:语句错误\n");

return NULL;

}

else {

if (root) {

root->data.content = NULL;

root->children = NULL;

root->sibling = NULL;

root->type = CONTINUE\_node;

return root;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

}

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:continue应出现在循环中\n");

return NULL;

}

break;

case (int)token\_kind::BREAK:

if (inRecycle == 1) {

w = GetTokenWithOutANNO();

if (w != (int)token\_kind::SEMI) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:语句错误\n");

return NULL;

}

else {

if (root) {

root->type = BREAK\_node;

root->data.content = NULL;

root->children = NULL;

root->sibling = NULL;

return root;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

}

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:break应出现在循环中\n");

return NULL;

}

break;

case (int)token\_kind::RETURN:

if (isreturn == 0) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:函数没有返回值\n");

return NULL;

}

inReturn = 1;

w = GetTokenWithOutANNO();

p = Statement();

inReturn = 0;

if (root) {

root->children = p;

root->data.content = NULL;

root->sibling = NULL;

root->type = NodeType::RETURN\_node;

return root;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

break;

default:

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:表达式错误\n");

return NULL;

break;

}

}

//运算符优先级

char precede[18][18] = {

{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1},

{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1},

{-1,-1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1},

{-1,-1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1},

{-1,-1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1},

{-1,-1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1},

{-1,-1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1},

{-1,-1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1},

{-1,-1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1},

{-1,-1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1},

{-1,-1, -1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, -1, 1, 1, -1, -1},

{-1,-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 1, 1, -1, 1, 1, -1, -1},

{-1,-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 1, 1, -1, -1},

{-1,-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 0, -1, -1, -1},

{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},

{-1,-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 1, 0, 1, 1},

{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1},

{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, 1, 1, -1, -1}

};

/\*<表达式>：： = <表达式> +<表达式> | <表达式> -<表达式> | <表达式> \*<表达式>

| <表达式> / <表达式> | INT\_CONST | IDENT | IDENT(<实参序列>)

| <表达式> == <表达式> | <表达式> != <表达式> | <表达式> > <表达式>

| <表达式> > <表达式> | <表达式> >= <表达式> | <表达式> < <表达式>

| <表达式> <= <表达式> | 标识符 = <表达式> \*/

int parentheses = 0; //纪录左括号是否有对应的右括号

ASTtree\* Expression(int endsym) { //调用处理表达式的子程序（结束符号为反小括号），正确时得到条件表达式子树结点指针

//调用该算法时，在调用者已经读入了第一个单词在w中

//表达式结束符号endsym可以是分号，如表达式语句，可以是反小括号，作为条件时使用

// 定义运算符栈 op; 并初始化，将起止符#入栈

std::stack<ASTtree\*> op\_stack;

ASTtree\* t = (ASTtree\*)malloc(sizeof(ASTtree));

if (t) {

t->type = NodeType::operation;

t->children = NULL;

t->sibling = NULL;

t->data.datatype = (int)Operator::POUND;

op\_stack.push(t);

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

// 定义操作数栈opn，元素是结点的指针

std::stack<ASTtree\*> opn\_stack;

std::stack<ASTtree\*> deal\_stack;

int error = 0, count = 0, c = 0;

ASTtree \*operator\_node, \*operator\_num1, \*operator\_num2,\*deal\_root = NULL;

if (inRecycle == 1) {

if (inFor == 1 && w == endsym) {

ASTtree\* p = (ASTtree\*)malloc(sizeof(ASTtree));

if (p) {

p->type = Ignore;

p->children = NULL;

p->sibling = NULL;

p->data.content = NULL;

p->data.datatype = w;

return p;

}

}

else if (inFor == 0 && w == endsym) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:while循环缺少条件\n");

return NULL;

}

}

while ((w != (int)Operator::POUND || op\_stack.top()->data.datatype != (int)Operator::POUND) && !error) //当运算符栈栈顶不是起止符号，并没有错误时

{

if (count >= 1 && w != (int)Operator::POUND) {

w = GetTokenWithOutANNO();

}

count++;

if (w == (int)token\_kind::IDENT || w == (int)token\_kind::INT\_CONST || w == (int)token\_kind::CHAR\_CONST || w == (int)token\_kind::FLOAT\_CONST)//w是标识符或常数等操作数时

{

if (w == (int)token\_kind::IDENT) {

Vars\* p = varRoot->next;

bool flag = false;

while (p) {

if (!strcmp(p->var,token\_text)) {

flag = true;

break;

}

p = p->next;

}

if (!flag) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:没有%s这个标识符\n",token\_text);

return NULL;

}

char\* identName = (char\*)malloc(sizeof(char) \* 30);

if (identName) {

identName[0] = '\0';

strcpy(identName, token\_text);

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

c = fgetc(fp);

if (c == '(') {

int seek\_index = ftell(fp);

w = GetTokenWithOutANNO();

ASTtree\* node = ActParaList(seek\_index);

if (node) {

node->data.content = identName;

opn\_stack.push(node);

w = (int)token\_kind::RP;

continue;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:实参序列 %s 表达式出现错误！\n", token\_text);

exit(-1);

}

}

else {

ungetc(c, fp);

}

}

ASTtree\* p = (ASTtree\*)malloc(sizeof(ASTtree)); // 根据w生成一个结点，结点指针进栈opn

if (p) {

p->type = NodeType::operator\_number;

p->children = NULL;

p->sibling = NULL;

p->data.content = NULL;

if (isreturn == 1) {

isreturn = 3;

}

p->data.datatype = w;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:指针为空\n");

return NULL;

}

if (inReturn == 1) {

if (isreturn == 2 && w == endsym) {

ASTtree\* p = (ASTtree\*)malloc(sizeof(ASTtree));

if (p) {

p->type = RETURN\_no\_node;

p->children = NULL;

p->sibling = NULL;

p->data.content = NULL;

return p;

}

}

else if (isreturn == 1 && w == endsym) {

return NULL;

}

else if (isreturn == 1 && w != endsym) {

}

else if (isreturn == 3) {

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:void函数不应该有返回值\n");

return NULL;

}

}

char\* number = (char\*)malloc(sizeof(char) \* 30);

if (number) {

number[0] = '\0';

strcpy(number, token\_text);

p->data.content = number;

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t內存分配失败\n");

return NULL;

}

opn\_stack.push(p);

}

else if (w == endsym) {

w = (int)token\_kind::POUND;//遇到结束标记分号，w被替换成#

}

else if (w >= (int)token\_kind::MULTIPLE && w <= (int)token\_kind::AUTO\_DECREASING) //w是运算符,按照excel排

{

Operator cal\_op = Operator::POUND;

char\* buf = NULL;

switch (precede[op\_stack.top()->data.datatype][w]) {

case -1:

//解決括號

if (w == LP) {

w = GetTokenWithOutANNO();

parentheses++;

deal\_root = Expression(RP);

parentheses--;

if (deal\_root) {

opn\_stack.push(deal\_root);

w = (int)token\_kind::IDENT;

}

else {

return NULL;

}

}

else {

operator\_node = (ASTtree\*)malloc(sizeof(ASTtree));

if (!operator\_node) return NULL;

operator\_node->type = NodeType::operation;

operator\_node->children = NULL;

operator\_node->sibling = NULL;

operator\_node->data.datatype = w;

op\_stack.push(operator\_node);

}

break;

case 0:

if (op\_stack.size() == 0) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:表达式错误\n");

return NULL;

}

operator\_node = op\_stack.top();

op\_stack.pop();

if (!operator\_node) {

error++;

}

w = GetTokenWithOutANNO();

break; //去括号

case 1:

//Wendy Do

if (w == (int)token\_kind::AUTO\_INCREASING || w == (int)token\_kind::AUTO\_DECREASING) {

if (opn\_stack.size() == 0) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:表达式错误\n");

return NULL;

}

operator\_num2 = opn\_stack.top();

opn\_stack.pop();

if (!operator\_num2) {

error++;

}

operator\_num1 = NULL;

operator\_node = (ASTtree\*)malloc(sizeof(ASTtree));

if (!operator\_node) return NULL;

operator\_node->type = NodeType::operation;

operator\_node->children = NULL;

operator\_node->sibling = NULL;

operator\_node->data.datatype = w;

}

else {

if (w == (int)token\_kind::RP && parentheses == 0) {

if (endsym != (int)token\_kind::COMMA) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:表达式括号匹配错误\n");

}

return NULL;

}

if (opn\_stack.size() == 0) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:表达式错误\n");

return NULL;

}

operator\_num2 = opn\_stack.top();

opn\_stack.pop();

if (!operator\_num2) {

error++;

}

if (opn\_stack.size() == 0) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:表达式错误\n");

return NULL;

}

operator\_num1 = opn\_stack.top();

opn\_stack.pop();

if (!operator\_num1) {

error++;

}

if (op\_stack.size() == 0) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:表达式错误\n");

return NULL;

}

operator\_node = op\_stack.top();

op\_stack.pop();

if (!operator\_node) {

error++;

}

}

//根据运算符栈退栈得到的运算符t和操作数的结点指针t1和t2，

//完成建立生成一个运算符的结点，结点指针进栈opn

if (!operator\_node) return NULL;

/\*Max Do\*/

if (!operator\_num1) {

operator\_node->children = operator\_num2;

}

else {

operator\_node->children = operator\_num1;

operator\_node->sibling = operator\_num2;

}

/\*Max End\*/

cal\_op = (Operator)operator\_node->data.datatype;

buf = (char\*)malloc(sizeof(char) \* 30);

\*(buf) = '\0';

if (!buf) {

printf("\tmalloc分配空间失败\n");

return NULL;

}

switch (cal\_op)

{

case Operator::MULTIPLE:

if (!operator\_num1 || !operator\_num2) return NULL;

//operator\_node->data.content = \_itoa(atoi(operator\_num1->data.content) \* atoi(operator\_num2->data.content), buf, 10);

operator\_node->data.content = strcpy(buf, "\*");

break;

case Operator::DIVIDE:

if (!operator\_num2) return NULL;

if (atoi(operator\_num2->data.content) == 0) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:分母不能是零\n");

error++;

free(buf);

//fun(a + b);

}

else {

if (!operator\_num1) return NULL;

//operator\_node->data.content = \_itoa(atoi(operator\_num1->data.content) / atoi(operator\_num2->data.content), buf, 10);

operator\_node->data.content = strcpy(buf, "/");

}

break;

case Operator::PLUS:

if (!operator\_num1 || !operator\_num2) return NULL;

//operator\_node->data.content = \_itoa(atoi(operator\_num1->data.content) + atoi(operator\_num2->data.content), buf, 10);

operator\_node->data.content = strcpy(buf, "+");

break;

case Operator::MINUS:

if (!operator\_num1 || !operator\_num2) return NULL;

//operator\_node->data.content = \_itoa(atoi(operator\_num1->data.content) - atoi(operator\_num2->data.content), buf, 10);

operator\_node->data.content = strcpy(buf, "-");

break;

case Operator::LESS\_EQUAL:

if (!operator\_num1 || !operator\_num2) return NULL;

// operator\_node->data.content = \_itoa(atoi(operator\_num1->data.content) <= atoi(operator\_num2->data.content), buf, 10);

operator\_node->data.content = strcpy(buf, "<=");

break;

case Operator::GREATER\_EQUAL:

if (!operator\_num1 || !operator\_num2) return NULL;

//operator\_node->data.content = \_itoa(atoi(operator\_num1->data.content) >= atoi(operator\_num2->data.content), buf, 10);

operator\_node->data.content = strcpy(buf, ">=");

break;

case Operator::LESS:

if (!operator\_num1 || !operator\_num2) return NULL;

//operator\_node->data.content = \_itoa(atoi(operator\_num1->data.content) < atoi(operator\_num2->data.content), buf, 10);

operator\_node->data.content = strcpy(buf, "<");

break;

case Operator::GREATER:

if (!operator\_num1 || !operator\_num2) return NULL;

//operator\_node->data.content = \_itoa(atoi(operator\_num1->data.content) > atoi(operator\_num2->data.content), buf, 10);

operator\_node->data.content = strcpy(buf, ">");

break;

case Operator::EQ:

if (!operator\_num1 || !operator\_num2) return NULL;

//operator\_node->data.content = \_itoa(atoi(operator\_num1->data.content) == atoi(operator\_num2->data.content), buf, 10);

operator\_node->data.content = strcpy(buf, "==");

break;

case Operator::NEQ:

if (!operator\_num1 || !operator\_num2) return NULL;

//operator\_node->data.content = \_itoa(atoi(operator\_num1->data.content) != atoi(operator\_num2->data.content), buf, 10);

operator\_node->data.content = strcpy(buf, "!=");

break;

case Operator::AND:

if (!operator\_num1 || !operator\_num2) return NULL;

//operator\_node->data.content = \_itoa(atoi(operator\_num1->data.content) && atoi(operator\_num2->data.content), buf, 10);

operator\_node->data.content = strcpy(buf, "&&");

break;

case Operator::OR:

if (!operator\_num1 || !operator\_num2) return NULL;

//operator\_node->data.content = \_itoa(atoi(operator\_num1->data.content) || atoi(operator\_num2->data.content), buf, 10);

operator\_node->data.content = strcpy(buf, "||");

break;

case Operator::ASSIGN:

if (!operator\_num1 || !operator\_num2) return NULL;

//operator\_node->data.content = operator\_num2->data.content;

operator\_node->data.content = strcpy(buf, "=");

break;

case Operator::AUTO\_INCREASING:

if (!operator\_num2) return NULL;

operator\_node->data.content = strcpy(buf, "++");

break;

case Operator::AUTO\_DECREASING:

if (!operator\_num2) return NULL;

operator\_node->data.content = strcpy(buf, "--");

break;

default:

free(buf);

break;

}

opn\_stack.push(operator\_node);

if (w == endsym) { //遇到结束标记）

w = (int)token\_kind::POUND; //，w被替换成#

}

break;

default:

if (w == endsym) { //遇到结束标记）

w = (int)token\_kind::POUND; //，w被替换成#

}

else {

error = 1;

}

break;

}

}

else {

error = 1;

}

}

if (opn\_stack.size() == 1 && op\_stack.top()->data.datatype == (int)Operator::POUND && !error) {

free(t);

t = NULL;

return opn\_stack.top(); //return操作数栈唯一的这个结点指针; //成功返回表达式语法树的根结点指针

}

else {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:表达式错误\n");

free(t);

t = NULL;

return NULL; //表达式分析有错

}

return NULL;

}

//打印出类型

void printType(NodeType type) {

switch (type) {

case ERROR\_TYPE:

printf("错误类型\n");

break;

case extern\_define:

printf("外部定义:\n");

break;

case extern\_variable\_define:

printf("外部变量定义:\n");

break;

case variable:

printf("变量:");

break;

case variable\_type:

printf("变量类型: ");

break;

case function\_define:

printf("函数定义:\n");

break;

case return\_type:

printf("返回值类型: ");

break;

case formal\_parameter:

printf("函数形参:\n");

break;

case formal\_parameter\_array:

printf("函数形参:\n");

break;

case formal\_parameter\_name:

printf("参数名: ");

break;

case formal\_parameter\_name\_array:

printf("参数名: ");

break;

case formal\_parameter\_type:

printf("参数类型: ");

break;

case formal\_parameter\_type\_array:

printf("参数类型: ");

break;

case complex\_statement:

printf("复合语句:\n");

break;

case statement:

printf("语句:\n");

break;

case expression:

printf("表达式:\n");

break;

case operation:

printf("操作符: ");

break;

case operator\_number:

printf("操作数: ");

break;

case IF\_ELSE\_node:

printf("IF - ELSE子句:\n");

break;

case IF\_node:

printf("IF子句:\n");

break;

case WHILE\_node:

printf("WHILE子句:\n");

break;

case FOR\_node:

printf("FOR子句:\n");

break;

case FOR\_node1:

printf("一个条件的FOR子句:\n");

break;

case FOR\_node2:

printf("两个条件的FOR子句:\n");

break;

case FOR\_node3:

printf("三个条件的FOR子句:\n");

break;

case FOR\_condition1:

printf("FOR的条件一:\n");

break;

case FOR\_condition2:

printf("FOR的条件二:\n");

break;

case FOR\_condition3:

printf("FOR的条件三:\n");

break;

case FOR\_no\_condition\_node:

printf("FOR子句沒有条件\n");

break;

case CONTINUE\_node:

printf("CONTINUE语句:\n");

break;

case BREAK\_node:

printf("BREAK语句:\n");

break;

case function\_name:

printf("函数名: ");

break;

case RETURN\_node:

printf("RETURN语句:\n");

break;

case RETURN\_no\_node:

printf("没有RETURN值\n");

break;

case array\_name:

printf("数组名: ");

break;

case array\_type:

printf("数组类型: ");

break;

case WHILE\_condition:

printf("WHILE条件:\n");

break;

case local\_variable\_define:

printf("局部变量定义:\n");

break;

case actual\_parameter\_sequence:

printf("实参序列: ");

break;

case no\_actual\_parameter\_sequence:

printf("实参序列: ");

break;

}

}

//前序遍历

void PreorderTranverse(ASTtree\* root, int depth) {

if (root == NULL) {

printf("");

}

else {

int iDepth = depth;

if (root->type) {

switch (root->type) {

case NodeType::no\_complex\_statement:

case NodeType::no\_formal\_parameter:

case NodeType::extern\_define\_sequence:

case NodeType::local\_variable\_define\_sequence:

case NodeType::statement\_sequence:

case NodeType::variable\_sequence:

case NodeType::formal\_parameter\_sequence:

case NodeType::Ignore:

break;

case NodeType::formal\_parameter:

case NodeType::complex\_statement:

for (int i = 0; i < iDepth - 1; i++) {

printf(" ");

}

break;

default:

for (int i = 0; i < iDepth; i++) {

printf(" ");

}

iDepth++;

break;

}

if (root->type == NodeType::actual\_parameter\_sequence) {

if (root->data.content) {

printType(root->type);

}

else {

printf("实参序列:\n");

}

}

else {

printType(root->type);

}

}

//打印出数组容量

if (root->data.content != NULL) {

printf("%s\n", root->data.content);

Vars\* p = varRoot->next;

switch (root->type) {

case NodeType::variable:

case NodeType::extern\_variable\_define:

case NodeType::array\_name:

while (p) {

if (p->var){

if (strcmp(root->data.content, p->var) == 0) {

if (p->size != 0) {

for (int i = 0; i < iDepth - 1; i++) {

printf(" ");

}

printf("数组容量: %d\n", p->size);

}

break;

}

}

p = p->next;

}

break;

}

}

PreorderTranverse(root->children, iDepth);

PreorderTranverse(root->sibling, iDepth);

}

}

int add\_vars(char \*varname, int cap, DataType kind) {

char\* cpyVarname = (char \*)malloc(sizeof(char)\*30);

if (cpyVarname) {

\*(cpyVarname) = '\0';

strcpy(cpyVarname, varname);

Vars\* p = (Vars\*)malloc(sizeof(Vars));

if (p) {

p->var = cpyVarname;

p->next = NULL;

p->size = cap;

p->datatype = kind;

varTail->next = p;

varTail = p;

return 0;

}

else {

return -1;

}

}

else {

return -1;

}

}

//检查是否重定义变量名

int check\_vars(char \*varname) {

if (countvarname >= 1) {

if (varRoot->next) {

Vars\* p = varRoot->next;

while (p) {

if (strcmp(p->var, varname) == 0) {

printf("\t文件中的第%d行出现错误\n", row\_count);

printf("\t错误:重定义变量名\n");

return -1;

}

p = p->next;

}

return 0;

}

else {

return -1;

}

}

else {

return 0;

}

}

//如果函数要返回值，但没有返回值

int had\_return(ASTtree \*root,DataType kind) {

if (root) {

if (root->type == NodeType::RETURN\_node) {

//检查是有返回值

countreturn++;

Vars\* p = varRoot->next;

while (p) {

if(root->children){

if(root->children->children->data.content)

if (strcmp(root->children->children->data.content, p->var) == 0) {

//检查是否有该返回的标识符

countvarreturn++;

if (p->datatype != kind) {

//检查返回值是否与函数一致

hadreturn++;

}

}

}

p = p->next;

}

}

if (root->children) {

had\_return(root->children, kind);

}

if (root->sibling) {

had\_return(root->sibling, kind);

}

}

if (hadreturn > 0) {

return -2;

}

else if ((countvarreturn == 0 && isreturn == 2)) {

return 0;

}

else if ((countvarreturn == 0 && isreturn == 3)||(countvarreturn >0 && isreturn == 1 && hadreturn > 0)) {

return -1;

}

else if (countvarreturn == 0) {

return -3;

}

else {

return 0;

}

}

1. outputFormat.h 头文件

#pragma once

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#include <string>

#include "syntaxAnalyse.h"

int outputFormat();

void PreorderTranverse2(ASTtree\* root, int depth);

1. outputFormat.cpp 源文件

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "outputFormat.h"

#include "syntaxAnalyse.h"

#include "gettoken.h"

extern FILE\* fp;

extern char token\_text[30];

extern int row\_count; //行数计数器

extern Vars\* varRoot;

extern char copy[100];

int countVar = 0;

int outputFormat() {

if ((fp = fopen("test3.txt", "r")) == NULL) {

printf("\t 打开文件失败!\n");

return -1;

}

else {

row\_count = 0;

printf("\t 打开文件成功!\n");

}

ASTtree\*root = program();

if (!root) {

printf("\t语法树有错\n");

exit(-1);

}

printf("%s", copy);

PreorderTranverse2(root, 0);

return 0;

}

void PreorderTranverse2(ASTtree\* root, int depth) {

if (root) {

char\* content = (char\*)malloc(sizeof(char) \* 50);

\*content = '\0';

if (root->data.content && root->type != function\_name) strcat(content, root->data.content);

Vars\* p = varRoot->next;

int idepth = 0;

switch (root->type) {

case Ignore:

case extern\_define: //外部定义

case extern\_define\_sequence: //外部定义序列

break;

case extern\_variable\_define: //外部变量定义

case local\_variable\_define: //局部变量定义

idepth = depth;

case formal\_parameter\_array:

case formal\_parameter: //形参

case function\_define: //函数定义

strcat(content, " ");

break;

case variable\_sequence: //变量序列

if(root->sibling) strcat(content, ", ");

else strcat(content, ";\n");

break;

case return\_type: //返回值类型

break;

case formal\_parameter\_sequence: //形参序列

if (root->sibling->type != NodeType::no\_formal\_parameter) {

strcat(content, ", ");

}

break;

case formal\_parameter\_type\_array:

case formal\_parameter\_type: //参数类型

case formal\_parameter\_name:

case complex\_statement: //复合语句

break;

case IF\_content:

strcat(content, ";\n");

for (int i = 0; i < depth - 1; i++) {

strcat(content, "\t");

}

strcat(content, "}\n");

idepth = depth;

break;

case statement: //語句

if (root->children) {

if (root->children->type != IF\_ELSE\_node && root->children->type != IF\_node && root->children->type != WHILE\_node && root->children->type != FOR\_node) {

strcat(content, ";\n");

}

}

idepth = depth;

break;

case statement\_sequence: //语句序列

case expression: //表达式

case operation: //操作符

case expression\_ident: //表达式中的操作符

case operator\_number: //操作数

break;

case IF\_ELSE\_node: //IF - ELSE结点

case IF\_node: //IF结点

for (int i = 0; i < depth; i++) {

printf("\t");

}

printf("if (");

strcat(content, "){\n");

break;

case ELSE\_node:

for (int i = 0; i < depth-1; i++) {

printf("\t");

}

printf("else {\n");

strcat(content, ";\n");

idepth = depth;

break;

case WHILE\_node: //WHILE结点

for (int i = 0; i < depth; i++) {

printf("\t");

}

printf("while ");

break;

case WHILE\_condition:

printf("(");

strcat(content, ") {\n");

break;

case RETURN\_no\_node:

case SWITCH\_node: //SWITCH结点

break;

case FOR\_node:

for (int i = 0; i < depth; i++) {

printf("\t");

}

printf("for (");

break;

case FOR\_condition1:

strcat(content, ";");

break;

case FOR\_condition2:

strcat(content, ";");

break;

case FOR\_condition3:

strcat(content, ") {\n");

break;

case FOR\_node1:

case FOR\_node2:

case FOR\_node3:

case FOR\_no\_condition\_node:

break;

case formal\_parameter\_name\_array:

case array\_name: //数组名

while (p) {

if (p->var) {

if (!strcmp(root->data.content, p->var)) {

if (p->size != 0) {

sprintf(content, "[%d]", p->size);

}

break;

}

}

p = p->next;

}

break;

case array\_type: //数组类型

break;

case no\_actual\_parameter\_sequence:

strcat(content, "()");

break;

case actual\_parameter\_sequence: //實參序列

if (root->data.content) {

printf("%s", root->data.content);

printf("(");

}

content[0] = '\0';

if (root->sibling) strcat(content, ", ");

else strcat(content, ")");

break;

case variable: //变量

break;

case function\_name: //函数名称

printf("%s", root->data.content);

printf("(");

break;

case no\_formal\_parameter: //函數空參數時

printf("){\n");

break;

case variable\_type: //变量類型

break;

case CONTINUE\_node:

for (int i = 0; i < depth; i++) {

printf("\t");

}

printf("continue;\n");

break;

case BREAK\_node:

for (int i = 0; i < depth; i++) {

printf("\t");

}

printf("break;\n");

break;

case RETURN\_node: //RETURN结点

for (int i = 0; i < depth; i++) {

printf("\t");

}

printf("return ");

strcat(content, ";\n");

break;

}

if (root->children) {

for (int i = 0; i < idepth; i++) {

printf("\t");

}

//遍历左孩子

PreorderTranverse2(root->children, depth);

}

printf("%s", content);

content[0] = '\0';

idepth = 0;

if (root->sibling) {

switch (root->sibling->type) {

case Ignore:

case extern\_define: //外部定义

case extern\_define\_sequence: //外部定义序列

case statement\_sequence: //语句序列

case extern\_variable\_define: //外部变量定义

case variable\_sequence: //变量序列

case return\_type: //返回值类型

case formal\_parameter\_sequence: //形参序列

case formal\_parameter\_type\_array:

case formal\_parameter\_type: //参数类型

case formal\_parameter: //形参

case formal\_parameter\_array:

case formal\_parameter\_name\_array:

case formal\_parameter\_name:

break;

case no\_complex\_statement:

case IF\_ELSE\_node: //IF - ELSE结点

case IF\_node: //IF结点

case IF\_content:

case WHILE\_node: //WHILE结点

case complex\_statement: //复合语句

depth++;

break;

case ELSE\_node:

case local\_variable\_define: //局部变量定义

case statement: //語句

case expression: //表达式

case operation: //操作符

case expression\_ident: //表达式中的操作符

case operator\_number: //操作数

case WHILE\_condition:

case RETURN\_no\_node:

case SWITCH\_node: //SWITCH结点

case FOR\_node:

case FOR\_node1:

case FOR\_node2:

case FOR\_node3:

case FOR\_condition1:

case FOR\_condition2:

case FOR\_condition3:

case FOR\_no\_condition\_node:

case CONTINUE\_node:

case BREAK\_node:

case array\_name: //数组名

case array\_type: //数组类型

case actual\_parameter\_sequence: //實參序列

case function\_define: //函数定义

case variable: //变量

case function\_name: //函数名称

case variable\_type: //变量類型

case RETURN\_node: //RETURN结点

break;

}

//遍历右孩子

PreorderTranverse2(root->sibling, depth);

}

if (root) {

switch (root->type) {

case Ignore:

case extern\_define: //外部定义

case extern\_define\_sequence: //外部定义序列

case statement\_sequence: //语句序列

case extern\_variable\_define: //外部变量定义

case variable\_sequence: //变量序列

case return\_type: //返回值类型

case formal\_parameter\_sequence: //形参序列

case formal\_parameter\_type\_array:

case formal\_parameter\_type: //参数类型

case formal\_parameter\_array:

case formal\_parameter: //形参

case formal\_parameter\_name\_array:

case formal\_parameter\_name:

case IF\_ELSE\_node: //IF - ELSE结点

break;

case IF\_node: //IF结点

case IF\_content:

case complex\_statement: //复合语句

strcat(content, "}\n");

idepth = depth;

break;

case no\_complex\_statement:

printf("\n");

strcat(content, "}\n");

idepth = depth;

break;

case WHILE\_node: //WHILE结点

case ELSE\_node:

case local\_variable\_define: //局部变量定义

case statement: //語句

case expression: //表达式

case operation: //操作符

case expression\_ident: //表达式中的操作符

case operator\_number: //操作数

case WHILE\_condition:

case RETURN\_no\_node:

case SWITCH\_node: //SWITCH结点

case FOR\_node:

case FOR\_node3:

case FOR\_condition1:

case FOR\_condition2:

case FOR\_condition3:

case CONTINUE\_node:

case BREAK\_node:

case array\_name: //数组名

case array\_type: //数组类型

case actual\_parameter\_sequence: //實參序列

case function\_define: //函数定义

case variable: //变量

case function\_name: //函数名称

case variable\_type: //变量類型

case RETURN\_node: //RETURN结点

break;

case FOR\_no\_condition\_node:

strcat(content, ";;) {\n");

break;

case FOR\_node1:

strcat(content, ";) {\n");

break;

case FOR\_node2:

strcat(content, ") {\n");

break;

}

for (int i = 0; i < idepth-1; i++) {

printf("\t");

}

printf("%s", content);

}

free(content);

}

}