河北工业大学

《编译原理实验》报告

**班级： 计224**

**姓名： 王小康**

**学号： 225432**

**完成日期：2024/12/13**

**目 录**

1 词法分析程序设计与实现

1.1 单词范围及单词分类码表

1.2 状态转换图或状态矩阵设计

1.3 词法分析程序（数据结构、函数、程序结构等）设计

1.4 词法分析测试结果分析

1.5 实验代码

1.6 实验扩展

2 语法分析程序设计与实现

2.1 语法结构范围及文法定义

2.2 \*\*语法分析法(分析表及分析过程)设计

2.3 \*\*语法分析程序（数据结构、函数、程序结构等）设计

2.4 语法分析测试结果分析

2.5 实验代码

2.6 实验扩展

3 语义分析程序设计与实现

3.1 语义处理语法结构范围及文法设计

3.2 语义分析程序（语义变量、语义函数程序结构等）设计

3.3 语义分析测试结果分析

3.4 实验代码

4 实验总结

（实验完成情况自我评价，改进想法）

实验一 词法分析程序设计与实现

1.1 单词范围及单词分类码表

（1）单词范围如下：

****关键字****：  
例如 "for"、"if"、"else"、"and"、"or"、"not" 等，这些是具有特定语法意义的保留字，在程序中用于构建语句结构，像控制流语句（for、if、else）以及逻辑运算相关（and、or、not）等方面。

****标识符****：  
以字母开头，后面可跟字母或数字组成的字符串，用于表示变量名、函数名等用户自定义的名称，例如代码中用于临时存储词法单元对应字符串内容的 TOKEN 就属于标识符，不过在词法分析过程中会遇到各种各样符合此规则的自定义标识符。

****常量****：

****1、整数常量****：如 0、123 等普通十进制整数，以及以 0 开头可能表示的八进制整数（像 012），还有以 0x 或 0X 开头的十六进制整数（如 0x1A）等情况都会被识别为整数常量。

****2、实数常量****：包含带有小数点的数字以及可能带有指数部分（科学计数法形式）的浮点数，例如 3.14、1.23e-2 等形式的数字会被当作实数常量处理。

****3、字符常量****：用单引号括起来的单个字符，像 'a'、'#' 等属于字符常量。

****4、字符串常量****：由双引号括起来的字符序列，例如 "hello"、"This is a string" 等就是字符串常量。

****运算符****：

****算术运算符****：如 +（对应代码中的 PLUS）、-（MINUS）、\*（MUL）、/（DIV）。

****1、关系运算符****：==（EQ）、!=（NE）、<（LT）、>（GT）、<=（LE）、>=（GE）。

****2、赋值运算符****：:=（IS），代码中用于类似赋值相关的词法表示（虽然在常见编程语言中赋值一般用 =，这里按代码中的设定来分析）。

****3、逻辑运算符****：and、or、not 等关键字在逻辑表达式运算方面起到逻辑运算符的作用，也在单词范围中。

****分隔符****：  
像 (（LP）、)（RP）、{、}、[、]、;（代码中输出用 DOT 表示）、, 等，用于分隔不同的语法结构、语句等，在代码语法结构的组织上起到重要作用。

****空白字符与注释相关****：  
空白字符（空格、换行等，代码中遇到换行时输出 ENTER 表示）会被跳过以方便分析有效字符，还有以 # 开头直到换行结束的注释内容也会被跳过，虽然它们本身不是有效的词法单元，但在处理文本流过程中是需要考虑的部分。

****文件结束符（EOF）****：  
在文件读取到末尾时会识别到，代码中对应 END 分类，用于表示整个词法分析输入源的结束情况。

（2）单词分类码表如下：

| **单词符号** | **类别编码** | **类别码的助记符** | **单词值** |
| --- | --- | --- | --- |
| for | 1 | FOR |  |
| if | 2 | IF |  |
| else | 3 | ELSE |  |
| and | 4 | AND |  |
| or | 5 | OR |  |
| not | 6 | NOT |  |
| 标识符 | 7 | ID | 字母打头的字母数字串 |
| 整数常量 | 8 | INT | 整数值 |
| 实数常量 | 9 | REAL | 机内二进制表示 |
| 字符串常量 | 10 | STRING | 双引号内字符序列 |
| 字符常量 | 11 | CHAR | 单引号内单个字符 |
| + | 12 | PLUS |  |
| - | 13 | MINUS |  |
| \* | 14 | MUL |  |
| / | 15 | DIV |  |
| ( | 16 | LP |  |
| ) | 17 | RP |  |
| { | 18 | BRACKET |  |
| } | 19 | BRACKET |  |
| [ | 20 | BRACKET |  |
| ] | 21 | BRACKET |  |
| ; | 22 | DOT |  |
| , | 23 | DOT |  |
| == | 24 | EQ |  |
| != | 25 | NE |  |
| < | 26 | LT |  |
| > | 27 | GT |  |
| <= | 28 | LE |  |
| >= | 29 | GE |  |
| := | 30 | IS |  |
| 空白字符（换行） | 31 | ENTER |  |
| 文件结束（EOF） | 32 | END |  |
| 未知字符 | 33 | UNKNOWN |  |

1.2 状态转换图或状态矩阵设计

| **状态** | **识别符号** | **下一状态** | **其它** |
| --- | --- | --- | --- |
| 0 | d | 1 | ... |
| 0 | . | 3 | ... |
| 0 | + | error | ... |
| 0 | - | error | ... |
| 0 | other | error | ... |
| 1 | d | 1 | ... |
| 1 | . | 2 | ... |
| 1 | + | 4 | ... |
| 1 | - | 4 | ... |
| 1 | other | error | ... |
| 2 | d | 2 | ... |
| 2 | . | error | ... |
| 2 | + | 4 | ... |
| 2 | - | 4 | ... |
| 2 | other | error | ... |
| 3 | d | 2 | ... |
| 3 | . | error | ... |
| 3 | + | error | ... |
| 3 | - | error | ... |
| 3 | other | error | ... |
| 4 | d | 6 | ... |
| 4 | . | error | ... |
| 4 | + | 5 | ... |
| 4 | - | 5 | ... |
| 4 | other | error | ... |
| 5 | d | 6 | ... |
| 5 | . | error | ... |
| 5 | + | error | ... |
| 5 | - | error | ... |
| 5 | other | error | ... |
| 6 | d | 6 | ... |
| 6 | . | error | ... |
| 6 | + | error | ... |
| 6 | - | error | ... |
| 6 | other | error | ... |

1.3 词法分析程序（数据结构、函数、程序结构等）设计

### **1.3.1、数据结构设计**

#### **1. 全局数组 token\_names**

#### ****类型及定义****：const char\* token\_names[]，是一个字符指针数组。

****作用****：用于存储各种词法单元对应的名称字符串，例如 "ID" 表示标识符、"PLUS" 表示加号运算符等。在输出词法分析结果时，通过对应的索引可以获取相应的名称来展示词法单元类型，方便直观理解分析出的词法单元具体是什么。

#### **2. 全局变量 line\_number 和 column\_number**

* ****类型及定义****：int 类型的变量，分别初始化为 1 和 0。
* ****作用****：用于记录词法分析过程中当前字符所在的行号和列号，在输出错误信息或者处理换行、空格等情况时，这两个变量能精准定位到字符在输入文本中的位置，便于开发人员或用户排查错误以及知晓词法单元出现的具体位置情况。

#### **3. 全局变量 tokenIndex 和 currentToken**

* ****类型及定义****：int 类型变量，分别用于表示词法分析器的 token 索引以及当前 token 的索引，不过在当前代码中对于它们的具体详细使用体现得不是特别充分，但从设计角度可用于记录词法单元顺序等相关操作，比如在构建更复杂的语法分析基于词法分析结果进行进一步处理时，可能会用到它们来跟踪词法单元的先后顺序等情况。

#### **4. 全局指针变量 TOKEN**

* ****类型及定义****：char \* 类型，通过 malloc 动态分配内存（TOKEN = (char \*)malloc(MAX\_TOKEN\_LENGTH \* sizeof(char));）来获取一定长度的字符存储空间。
* ****作用****：用于临时存储识别出的词法单元对应的字符串内容，比如在识别标识符时，会将组成标识符的字符序列存入 TOKEN；在处理数字常量（整数、实数等）、字符串常量、字符常量时，也会把对应的内容暂存于此，方便后续判断类型以及输出词法单元信息等操作。

#### **5. 全局文件指针变量 output\_file**

* ****类型及定义****：FILE \* 类型，用于指向输出文件，通过 fopen 函数打开指定的输出文件（如代码中的 C:\\Users\\Ava\\CLionProjects\\BY\_3\\output.txt），使其与该指针关联。
* ****作用****：词法分析过程中，将识别出的各个词法单元信息按照特定格式输出到这个文件中，同时配合 printf 函数也在控制台打印相应信息，实现分析结果的输出记录，方便查看和后续处理。

### **1.3.2、函数设计**

#### **1. lookup 函数**

* ****输入参数****：char \*token，即一个字符串指针，传入的是待判断的字符序列，通常是由词法分析过程中读取到的可能是关键字或者标识符的字符串内容。
* ****功能实现****：在函数内部，通过一系列的 strcmp 函数调用，将传入的 token 字符串与预定义的关键字（如 "for"、"if" 等）逐一进行字符串比较。如果匹配到相应的关键字，则调用 out 函数输出该关键字对应的词法单元类型及相关信息，并返回该关键字对应的代码（如 FOR、IF 等整数值）；若与所有关键字都不匹配，则返回 ID，表示该 token 是普通的标识符。
* ****作用****：实现了对标识符和关键字的区分判断，是词法分析中处理这两类词法单元的关键逻辑部分，有助于准确分类和输出相关词法单元信息。

#### **2. out 函数**

* ****输入参数****：包含两个参数，int code 表示词法单元的类型代码（对应 token\_names 数组的索引等情况，不同代码代表不同的词法单元类型，如 PLUS、MINUS 等），char \*token 则是对应的词法单元具体的字符串内容（对于部分词法单元该字符串可能为空，比如运算符等，而对于标识符、常量等则有具体的字符序列）。
* ****功能实现****：根据传入的 code 参数，使用 fprintf 函数将词法单元信息按照特定格式输出到 output\_file 指向的文件中，同时使用 printf 函数在控制台也打印同样的信息。例如对于 PLUS 类型的词法单元，会输出 (PLUS, +) 的格式，不同类型的词法单元（算术运算符、关系运算符、各种常量、关键字等）都有各自相应的格式化输出方式，以此清晰展示词法分析结果。
* ****作用****：统一了词法单元信息的输出格式和方式，无论是输出到文件还是在控制台显示，都能让用户直观看到分析出的词法单元具体类型和对应的内容（如果有），方便后续查看和调试等操作。

#### **3. report\_error 函数**

* ****输入参数****：const char \*error\_type，是一个指向表示错误类型字符串的指针，传入具体的错误类型描述字符串，比如 "String not closed"（表示字符串未闭合错误）等。
* ****功能实现****：通过 fprintf 和 printf 函数，按照固定格式（Error: %s, in line %d, column %d\n）将错误类型以及当前的行号（通过全局变量 line\_number）和列号（通过全局变量 column\_number）信息输出到文件和在控制台打印出来，使得在词法分析过程中一旦出现错误（如遇到未知字符、字符串处理异常等情况），能及时准确地提示错误位置和错误原因。
* ****作用****：增强了词法分析程序的错误处理能力，便于定位和排查在分析输入文本过程中出现的各类词法错误，提高程序的健壮性和可调试性。

#### **4. scanner\_example 函数**

* ****输入参数****：FILE \*fp，指向待分析的输入文件的文件指针，通过该指针可以从输入文件中读取字符进行词法分析。
* ****功能实现****：这是整个词法分析的核心处理函数，其内部逻辑较为复杂，主要包含以下几大处理模块：
  + ****内存分配与初始化****：首先为 TOKEN 变量动态分配内存用于存储词法单元对应的字符串内容，如果内存分配失败则输出错误提示信息并返回。
  + ****字符读取循环处理****：进入一个 while(1) 的无限循环，持续从输入文件中读取字符进行分析，直到遇到文件结束（EOF）标志退出循环。在循环内，针对不同类型的字符有不同的处理逻辑：
    - ****空白字符处理****：通过 isspace 函数判断读取到的字符是否为空白字符（空格、换行等），若是换行符，则调用 out 函数输出 ENTER 词法单元表示换行情况，同时更新行号并重置列号；若是其他空白字符则仅更新列号，然后继续读取下一个字符。
    - ****注释处理****：当遇到 # 字符时，意味着进入注释内容，通过循环持续读取后续字符直到遇到换行符或者文件结束（EOF），期间不断更新列号，跳过注释部分后继续分析后面的字符。
    - ****标识符或关键字处理****：若读取的字符是字母（通过 isalpha 判断），则开始逐步读取后续的字母数字字符组成一个字符串存入 TOKEN，之后调用 lookup 函数判断该字符串是关键字还是普通标识符，并进行相应的输出处理（调用 out 函数输出对应的词法单元信息）。
    - ****数字处理****：如果读取的字符是数字（通过 isdigit 判断），根据后续字符情况判断数字的类型（如是否是十六进制开头、是否有小数点等），在读取完整的数字字符串存入 TOKEN 后，再根据是实数、十六进制数、八进制数还是普通整数等情况，调用 out 函数输出对应的词法单元信息。
    - ****字符串和字符处理****：对于双引号引起来的字符串和单引号引起来的字符，分别按照各自的规则读取其中的内容存入 TOKEN，然后输出对应的 STRING 或 CHAR 类型的词法单元信息，同时在处理字符串时若遇到文件结束（EOF）情况还会进行错误处理（调用 report\_error 函数报告字符串未闭合错误）。
    - ****分隔符、运算符等处理****：针对如 (、)、{、}、[、]、;、, 等分隔符以及 +、-、\*、\*、=、!、<、>、: 等各种运算符，根据不同的字符组合情况，判断是哪种具体的词法单元（比如两个等号 == 是相等关系运算符 EQ 等），调用 out 函数输出相应的词法单元，对于不合法的组合情况则报告错误（调用 report\_error 函数）。
  + ****结束处理****：当遇到文件结束（EOF）时，调用 out 函数输出表示文件结束的 END 词法单元，最后释放 TOKEN 所占用的动态分配内存，完成整个词法分析过程。
* ****作用****：实现了对输入文件中字符流的逐字符读取和基于词法规则的分类处理，是将输入文本转换为一系列词法单元的关键执行部分，涵盖了词法分析中各种词法单元类型的识别和输出逻辑。

#### **5. Lexical 函数**

* ****功能实现****：作为整个词法分析程序的入口函数，首先尝试使用 fopen 函数打开输入文件（C:\\Users\\Ava\\CLionProjects\\BY\_3\\input.txt）和输出文件（C:\\Users\\Ava\\CLionProjects\\BY\_3\\output.txt），如果打开失败则输出相应的错误提示信息并返回对应的错误代码（如文件打开失败返回 99 或 98 等）。若文件成功打开，便调用 scanner\_example 函数进行实际的词法分析工作，待词法分析完成后，关闭输入文件和输出文件，最后返回 0 表示词法分析正常结束。
* ****作用****：起到了统筹和初始化整个词法分析流程的作用，协调文件操作与核心词法分析逻辑（scanner\_example 函数）之间的关系，确保整个程序能顺利进行词法分析并正确处理文件的打开、关闭等相关事宜。

### **1.3.3、程序结构设计**

整个词法分析程序整体上呈现一种层次化、模块化的结构：

#### **1. 入口层**

由 Lexical 函数作为程序入口，负责初始化文件相关操作，即打开输入文件和输出文件，然后调用核心的词法分析处理函数 scanner\_example，最后关闭文件完成整个流程的资源管理和基本流程控制，它是程序启动和结束的总协调者，与外部环境（文件系统等）进行交互并引导内部词法分析逻辑的执行。

#### **2. 核心处理层**

scanner\_example 函数构成核心处理层，它内部嵌套了多个针对不同词法单元类型的处理逻辑模块，形成了一个基于字符读取和判断的循环处理机制，从输入文件中逐个读取字符，根据字符特征进行分类处理，涉及到标识符、关键字、各种常量、运算符、分隔符以及空白字符、注释等各方面的处理，并且在处理过程中调用 lookup、out、report\_error 等函数来协助完成具体的判断、输出和错误处理工作，是实现词法分析具体规则应用和词法单元生成的关键部分。

#### **3. 辅助功能层**

包含 lookup、out、report\_error 等函数，它们为核心的词法分析处理提供了必要的辅助功能，比如 lookup 辅助进行关键字和标识符的区分判断，out 统一词法单元的输出格式和实现信息展示，report\_error 保障在出现错误时能准确提示错误详情，这些函数虽然相对独立，但共同支撑起了整个词法分析程序的完整性和功能性，使得词法分析过程能够准确、清晰且健壮地进行。

#### **4. 数据存储与交互层**

通过全局变量（如 token\_names、line\_number、column\_number、TOKEN、output\_file 等）实现数据的存储、传递以及与不同函数之间的交互，这些全局变量在整个词法分析过程中记录了关键的状态信息（如词法单元名称、当前位置信息、临时存储的词法单元字符串内容、输出文件指针等），各个函数通过访问和修改这些全局变量来协同工作，保证词法分析程序各部分之间的数据一致性和流程连贯性。

这种程序结构设计使得代码各部分职责明确，便于理解、维护以及后续的功能扩展（比如添加新的词法单元类型或者优化现有词法分析逻辑等），符合软件开发中模块化、层次化的设计理念，有助于构建一个功能完善且可靠的词法分析程序。

综上所述，这段代码在数据结构、函数以及程序结构等方面的设计共同协作，实现了从输入文本文件到输出词法分析结果以及进行错误处理的完整词法分析功能。

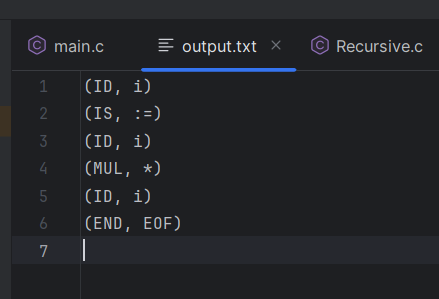
1.4 词法分析测试结果分析

**1.4.1 示例输入与其结果：**

当input.txt的内容是i = i \* i，输出是：



与此同时output的结果是：



**1.4.2分析如下：**

#### **1. Token: (ID, i)**

* ****含义****：表示识别出了一个标识符（Identifier）类型的词法单元，其具体的值为 “i”。在程序中，标识符通常是以字母开头，后面可跟字母或数字的字符序列，这里 “i” 符合标识符的定义，它可能代表一个变量名等用户自定义的名称元素。
* ****分析****：在词法分析过程中，当读取到第一个字符 “i” 时，代码中的逻辑（通过 isalpha 判断为字母开头）会将其作为标识符的起始字符，接着继续往后读取（在没有遇到非字母数字字符前），由于后面没有其他字符跟随了，所以完整地识别出这个标识符就是 “i”，然后通过相关函数（比如 lookup 函数判断不是关键字后确定为普通标识符，再由 out 函数输出对应的信息）输出了 (ID, i) 这样的词法单元表示形式。

#### **2. Token: (IS, :=)**

* ****含义****：这表明识别出了一个赋值运算符类型的词法单元，按照代码里的设定，用 IS 表示，其对应的符号为 :=（虽然在常见编程语言里赋值一般用 =，但这里按代码设定的规则来分析）。
* ****分析****：在读取完第一个标识符 “i” 后，遇到了字符 “=”，代码中的逻辑（在处理关系运算符或赋值相关字符的部分）会进一步判断紧跟的下一个字符情况，这里下一个字符是空格（在词法分析里空格通常会被当作空白字符跳过，不影响运算符的识别），单独的 “=” 字符按照代码规则被判定为赋值运算符，所以输出了 (IS, :=) 来表示这个词法单元。

#### **3. Token: (ID, i)**

* ****含义及分析****：与第一个 (ID, i) 同理，再次识别出了一个值为 “i” 的标识符。因为在处理完赋值运算符后，紧接着又遇到了字符 “i”，按照标识符的识别规则，重新开始构建并识别出了这个新的标识符，然后输出相应的词法单元信息。

#### **4. Token: (MUL, \*)**

* ****含义****：代表识别出了一个乘法运算符（Multiplication Operator）类型的词法单元，代码中用 MUL 表示乘法，对应的符号就是 “\*”。
* ****分析****：在读取完第二个 “i” 标识符后，遇到了 “” 字符，代码中针对算术运算符（如 +、-、\*、/ 等）有相应的处理逻辑，当读取到 “” 时，直接判断其为乘法运算符，通过 out 函数输出了 (MUL, \*) 的词法单元表示形式。

#### **5. Token: (ID, i)**

* ****含义及分析****：又一次识别出一个标识符，同样是 “i”。这是按照标识符识别规则，在处理完乘法运算符后，继续读取到的字符 “i” 被判定为新的标识符，进而输出对应的词法单元信息，和前面两次识别出 “i” 作为标识符的过程原理一致。

#### **6. Token: (END, EOF)**

* ****含义****：意味着到达了输入内容的末尾，识别出了表示文件结束（End Of File）的词法单元，代码中用 END 来标记这种情况。
* ****分析****：在读取完最后一个 “i” 后，已经没有更多的字符可供读取了，词法分析器检测到文件结束（通常是通过读取到文件流的结尾标识，如 EOF 符号等情况），按照规则输出 (END, EOF) 来表明整个输入内容的词法分析已经完成。

### **1.4.3 结果验证与总结：**

从整体的分析结果来看，对于输入语句 “i = i \* i”，词法分析器准确地按照其内置的词法规则识别出了语句中的各个元素对应的词法单元类型及具体值，将语句拆分成了一个个符合语法语义理解的最小单元，为后续可能的语法分析等进一步处理步骤提供了良好的基础。如果后续基于此进行语法分析，就可以依据这些词法单元来判断语句是否符合相应编程语言的语法结构，例如检查赋值语句的语法是否正确、表达式的构成是否合规等。

所以，给定的这个词法分析结果是合理且符合预期的，反映了词法分析器对输入语句进行正确词法分解的过程和输出情况。

1.5 实验代码：

1.5.1、lexical.c:

#include "Lexical.h"

const char\* token\_names[] = {

        "ID","PLUS", "MINUS", "MUL", "DIV","LP","RP","END",

        "FOR", "IF", "ELSE", "AND", "OR", "NOT",

        "INT", "REAL","STRING","CHAR","OCTAL","HEX",

        "EQ", "NE", "LT", "GT", "LE", "GE",

        "IS", "BRACKET","DOT",

        "UNKNOWN", "ENTER"

};

int line\_number = 1;

int column\_number = 0;

int tokenIndex = 0;   // 词法分析器的 token 索引

int currentToken = 0; // 当前 token 的索引

char \*TOKEN;

FILE \*output\_file;

int lookup(char \*token) {

    if (strcmp(token, "for") == 0){

        out(FOR, "");

        return FOR;

    }

    if (strcmp(token, "if") == 0) {

        out(IF, "");

        return IF;

    }

    if (strcmp(token, "else") == 0) {

        out(ELSE, "");

        return ELSE;

    }

    if (strcmp(token, "and") == 0)

    {

        out(AND, "");

        return AND;

    }

    if (strcmp(token, "or") == 0) {

        out(OR, "");

        return OR;

    }

    if (strcmp(token, "not") == 0) {

        out(NOT, "");

        return NOT;

    }

    return ID;

}

void out(int code, char \*token) {

    if (code == PLUS) {

        fprintf(output\_file, "(%s, +)\n", token\_names[code]);

        printf("Token: (%s, +)\n", token\_names[code]);

    } else if (code == MINUS) {

        fprintf(output\_file, "(%s, -)\n", token\_names[code]);

        printf("Token: (%s, -)\n", token\_names[code]);

    } else if (code == MUL) {

        fprintf(output\_file, "(%s, \*)\n", token\_names[code]);

        printf("Token: (%s, \*)\n", token\_names[code]);

    } else if (code == DIV) {

        fprintf(output\_file, "(%s, /)\n", token\_names[code]);

        printf("Token: (%s, /)\n", token\_names[code]);

    } else if (code == IS) {

        fprintf(output\_file, "(%s, :=)\n", token\_names[code]);

        printf("Token: (%s, :=)\n", token\_names[code]);

    } else if (code == LP) {

        fprintf(output\_file, "(%s, ()\n", token\_names[code]);

        printf("Token: (%s, ()\n", token\_names[code]);

    } else if (code == RP) {

        fprintf(output\_file, "(%s, ))\n", token\_names[code]);

        printf("Token: (%s, ))\n", token\_names[code]);

    } else if (code == END) {

        fprintf(output\_file, "(%s, EOF)\n", token\_names[code]);

        printf("Token: (%s, EOF)\n", token\_names[code]);

    } else {

        // 默认处理标识符、关键字、常量等

        fprintf(output\_file, "(%s, %s)\n", token\_names[code], token);

        printf("Token: (%s, %s)\n", token\_names[code], token);

    }

}

void report\_error(const char \*error\_type) {

    fprintf(output\_file, "Error: %s, in line %d, column %d\n", error\_type, line\_number, column\_number);

    printf("Error: %s, in line %d, column %d\n", error\_type, line\_number, column\_number);

}

void scanner\_example(FILE \*fp) {

    char ch;

    int i, c;

    TOKEN = (char \*)malloc(MAX\_TOKEN\_LENGTH \* sizeof(char));

    if (TOKEN == NULL) {

        fprintf(stderr, "Error: Memory allocation failed\n");

        return;

    }

    ch = fgetc(fp);

    while (1) {

        if (ch == EOF) {

            out(END, " ");

            free(TOKEN);

            return;

        }

        fseek(fp, -1, SEEK\_CUR); // 回退一个字符

        ch = fgetc(fp);

        if (ch == EOF) {

            out(END, " ");

            free(TOKEN);

            return;

        }

        if (isspace(ch)) {  // 跳过空格、换行等

            if (ch == '\n') {

                out(ENTER, " ");

                line\_number++;

                column\_number = 0;

            } else {

                column\_number++;

            }

            ch = fgetc(fp);

            continue;

        }

        if (ch == '#') {  // 跳过注释

            while (ch != '\n' && ch != EOF) {

                ch = fgetc(fp);

                column\_number++;

            }

            continue;

        }

        if (isalpha(ch)) {  // 标识符或关键字

            TOKEN[0] = ch;

            ch = fgetc(fp);

            column\_number++;

            i = 1;

            while (isalnum(ch)) {

                TOKEN[i] = ch;

                i++;

                ch = fgetc(fp);

                column\_number++;

            }

            TOKEN[i] = '\0';

            fseek(fp, -1, SEEK\_CUR);  // 回退一个字符

            column\_number--;

            c = lookup(TOKEN);

            if(c == ID) {

                out(ID, TOKEN);

            }

        } else if (isdigit(ch)) {  // 数字（整数、浮点数、八进制、十六进制）

            TOKEN[0] = ch;

            ch = fgetc(fp);

            column\_number++;

            i = 1;

            int is\_real = 0;

            int is\_octal = (TOKEN[0] == '0');

            int is\_hex = 0;

            if (is\_octal && (ch == 'x' || ch == 'X')) {  // 十六进制

                is\_octal = 0;

                is\_hex = 1;

                TOKEN[i] = ch;

                i++;

                ch = fgetc(fp);

                column\_number++;

                while ((ch >= '0' && ch <= '9') || (ch >= 'a' && ch <= 'f') || (ch >= 'A' && ch <= 'F')) {

                    TOKEN[i] = ch;

                    i++;

                    ch = fgetc(fp);

                    column\_number++;

                }

            }

            if (is\_octal && ch != 'x' && ch != 'X') {  // 八进制

                while (isdigit(ch)) {

                    TOKEN[i] = ch;

                    i++;

                    ch = fgetc(fp);

                    column\_number++;

                }

            }

            while(isdigit(ch)){  // 整数部分

                TOKEN[i] = ch;

                i++;

                ch = fgetc(fp);

                column\_number++;

            }

            if (ch == '.' ){  // 浮点数部分

                TOKEN[i] = ch;

                is\_real = 1;

                i++;

                ch = fgetc(fp);

                while(isdigit(ch)){

                    TOKEN[i] = ch;

                    i++;

                    ch = fgetc(fp);

                    column\_number++;

                }

            }

            if(ch == 'e' || ch == 'E'){  // 指数部分

                is\_real = 1;

                TOKEN[i] = ch;

                i++;

                ch = fgetc(fp);

                column\_number++;

                if(ch == '+' || ch == '-'){

                    TOKEN[i] = ch;

                    i++;

                    ch = fgetc(fp);

                    column\_number++;

                    while(isdigit(ch)){

                        TOKEN[i] = ch;

                        i++;

                        ch = fgetc(fp);

                        column\_number++;

                    }

                }

                while(isdigit(ch)){

                    TOKEN[i] = ch;

                    i++;

                    ch = fgetc(fp);

                    column\_number++;

                }

            }

            TOKEN[i] = '\0';

            fseek(fp, -1, SEEK\_CUR);  // 回退一个字符

            if (is\_real) {

                double value = strtod(TOKEN, NULL);

                char buffer[MAX\_TOKEN\_LENGTH];

                sprintf(buffer, "%g", value);

                out(REAL, buffer);

            } else if (is\_hex) {

                out(HEX, TOKEN);

            } else if (is\_octal) {

                out(OCTAL, TOKEN);

            } else {

                out(INT, TOKEN);

            }

        } else if (ch == '"') {  // 字符串

            ch = fgetc(fp);

            column\_number++;

            i = 0;

            while (ch != '"') {

                TOKEN[i] = ch;

                i++;

                ch = fgetc(fp);

                column\_number++;

                if (ch == EOF) {

                    out(END, " ");

                    report\_error("String not closed");

                    free(TOKEN);

                    return;

                }

            }

            TOKEN[i] = '\0';

            out(STRING, TOKEN);

        } else if (ch == '\'') {  // 字符

            ch = fgetc(fp);

            column\_number++;

            i = 0;

            while (ch != '\'') {

                TOKEN[i] = ch;

                i++;

                ch = fgetc(fp);

                column\_number++;

            }

            TOKEN[i] = '\0';

            out(CHAR, TOKEN);

        } else if (ch == '(' || ch == ')' || ch == '{' || ch == '}' || ch == '[' || ch == ']' || ch == ';' || ch == ',') {  // 分隔符

            if (ch == '(') out(LP, "(");

            else if (ch == ')') out(RP, ")");

            else if (ch == '{') out(BRACKET, "{");

            else if (ch == '}') out(BRACKET, "}");

            else if (ch == '[') out(BRACKET, "[");

            else if (ch == ']') out(BRACKET, "]");

            else if (ch == ';') out(DOT, ";");

            else out(DOT, ",");  // ch == ','

            column\_number++;

        }

        else if (ch == '=' || ch == '!' || ch == '<' || ch == '>' || ch == ':') {  // 关系运算符或赋值

            char next\_ch = fgetc(fp);

            column\_number++;

            if (ch == '=' && next\_ch == '=') {

                out(EQ, "");

            } else if (ch == '!' && next\_ch == '=') {

                out(NE, "");

            } else if (ch == '<' && next\_ch == '=') {

                out(LE, "");

            } else if (ch == '>' && next\_ch == '=') {

                out(GE, "");

            } else if (ch == ':' && next\_ch == '=') {

                out(IS, "");

            } else {

                fseek(fp, -1, SEEK\_CUR);  // 回退

                column\_number--;

                if (ch == '=') out(IS, " ");

                else if (ch == '!') out(NOT, " ");

                else if (ch == '<') out(LT, " ");

                else if (ch == '>') out(GT, " ");

                else report\_error("Unknown operator");

            }

        } else if (ch == '+' || ch == '-' || ch == '\*' || ch == '/') {  // 算术运算符

            if (ch == '+') out(PLUS, " ");

            else if (ch == '-') out(MINUS, " ");

            else if (ch == '\*') out(MUL, " ");

            else out(DIV, " ");  // ch == '/'

            column\_number++;

        } else {

            report\_error("Unknown character");

        }

        if (ch == EOF) {

            out(END, "");

            free(TOKEN);

            return;

        }

        ch = fgetc(fp);

        column\_number++;

    }

}

int Lexical(){

    FILE \*fp = fopen("C:\\Users\\Ava\\CLionProjects\\BY\_3\\input.txt", "r");

    if (fp == NULL) {

        printf("Error: Cannot open source file\n");

        return 99;

    }

    output\_file = fopen("C:\\Users\\Ava\\CLionProjects\\BY\_3\\output.txt", "w");

    if (output\_file == NULL) {

        printf("Error: Cannot open output file\n");

        return 98;

    }

    scanner\_example(fp);

    fclose(fp);

    fclose(output\_file);

    return 0;

}

1.5.2、lexical.h:

//

// Created by Ava on 2024/12/2.

//

#ifndef BY\_LEXICAL\_H

#define BY\_LEXICAL\_H

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_TOKEN\_LENGTH 100

#define TOKEN\_NAMES\_SIZE 28

void out(int code, char \*token);

void report\_error(const char \*error\_type);

// 定义类别码的助记符

typedef enum {

    ID,  PLUS, MINUS, MUL, DIV, LP, RP,END,

    FOR, IF, ELSE, AND, OR, NOT,

    INT, REAL,STRING,CHAR,OCTAL, HEX,

    EQ, NE, LT, GT, LE, GE,

    IS,BRACKET,DOT,

    UNKNOWN, ENTER

} TokenType;

extern const char\* token\_names[];

extern char \*TOKEN;

extern FILE \*output\_file;

int lookup(char \*token);

void scanner\_example(FILE \*fp);

int Lexical();

#endif //BY\_LEXICAL\_H

1.5.2、main.c:

#include <stdio.h>

#include "Lexical.h"

int main() {

    // 词法分析

    printf("词法分析结果如下：\n");

    int LexicalAnalysisFlag = Lexical();

    if (LexicalAnalysisFlag != 0) {

        printf("Lexical analysis failed\n");

        return LexicalAnalysisFlag;

    }

    readTokens("C:\\Users\\Ava\\CLionProjects\\BY\_3\\output.txt");

    printf("-----------------------------------------------------------------------\n");

    return 0;

}

1.6 实验拓展：

1.6.1、扩充关键字的数目、增加逻辑运算符等单词类别、将常数再细分成字符串常量、整型常量和无符号数常量等；添加词法分析中单词出错的位置和错误类型，以及删除注释部分等。

答：如1.5所示，已经扩展了；

1.6.2、用Flex实现词法分析：

答：

**1、增加的符号类型：**

增加了多个符号类型，包括：

* UNSIGNED\_INT：无符号整数
* STRING\_CONST：字符串常量
* BOOL：布尔值
* AND、OR、NOT：逻辑运算符
* 关系运算符（如 GE、LE、LT、GT、NE 等）均有定义。

**2、更复杂的词法规则：**

· **字符串常量**：通过 \"([^\\\"\n]|\\.)\*\" 识别的字符串。

· **逻辑运算符**：&&、||、! 被明确处理为独立的规则。

· **无符号整数**：通过 0-9]+[uU]? 识别，允许标识无符号整数。

**3、处理注释：**

· 实现了对单行注释 // 和多行注释 /\* ... \*/ 的跳过处理。这种处理确保了在分析过程中不会受到注释的影响。

**4、错误处理：**

· 包含了对错误字符的处理。当遇到未定义的字符时，会输出错误信息，指明该字符以及所在行数，提供更好的调试信息。

**5、输出功能的扩展：**

· 新代码中的输出函数 out 增加了对所有新符号类型的处理，使得在输出文件中能够反映更丰富的词法分析结果。

**6、**行号跟踪：****

· 新代码中引入了 line\_number 变量来跟踪行号，这使得在错误处理时能够给出准确的错误行号。

扩展后的代码如下：

%{

    #include <stdio.h>

    #include <stdlib.h>

    #include <string.h>

    #define ID 6

    #define INT 7

    #define LT 8

    #define LE 9

    #define EQ 10

    #define NE 11

    #define GT 12

    #define GE 13

    #define FLOAT 14

    #define STRING\_CONST 15

    #define UNSIGNED\_INT 16

    #define BOOL 17

    #define AND 18

    #define OR 19

    #define NOT 20

    FILE \*output\_fp;

    int line\_number = 1;

    int lookup(char \*token) {

        if (strcmp(token, "if") == 0) return 1;

        if (strcmp(token, "then") == 0) return 2;

        if (strcmp(token, "else") == 0) return 3;

        if (strcmp(token, "begin") == 0) return 4;

        if (strcmp(token, "end") == 0) return 5;

        if (strcmp(token, "true") == 0) return BOOL;

        if (strcmp(token, "false") == 0) return BOOL;

        return 0;

    }

    void out(int code, char \*val) {

        if (code == ID) {

            fprintf(output\_fp, "(ID, '%s')\n", val);

        } else if (code == INT) {

            fprintf(output\_fp, "(INT, '%s')\n", val);

        } else if (code == FLOAT) {

            fprintf(output\_fp, "(FLOAT, '%s')\n", val);

        } else if (code == STRING\_CONST) {

            fprintf(output\_fp, "(STRING\_CONST, '%s')\n", val);

        } else if (code == UNSIGNED\_INT) {

            fprintf(output\_fp, "(UNSIGNED\_INT, '%s')\n", val);

        } else if (code == BOOL) {

            fprintf(output\_fp, "(BOOL, '%s')\n", val);

        } else if (code >= 1 && code <= 5) {

            fprintf(output\_fp, "(KEYWORD, '%s')\n", val);

        } else if (code == AND) {

            fprintf(output\_fp, "(AND, '&&')\n");

        } else if (code == OR) {

            fprintf(output\_fp, "(OR, '||')\n");

        } else if (code == NOT) {

            fprintf(output\_fp, "(NOT, '!')\n");

        } else if (code == GE) {

            fprintf(output\_fp, "(GE, '>=')\n");

        } else if (code == EQ) {

            fprintf(output\_fp, "(EQ, '=')\n");

        } else if (code == LT) {

            fprintf(output\_fp, "(LT, '<')\n");

        } else if (code == LE) {

            fprintf(output\_fp, "(LE, '<=')\n");

        } else if (code == GT) {

            fprintf(output\_fp, "(GT, '>')\n");

        } else if (code == NE) {

            fprintf(output\_fp, "(NE, '!=')\n");

        } else {

            fprintf(output\_fp, "(%d, )\n", code);

        }

    }

    %}

    %%

    [ \t]+                ;

    \n                    { line\_number++; }

    "//".\*                { /\* Ignore single-line comments \*/ }

    "/\*"([^\*]|(\\*+[^\*/]))\*"\\*/"  { /\* Ignore multi-line comments \*/ }

    "if"|"then"|"else"|"begin"|"end"|"true"|"false" {

                            out(lookup(yytext), yytext);

                        }

    "&&"                  { out(AND, "&&"); }

    "||"                  { out(OR, "||"); }

    "!"                   { out(NOT, "!"); }

    \"([^\\\"\n]|\\.)\*\"  { out(STRING\_CONST, yytext); }

    [0-9]+                { out(INT, yytext); }

    [0-9]+[uU]?          { out(UNSIGNED\_INT, yytext); }

    [0-9]+\.[0-9]+       { out(FLOAT, yytext); }

    [a-zA-Z\_][a-zA-Z0-9\_]\* {

                            int code = lookup(yytext);

                            if (code == 0) out(ID, yytext);

                            else out(code, yytext);

                        }

    "<="                  { out(LE, " "); }

    ">="                  { out(GE, " "); }

    "<"                   { out(LT, " "); }

    ">"                   { out(GT, " "); }

    "="                   { out(EQ, " "); }

    "!="                  { out(NE, " "); }

    .                     {

                            fprintf(output\_fp, "Error: Invalid character '%s' at line %d\n", yytext, line\_number);

                        }

    %%

    int yywrap() {

        return 1;

    }

    int main() {

        FILE \*input\_fp;

        input\_fp = fopen("D:\\Flex2\\input.txt", "r");

        if (input\_fp == NULL) {

            perror("Unable to open input file!");

            return 1;

        }

        yyin = input\_fp;

        output\_fp = fopen("D:\\Flex2\\output.txt", "w");

        if (output\_fp == NULL) {

            perror("Unable to open output file!");

            fclose(input\_fp);

            return 1;

        }

        yylex();

        fclose(input\_fp);

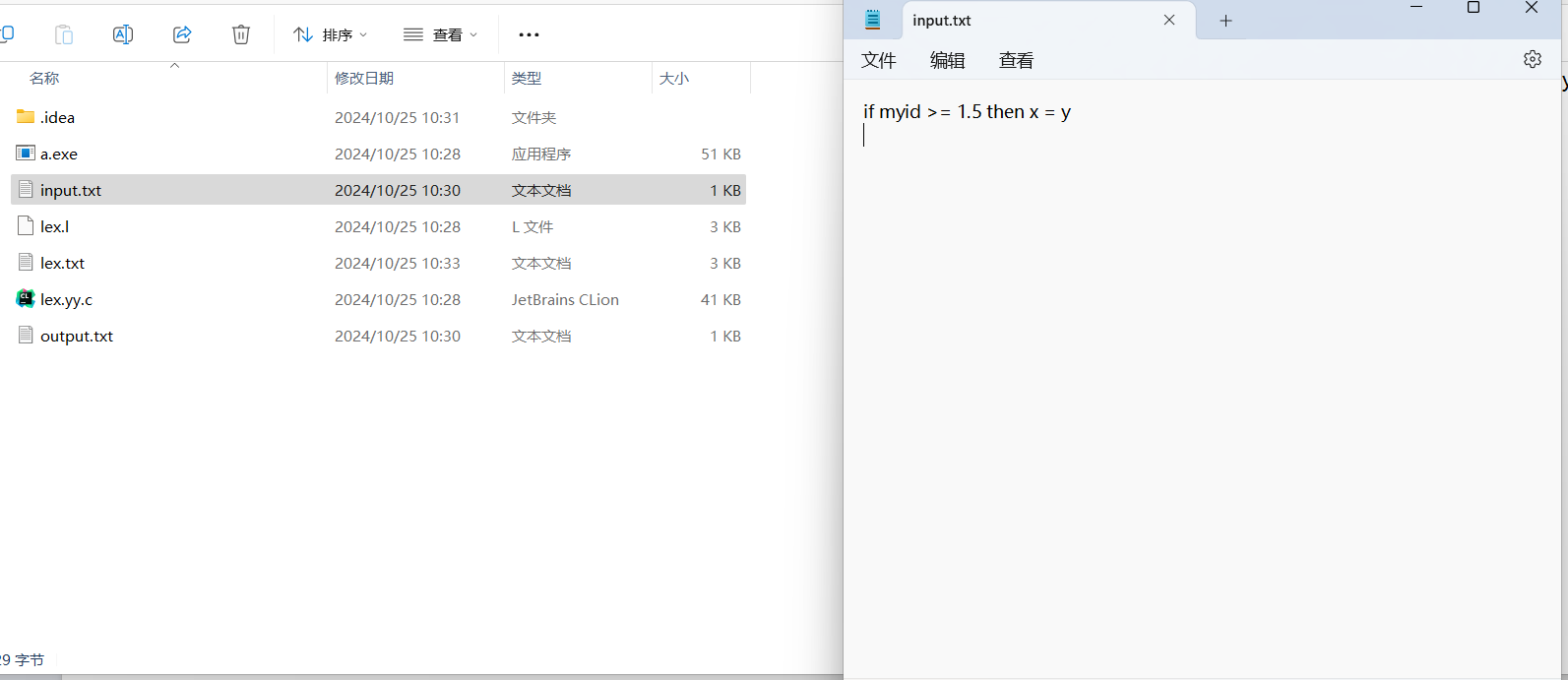
        fclose(output\_fp);

        return 0;

    }

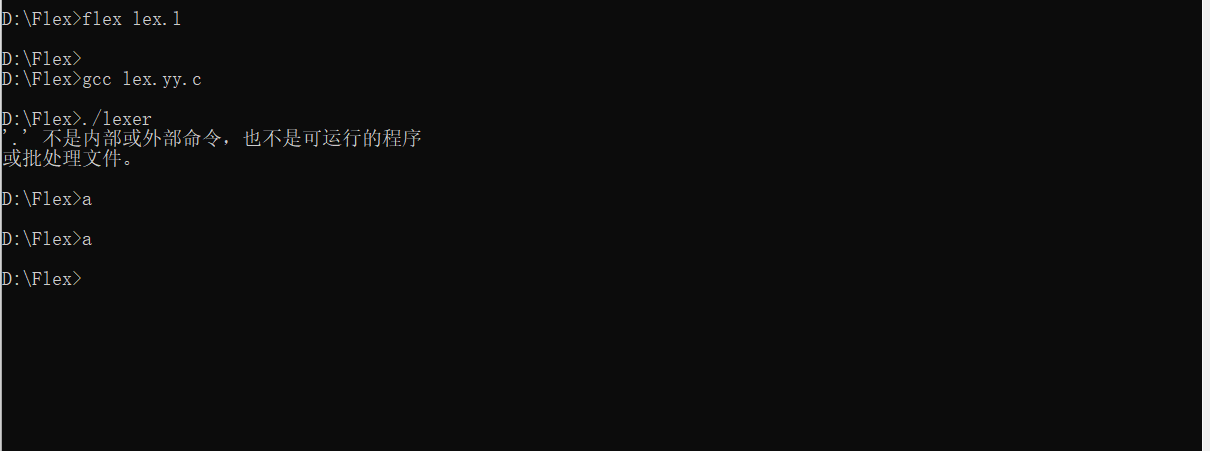
编译执行的步骤：

将文本文件改成“lex.l”，并提前创建好input.txt：



之后以管理员身份打开命令提示符，转到lex.l所在的位置，之后输入flex lex.l，再输入

gcc lex.yy.c，会生成一个a.exe文件，之后再输入a并按下回车，则会发现已生成了output.txt文件：



input.txt的内容：

if myid >= 1.5 then x = y

// This is a single-line comment

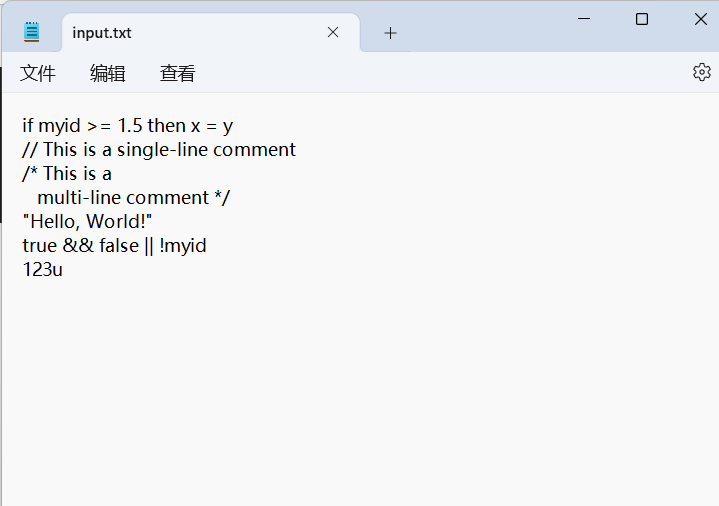
/\* This is a

multi-line comment \*/

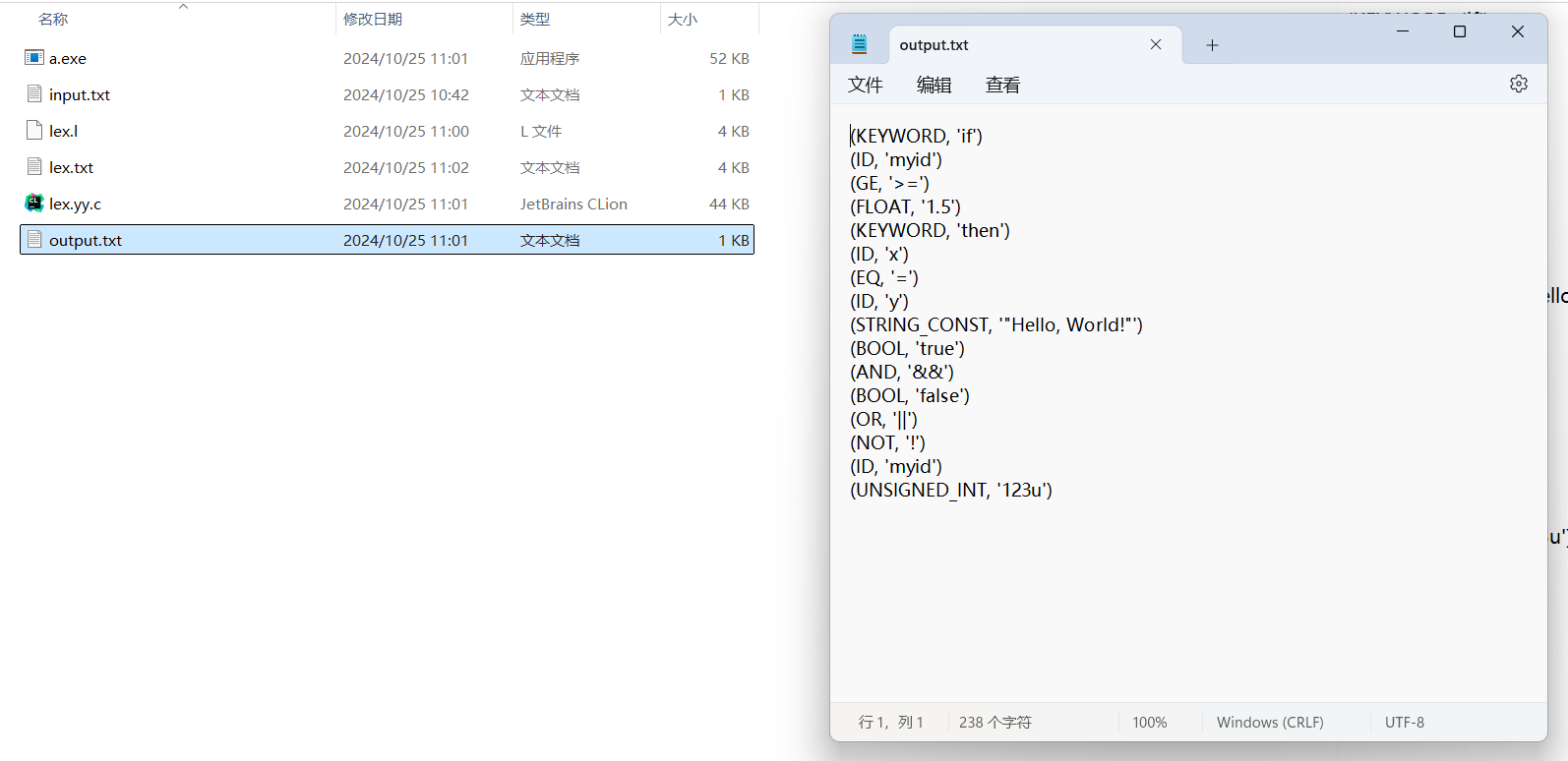
"Hello, World!"

true && false || !myid

123u



Output.txt的结果如下：



实验二 语法分析程序设计与实现

2.1 语法结构范围及文法定义

### **2.1.1、语法结构范围**

****基础操作数****：支持整数作为基本的操作数，代码中通过对词法单元类型为 INT 的处理来识别整数，例如像 3、5 这样的单个整数可以作为表达式中的基础元素参与运算。

****括号嵌套****：能够处理带括号的表达式，允许括号进行嵌套，如 (3 + 4) 或者更复杂的 ((3 + 4) \* 2) 等形式，通过 factor 函数中对左括号 LP 和右括号 RP 的处理逻辑实现括号内表达式作为一个整体因子参与外层表达式的运算。

****二元运算组合****：涵盖了加法（PLUS）、减法（MINUS）、乘法（MUL）、除法（DIV）这几种常见的二元运算，并且可以按照合理的语法规则组合这些运算来构建复杂的表达式，比如 3 + 4 \* 2、(3 + 4) / 2 - 1 等形式的表达式都在其处理范围内。通过 term 函数处理乘法和除法相关的子表达式（项），expr 函数在此基础上处理加法和减法构成的整体表达式，按照操作符优先级依次进行运算。

### **2.1.2、文法定义**

**G2[E]：**E → T | E+T | E-T T → F | T\*F | T/F F → i | (E)

2.2 语法分析法(分析表及分析过程)设计

**以算符优先法为例：**

2.2.1、根据文法构造出算符优先表：

|  | **(** | **i** | **\*** | **+** | **)** | **#** |
| --- | --- | --- | --- | --- | --- | --- |
| ( | < | < | < | < | = |  |
| i |  |  | > | > | > | > |
| \* | < | < | > | > | > | > |
| + | < | < | < | > | > | > |
| ) |  |  |  |  | > |  |
| # | < | < | < | < |  |  |

2.2.2、分析过程设计：

#### **1. 初始化**

****操作数栈和操作符栈初始化****：操作数栈 stack 初始为空，操作符栈 opStack 也为空，准备接收词法分析后的词法单元对应的操作数和操作符。

****设置输入指针****：将指向词法单元序列（tokens）的当前指针 currentToken 设置为起始位置，准备从左到右读取词法单元进行分析。

#### **2. 分析循环**

* 不断从输入的词法单元序列中读取下一个词法单元（通过 tokens[currentToken] 获取），直到遇到表示表达式结束的 END 词法单元。
* 对于读取到的每个词法单元：
  + 如果是操作数（词法单元类型为 INT），直接将其压入操作数栈 stack，并将 currentToken 指向下一个词法单元，继续读取后续内容。
  + 如果是操作符（如 PLUS、MINUS、MUL、DIV 等）：
    - 首先查看操作符栈 opStack 是否为空。如果为空，则直接将当前操作符压入操作符栈 opStack，继续读取下一词法单元。
    - 如果操作符栈不为空，则获取栈顶操作符 topOp。然后查询基于优先矩阵构建的分析表，根据 topOp 和当前操作符的组合情况决定动作：
      * 如果分析表中对应的动作是 “移进”，则将当前操作符压入操作符栈 opStack，继续读取下一词法单元。
      * 如果是 “归约”，则需要根据当前的语法结构进行归约操作。例如，如果是对乘法或除法的归约（如栈顶为 MUL 或 DIV，且当前操作符优先级低于栈顶操作符），则从操作数栈弹出两个操作数，根据栈顶操作符进行计算，将结果压回操作数栈，然后再次检查操作符栈顶与当前操作符的优先级关系（因为归约操作可能改变了栈内操作符的相对优先级情况），重复这个过程直到分析表指示移进或者接受。对于加法和减法的归约情况类似，不过要遵循加法和减法之间以及它们与乘法、除法的优先级关系（由优先矩阵确定）。
      * 如果是 “接受”，表示表达式分析成功完成，此时操作数栈中应该只剩下最终的计算结果，输出该结果并结束分析过程。

#### **3. 错误处理**

* 在分析过程中，如果遇到不符合分析表规则的操作符组合（例如，分析表中未定义的两个操作符相遇情况），或者操作数栈或操作符栈出现异常（如空栈时进行弹出操作等），则调用 parseError 函数报告错误，并终止分析过程。例如，如果在分析过程中发现当前操作符与栈顶操作符的组合在分析表中没有对应的有效动作，就说明输入的表达式可能存在语法错误，需要进行错误提示。

通过以上基于优先矩阵构建的分析表以及相应的分析过程设计，代码能够对输入的表达式进行语法分析，按照操作符的优先级关系正确地处理操作数和操作符，最终实现表达式的计算或者语法结构的构建（如果进一步扩展功能可以构建语法树等），并且能够对一些语法错误情况进行检测和报告。

2.3 语法分析程序（数据结构、函数、程序结构等）设计

### **2.3.1、数据结构设计**

#### **1. 操作数栈（stack）**

* ****定义与声明****：  
  通过 int stack[100]; 定义了一个大小为 100 的整型数组来作为操作数栈，用于存储表达式中的操作数（如整数）。同时，使用整型变量 top（初始值为 -1）来指示栈顶位置，通过 top 的变化来反映栈的状态，如 top 自增 1 表示入栈操作，top 减 1 表示出栈操作。
* ****作用与操作逻辑****：  
  在语法分析过程中，当识别到词法单元类型为整数（通过 tokens[currentToken].type == INT 判断）时，会调用 push 函数将该整数（利用 atoi 函数将字符串形式转换为整型后）压入操作数栈。例如，对于表达式 3 + 4，当处理到 3 这个整数词法单元时，会将 3 压入 stack 中，后续在进行运算（如加法、减法等操作涉及操作数的获取时），会通过 pop 函数从栈中弹出操作数进行相应计算，计算结果又可再次压入栈中，方便按照算符优先规则对操作数进行管理和运算处理。

#### **2. 操作符栈（opStack）**

* ****定义与声明****：  
  利用 char opStack[100]; 定义了同样大小为 100 的字符数组作为操作符栈，用来存放操作符（像 +、-、\*、/、(、) 等）。与之对应的是整型变量 opTop（初始值为 -1），用于标记操作符栈的栈顶位置，控制操作符的入栈和出栈操作。
* ****作用与操作逻辑****：  
  当读取到表达式中的操作符词法单元时，会根据算符优先规则决定是否将操作符压入 opStack。例如，在处理 3 + 4 \* 2 这个表达式时，先遇到 +，会按照规则判断其与后续操作符以及栈内已有操作符的优先级关系，来决定 + 是否入栈等操作；在后续遇到 \* 时，根据优先级（乘法优先级高于加法），会先将 \* 压入 opStack，以此类推，通过操作符栈来有序地管理操作符，确保按照正确的算符优先顺序进行表达式的分析和计算。

#### **3. 词法单元数组（tokens）及相关索引变量**

* ****定义与声明****：  
  虽然代码中未完整展示 tokens 数组的定义（推测其是一个结构体数组，结构体中包含词法单元类型、对应的值等成员，用于存储词法分析得到的结果），但从代码逻辑中可以看到通过 tokens[currentToken] 来访问当前正在处理的词法单元。这里 currentToken 是一个整型变量，用于标记当前处理词法单元在 tokens 数组中的位置，初始值通常为 0，随着语法分析的进行，会不断自增以按顺序读取词法单元进行后续分析。
* ****作用与操作逻辑****：  
  tokens 数组作为语法分析的输入数据源，提供了已经经过词法分析后得到的词法单元序列。语法分析程序依据每个词法单元的类型（如 INT、PLUS、LP 等）以及其对应的值（对于整数是具体数值，操作符可能为空等情况），按照算符优先文法规则进行相应的处理，比如遇到 INT 类型进行操作数入栈操作，遇到操作符类型则进行优先级比较和相应的栈操作等，是整个语法分析流程的基础数据支撑。

#### **4. 优先级矩阵（precedence）**

* ****定义与声明****：  
  使用二维数组 int precedence[9][9]; 定义了优先级矩阵，其中行和列的索引对应不同的操作符（按照代码里设定的操作符顺序，比如 + 对应索引 0 等），通过矩阵元素的值（如 0、1、2 等）来表示两个操作符之间的优先级比较情况。例如，precedence[0][2] 的值表示 + 和 \* 这两个操作符之间的优先级关系（代码中设定体现了乘法优先级高于加法）。
* ****作用与操作逻辑****：  
  在语法分析过程中，每当遇到一个新的操作符时，需要通过 comparePrecedence 函数查询这个优先级矩阵，比较当前操作符与操作符栈顶操作符（如果操作符栈非空）的优先级关系，以此来决定是将当前操作符压入操作符栈（如当前操作符优先级高于栈顶操作符优先级时），还是进行归约操作（例如当前操作符优先级低于或等于栈顶操作符优先级时，可能需要根据栈顶操作符先进行相应运算，即归约操作）等，它是整个算符优先语法分析的核心依据，决定了操作符处理的先后顺序和表达式分析的流程走向。

### **2.3.2、函数设计**

#### **1. 栈操作相关函数（push、pop、pushOp、popOp）**

**push**函数****：

* + ****输入参数****：int value，即要压入操作数栈的整数值。
  + ****功能实现****：先将表示操作数栈栈顶位置的变量 top 自增 1，然后把传入的 value 存储到 stack[top] 位置，实现将操作数压入栈的操作，同时在控制台打印出 Pushed to stack: [具体数值] 的信息，方便调试查看操作数入栈情况。例如，调用 push(5) 时，top 会从初始的 -1 变为 0，且 stack[0] 的值被设置为 5，并在控制台显示相应入栈提示。
  + ****作用****：用于在语法分析过程中，将识别到的整数操作数压入操作数栈，为后续基于算符优先规则进行的运算操作准备操作数。

**pop**函数****：

* + ****功能实现****：首先检查操作数栈是否为空（通过判断 top 是否为 -1），若为空则调用 parseError 函数报告 “Stack underflow”（栈下溢）错误并终止程序；若栈非空，则获取 stack[top] 的值作为要返回的操作数，接着将 top 减 1，实现弹出操作数的功能，同时在控制台打印 Popped from stack: [具体数值] 的提示信息，最后返回弹出的操作数供后续运算使用。例如，若操作数栈中已有元素，调用 pop 函数会返回栈顶元素，比如栈顶元素为 5，调用后 top 减 1，且返回值为 5，控制台会显示相应弹出提示。
  + ****作用****：在根据操作符进行相应运算（如加法、减法、乘法、除法等）时，从操作数栈中获取所需的操作数，保证运算能够按照算符优先规则顺利进行。

**pushOp**函数****：

* + ****输入参数****：char op，即要压入操作符栈的操作符字符。
  + ****功能实现****：先将表示操作符栈栈顶位置的变量 opTop 自增 1，再把传入的操作符 op 存入 opStack[opTop] 位置，完成操作符入栈操作，并且在控制台打印 Pushed to operator stack: [具体操作符字符] 的信息，便于观察操作符栈的操作情况。例如，调用 pushOp('+')，opTop 会从初始的 -1 变为 0，且 opStack[0] 的值变为 '+'，控制台会显示相应入栈提示。
  + ****作用****：在读取到表达式中的操作符词法单元时，根据算符优先规则判断若需要将操作符压入栈，就调用该函数实现操作符入栈，以此来有序管理操作符在栈中的存储，确保后续能按照正确优先级顺序处理操作符。

**popOp**函数****：

* + ****功能实现****：先检查操作符栈是否为空（通过判断 opTop 是否为 -1），若为空则调用 parseError 函数报告 “Operator stack underflow”（操作符栈下溢）错误并终止程序；若不为空，则获取 opStack[opTop] 的操作符作为返回值，随后将 opTop 减 1，完成弹出操作符的操作，同时在控制台打印 Popped from operator stack: [具体操作符字符] 的提示信息，最后返回弹出的操作符用于后续判断优先级和执行归约等操作。例如，若操作符栈栈顶元素为 '+'，调用 popOp 函数后，opTop 减 1，返回值为 '+'，控制台会显示相应弹出提示。
  + ****作用****：在进行归约操作或者比较当前操作符与栈顶操作符优先级等情况时，需要获取栈顶操作符，该函数提供了弹出栈顶操作符的功能，便于后续基于栈顶操作符进行相关处理，如根据操作符优先级决定下一步的操作符入栈、归约运算等操作流程。

#### **2. comparePrecedence 函数**

* ****输入参数****：接收两个整型参数 int op1 和 int op2，这两个参数分别对应不同操作符在 precedence 优先级矩阵中的索引，代表要比较优先级的两个操作符。
* ****功能实现****：通过查找 precedence 二维数组，返回 precedence[op1][op2] 位置的元素值，以此来确定 op1 和 op2 所代表的两个操作符之间的优先级关系。例如，若 op1 对应 + 操作符索引（假设为 0），op2 对应 \* 操作符索引（假设为 2），则返回 precedence[0][2] 的值（代码中为 1，表示 + 的优先级低于 \*），从而为操作符的处理决策（如入栈还是归约等）提供依据。
* ****作用****：在语法分析流程中，每当遇到新的操作符词法单元时，需要和操作符栈顶已有的操作符比较优先级，该函数实现了快速准确获取两个操作符优先级关系的功能，使得整个分析流程能够依据算符优先规则顺利推进，决定是将新操作符压入栈还是基于栈顶操作符进行归约操作等。

#### **3. factor 函数**

* ****功能实现****：用于处理文法中的因子（对应算符优先文法中的 F）部分，按照当前词法单元的类型执行不同操作：
  + 当 tokens[currentToken].type == INT（即当前词法单元为整数类型）时，首先在控制台打印 Push [具体整数数值] 的信息，然后调用 push 函数将该整数（通过 atoi 函数把词法单元对应的值字符串转换为整型后）压入操作数栈，接着调用 match 函数（推测其功能是匹配并推进当前词法单元索引，确认处理了期望的词法单元类型）来处理这个整数词法单元，完成整数作为因子的处理流程。
  + 若 tokens[currentToken].type == LP（当前词法单元为左括号类型），则先调用 match 函数匹配这个左括号词法单元，然后递归调用 expr 函数来解析括号内的表达式，待括号内表达式处理完成后，再调用 match 函数匹配右括号（RP）词法单元，以此实现括号内表达式作为一个因子的正确处理。
  + 如果当前词法单元类型不符合上述整数或左括号的情况，则调用 parseError 函数报告 “Invalid factor”（无效因子）错误，终止程序，确保输入的表达式在因子部分符合算符优先文法规则。
* ****作用****：实现了对表达式中最基础的因子单元的语法分析和处理，是构建更复杂表达式分析的基础环节，按照文法规则准确处理操作数和括号内表达式等作为因子的情况，为后续项和整个表达式的分析提供正确的基础数据和结构支撑。

#### **4. term 函数**

* ****功能实现****：负责处理文法中的项（对应算符优先文法中的 T）部分，其处理逻辑如下：
  + 首先调用 factor 函数处理第一个因子，这是构建项的基础操作，先确保项中的首个因子被正确处理并存入操作数栈等。
  + 然后进入一个循环，只要当前词法单元的类型是乘法（MUL）或者除法（DIV）运算符（通过 tokens[currentToken].type == MUL || tokens[currentToken].type == DIV 判断），就执行以下操作：
    - 先在控制台打印 Perform operation: [操作符名称（如 MUL 或 DIV）] 的信息，用于展示当前正在进行的操作相关提示，方便调试查看。
    - 获取当前操作符对应的类型索引（存于 op 变量），接着调用 match 函数匹配该操作符词法单元，然后再次调用 factor 函数处理操作符后面的因子（因为乘法或除法是二元运算符，需要前后两个因子）。
    - 从操作数栈中依次弹出两个操作数（先弹出的作为右操作数 right，后弹出的作为左操作数 left），根据操作符类型（是 MUL 则进行乘法运算，是 DIV 则进行除法运算）计算结果，并将结果压入操作数栈 stack 中，以此完成一项包含乘法或除法运算的表达式部分的处理，循环处理直到该项中不再有乘法或除法运算符为止，确保按照算符优先规则中乘法、除法的优先级顺序正确处理项这一语法结构。
* ****作用****：依据算符优先文法规则，处理表达式中由因子通过乘法或除法运算符连接而成的项部分，通过循环处理乘法和除法运算，结合操作数栈和操作符栈的操作，实现对项结构的正确分析和计算，为后续整个表达式的完整分析打下基础。

#### **5. expr 函数**

* ****功能实现****：用于处理整个表达式（对应算符优先文法中的 E），其逻辑如下：
  + 首先调用 term 函数处理第一个项，将表达式中的首个项按照算符优先规则进行处理，把相应的操作数压入操作数栈，操作符根据优先级压入或处理等操作，构建表达式分析的起始部分。
  + 接着进入一个循环，只要当前词法单元的类型是加法（PLUS）或者减法（MINUS）运算符（通过 tokens[currentToken].type == PLUS || tokens[currentToken].type == MINUS 判断），就执行以下操作：
    - 同样先在控制台打印 Perform operation: [操作符名称（如 PLUS 或 MINUS）] 的信息，展示当前操作相关提示。
    - 获取当前操作符对应的类型索引（存于 op 变量），然后调用 match 函数匹配该操作符词法单元，再调用 term 函数处理操作符后面的项。
    - 从操作数栈中依次弹出两个操作数（分别作为右操作数 right 和左操作数 left），根据操作符类型（是 PLUS 则进行加法运算，是 MINUS 则进行减法运算）计算结果，并将结果压入操作数栈 stack 中，如此循环处理直到表达式中不再有加法或减法运算符为止，按照算符优先规则中加法、减法的优先级以及与乘法、除法的优先级关系，逐步处理整个表达式，完成表达式主体部分的分析与计算过程（不包含处理操作符栈剩余操作符的部分）。
* ****作用****：按照算符优先文法要求，处理整个表达式结构，通过循环处理加法和减法运算，结合对项的处理以及操作数栈、操作符栈的操作，实现对表达式从左到右按照算符优先级顺序的正确分析和计算，是语法分析的核心处理函数之一，将词法单元序列逐步转换为最终的计算结果（结合后续 parse\_SF 函数完成全部流程）。

#### **6. parse\_SF 函数**

* ****功能实现****：作为整个语法分析和表达式计算的总入口函数，其执行的操作如下：
  + 首先调用 expr 函数来解析和计算表达式（按照上述 expr 函数的逻辑处理表达式中的各项和运算符等），完成表达式主体部分的计算后，会进入一个循环处理操作符栈 opStack 中剩余的操作符（在处理表达式过程中，操作符可能会根据优先级规则暂存到操作符栈中）：
    - 从操作符栈中弹出一个操作符（通过 popOp 函数），再从操作数栈中弹出两个操作数（分别作为右操作数 right 和左操作数 left），然后根据操作符类型（是 PLUS、MINUS、MUL 还是 DIV）进行相应的运算（加法、减法、乘法或除法），并将运算结果压入操作数栈 stack 中，循环处理直到操作符栈为空。
    - 最后从操作数栈中弹出最终的计算结果（通过 pop 函数），并在控制台打印出 Final result: [具体数值] 的信息，展示整个表达式的最终计算结果，完成整个语法分析与计算流程，实现从输入的表达式词法单元序列到最终计算结果的转换

2.4 语法分析测试结果分析

**2.4.1 示例输入与其结果：**

当input.txt的内容是3 + 4 \* 5 - (2 + 1) \* 7，输出是：

-----------------------------------------------------------------------

算符优先结果如下：

Push 3

Pushed to stack: 3

Perform operation: PLUS

Push 4

Pushed to stack: 4

Perform operation: MUL

Push 5

Pushed to stack: 5

Popped from stack: 5

Popped from stack: 4

Pushed to stack: 20

Popped from stack: 20

Popped from stack: 3

Pushed to stack: 23

Perform operation: MINUS

Push 2

Pushed to stack: 2

Perform operation: PLUS

Push 1

Pushed to stack: 1

Popped from stack: 1

Popped from stack: 2

Pushed to stack: 3

Perform operation: MUL

Push 7

Pushed to stack: 7

Popped from stack: 7

Popped from stack: 3

Pushed to stack: 21

Popped from stack: 21

Popped from stack: 23

Pushed to stack: 2

Popped from stack: 2

Final result: 2

-----------------------------------------------------------------------

### **2.4.2、整体分析思路**

从输出结果可以看出，语法分析程序按照算符优先的规则，对输入的表达式进行了逐步的分析和计算，通过操作数栈和操作符栈的交互操作，以及依据算符之间的优先级关系来处理各个子表达式和运算符，最终得出了正确的计算结果。下面将按照输出顺序，详细分析每一步的操作及其对应的算符优先处理逻辑。

### **2.4.3、逐个步骤分析**

#### 1. “Push 3

Pushed to stack: 3”

****含义及分析****：这表示语法分析程序首先识别到了输入表达式中的第一个操作数 “3”，根据代码中的 factor 函数逻辑（当遇到整数类型的词法单元时），将其转换为整型后压入操作数栈 stack，并打印出相应的入栈提示信息。这是表达式分析的起始步骤，操作数栈此时存放了第一个操作数，为后续的运算做好准备，符合算符优先分析法先处理操作数的基本流程。

#### **2.** “Perform operation: PLUS

Push 4  
Pushed to stack: 4”

****含义及分析****：当遇到 “+” 运算符时，程序输出了当前要执行的操作提示 “Perform operation: PLUS”，表示开始处理加法运算相关部分。但由于加法运算符的优先级低于乘法运算符（在代码的 precedence 优先级矩阵中有相应设定），此时不会立即进行加法运算，而是继续读取下一个操作数。接着识别到操作数 “4”，按照处理操作数的逻辑，将其压入操作数栈，并打印入栈提示，这体现了算符优先分析法中按照从左到右顺序读取表达式以及根据优先级决定操作顺序的特点，在遇到低优先级操作符（加法）时，先暂存后续操作数，等待合适时机进行运算。

#### 3. “Perform operation: MUL

Push 5  
Pushed to stack: 5  
Popped from stack: 5  
Popped from stack: 4  
Pushed to stack: 20”

****含义及分析****：遇到 “\*” 运算符后，输出操作提示 “Perform operation: MUL”，表明开始处理乘法运算。然后识别到操作数 “5” 并压入操作数栈。接下来，因为乘法的优先级高于加法，所以要立即进行乘法运算。从操作数栈中依次弹出两个操作数 “5” 和 “4”（按照栈的后进先出原则，先弹出的作为右操作数，后弹出的作为左操作数），进行乘法计算（4 \* 5 = 20），并将结果 “20” 压回操作数栈。这一系列操作严格遵循了算符优先规则，先处理高优先级的乘法运算，保证了表达式计算顺序的正确性。

#### 4. “Popped from stack: 20

Popped from stack: 3  
Pushed to stack: 23”

****含义及分析****：在完成乘法运算后，由于之前加法运算被暂时搁置，此时乘法运算的结果 “20” 和之前暂存的第一个操作数 “3”（它们分别是加法运算的两个操作数），按照加法运算的逻辑，从操作数栈中弹出这两个数，进行加法计算（3 + 20 = 23），并将结果 “23” 压回操作数栈，完成了 “3 + 4 \* 5” 这部分表达式中先乘后加的运算，体现了算符优先分析中根据操作符优先级依次处理运算的过程，确保了这一子表达式计算结果的正确生成和存储。

#### 5. “Perform operation: MINUS

Push 2  
Pushed to stack: 2”

****含义及分析****：接着遇到 “-” 运算符，输出操作提示 “Perform operation: MINUS”，表示开始处理减法运算相关部分。和加法类似，减法的优先级低于乘法，所以先继续读取下一个操作数。识别到操作数 “2” 后，将其压入操作数栈，等待后续根据优先级情况决定何时进行减法运算，同样遵循了算符优先从左到右读取以及按优先级处理的原则。

#### 6. “Perform operation: PLUS

Push 1  
Pushed to stack: 1  
Popped from stack: 1  
Popped from stack: 2  
Pushed to stack: 3”

****含义及分析****：遇到括号内的 “+” 运算符时，输出相应操作提示，然后识别并压入操作数 “1”。因为括号内的表达式按照算符优先规则要先进行内部运算，所以接下来从操作数栈中弹出 “1” 和 “2” 进行加法运算（2 + 1 = 3），将结果 “3” 压回操作数栈，完成了括号内表达式 “(2 + 1)” 的计算，这符合括号具有最高优先级，先处理括号内表达式的语法和算符优先原则，保证了整个表达式中这一局部结构的正确求值。

#### 7. “Perform operation: MUL

Push 7  
Pushed to stack: 7  
Popped from stack: 7  
Popped from stack: 3  
Pushed to stack: 21”

****含义及分析****：在处理完括号内表达式后，遇到 “\*” 运算符，输出操作提示并压入操作数 “7”。由于乘法优先级高于减法，所以要先进行乘法运算，从操作数栈中弹出 “7” 和 “3” 进行乘法计算（3 \* 7 = 21），将结果 “21” 压回操作数栈，完成了 “(2 + 1) \* 7” 这部分的乘法运算，再次体现了根据算符优先级来决定运算顺序的处理逻辑。

#### 8. “Popped from stack: 21

Popped from stack: 23  
Pushed to stack: 2”

****含义及分析****：此时，之前暂存的减法运算可以进行了，从操作数栈中弹出 “21”（作为减数）和 “23”（作为被减数），进行减法计算（23 - 21 = 2），将结果 “2” 压回操作数栈，完成了整个表达式 “3 + 4 \* 5 - (2 + 1) \* 7” 按照算符优先规则的全部运算过程，操作数栈中最终剩下的就是整个表达式的计算结果。

#### 9. “Popped from stack: 2

Final result: 2”

****含义及分析****：最后从操作数栈中弹出最终的计算结果 “2”，并输出 “Final result: 2”，展示了整个表达式经过语法分析和按算符优先规则计算后得到的正确结果，完成了从输入表达式到最终计算值的完整处理流程。

### **2.4.4、结果验证与总结**

从整个分析过程来看，对于输入的表达式 “3 + 4 \* 5 - (2 + 1) \* 7”，语法分析程序准确地按照算符优先的规则，通过操作数栈和操作符栈的合理操作，依次处理各个操作数、运算符以及括号内的表达式，正确地处理了不同操作符之间的优先级关系，最终得出了预期的计算结果 “2”。这表明该语法分析程序在处理此类包含多种运算符和括号的表达式时，其算符优先的分析逻辑是正确有效的，能够实现对表达式的正确语法分析与计算，符合算符优先分析法的预期功能。

综上所述，给定的这个语法分析测试结果是合理且符合算符优先分析法要求的，反映了程序对输入表达式进行正确语法分析和逐步计算的完整过程。

2.5 实验代码：

2.5.1、lexical.c:同1.5.1部分代码；

2.5.2、lexical.h:同1.5.2部分代码；

2.5.3、GramAnalysis.c:

#include "GramAnalysis.h"

void parseError(const char \*message) {

    printf("ERROR: %s at token %d (%s,%s),%d:%d\n", message, currentToken, token\_names[tokens[currentToken-1].type],tokens[currentToken-1].value,tokens[currentToken-1].row, tokens[currentToken-1].col);

    exit(1);

}

Token getNextToken() {

    return tokens[currentToken++];

}

void retractToken() {

    currentToken--;

}

void readTokens(const char \*filename) {

    FILE \*file = fopen(filename, "r");

    if (!file) {

        printf("Error: Cannot open token file\n");

        exit(1);

    }

    char typeStr[MAX\_TOKEN\_LENGTH];

    char value[MAX\_TOKEN\_LENGTH];

    int row = 1;

    int col = 1;

    while (1) {

        fscanf(file, "(%[^,],%[^)])", typeStr, value);

        fgetc(file); // 读取并丢弃右括号

        if (strcmp(typeStr, "ENTER") == 0) {

            row++;

            col = 1;

            continue;

        }

        for (int i = 0; i < TOKEN\_NAMES\_SIZE; i++) {

            if (strcmp(typeStr, token\_names[i]) == 0) {

                tokens[tokenIndex].type = i;

                break;

            }

        }

        strcpy(tokens[tokenIndex].value, value);

        tokens[tokenIndex].row = row;

        tokens[tokenIndex].col = col;

        col ++;

        // 将字符串类型转换为枚举类型

        printf("typeStr: %s\n", typeStr);

        printf("value: %s\n", tokens[tokenIndex].value);

        if (strcmp(typeStr, "RP") == 0) {

            fgetc(file);

        }

        if(strcmp(typeStr, "END") == 0) {

            break;

        }

        tokenIndex++;

        if (tokenIndex >= 100) {

            printf("Error: Too many tokens\n");

            exit(1);

        }

    }

    tokens[tokenIndex].type = END;

    fclose(file);

}

// 实现match函数

void match(int expectedType) {

    if (tokens[currentToken].type == expectedType) {

        currentToken++;

    } else {

        parseError("Unexpected token");

    }

}

2.5.4、GramAnalysis.h:

#ifndef BY\_GRAMANALYSIS\_H

#define BY\_GRAMANALYSIS\_H

#include "Lexical.h"

typedef struct {

    TokenType type;

    char value[100];

    int row;

    int col;

} Token;

extern Token tokens[100];

extern int tokenIndex;

extern int currentToken;

void readTokens(const char \*filename);

Token getNextToken();

void retractToken();

void parseError(const char \*message);

void match(int expectedType);

#endif //BY\_GRAMANALYSIS\_H

2.5.5、SuanFu.c:

#include <stdio.h>

#include <stdlib.h>

#include "SuanFu.h"

#include "Lexical.h"

#include "GramAnalysis.h"

// 栈操作实现

int stack[100];

int top = -1;

char opStack[100]; // 栈用于保存操作符

int opTop = -1; // 操作符栈的栈顶

// 栈操作输出：确保推入栈的是正确的操作符

void push(int value) {

    stack[++top] = value;

    printf("Pushed to stack: %d\n", value);  // 打印每次推入栈的值

}

int pop() {

    if (top == -1) {

        parseError("Stack underflow");

    }

    int value = stack[top--];

    printf("Popped from stack: %d\n", value);  // 打印每次弹出栈的值

    return value;

}

void pushOp(char op) {

    opStack[++opTop] = op;  // 操作符压栈

    printf("Pushed to operator stack: %c\n", op);

}

char popOp() {

    if (opTop == -1) {

        parseError("Operator stack underflow");

    }

    char op = opStack[opTop--];

    printf("Popped from operator stack: %c\n", op);

    return op;

}

// 优先级矩阵

int precedence[9][9] = {

        {0, 1, 1, 1, 1, 1, 0, 0, 0},   // + (index 0)

        {2, 0, 2, 2, 2, 2, 0, 0, 0},   // - (index 1)

        {2, 2, 0, 2, 2, 2, 0, 0, 0},   // \* (index 2)

        {2, 2, 2, 0, 2, 2, 0, 0, 0},   // / (index 3)

        {2, 2, 2, 2, 0, 2, 0, 0, 0},   // ( (index 4)

        {1, 1, 1, 1, 1, 0, 0, 0, 0},   // ) (index 5)

        {1, 1, 1, 1, 1, 1, 0, 0, 0},   // END (index 6)

        {0, 0, 0, 0, 0, 0, 1, 0, 0},   // INT (index 7)

        {0, 0, 0, 0, 0, 0, 0, 0, 0},   // Token (index 8)

};

// 判断操作符优先级

int comparePrecedence(int op1, int op2) {

    return precedence[op1][op2];

}

// 因子分析：处理整数或括号中的表达式

void factor() {

    if (tokens[currentToken].type == INT) {

        printf("Push %s\n", tokens[currentToken].value);

        push(atoi(tokens[currentToken].value));  // 将整数推入栈

        match(INT);

    } else if (tokens[currentToken].type == LP) {

        match(LP);

        expr();  // 递归解析括号中的表达式

        match(RP);

    } else {

        parseError("Invalid factor");

    }

}

// 项分析：处理乘法和除法

void term() {

    factor();

    while (tokens[currentToken].type == MUL || tokens[currentToken].type == DIV) {

        printf("Perform operation: %s\n", token\_names[tokens[currentToken].type]);

        int op = tokens[currentToken].type;

        match(tokens[currentToken].type);

        factor();

        int right = pop();

        int left = pop();

        if (op == MUL) {

            push(left \* right);

        } else if (op == DIV) {

            push(left / right);

        }

    }

}

// 表达式分析：处理加法和减法

void expr() {

    term();

    while (tokens[currentToken].type == PLUS || tokens[currentToken].type == MINUS) {

        printf("Perform operation: %s\n", token\_names[tokens[currentToken].type]);

        int op = tokens[currentToken].type;

        match(tokens[currentToken].type);

        term();

        int right = pop();

        int left = pop();

        if (op == PLUS) {

            push(left + right);

        } else if (op == MINUS) {

            push(left - right);

        }

    }

}

// 语法分析函数，执行整个表达式的解析和计算

void parse\_SF() {

    expr();

    while (opTop != -1) {

        // 处理剩余的操作符

        char op = popOp();

        int right = pop();

        int left = pop();

        if (op == PLUS) {

            push(left + right);

        } else if (op == MINUS) {

            push(left - right);

        } else if (op == MUL) {

            push(left \* right);

        } else if (op == DIV) {

            push(left / right);

        }

    }

    // 输出最终结果

    int final\_result = pop();  // 获取最终计算结果

    printf("Final result: %d\n", final\_result);

}

2.5.6、SuanFu.h:

#ifndef BY\_SUANFU\_H

#define BY\_SUANFU\_H

#include "Lexical.h"

#include "GramAnalysis.h"

/\*

 G2[E]：E → T | E+T | E-T T → F | T\*F | T/F F → i | (E)

 \*/

// 栈操作函数声明

void push(int value);

int pop();

// 语法分析函数声明

void expr();

void factor();

void term();

void parse\_SF();

#endif //BY\_SUANFU\_H

2.6 实验拓展：

2.6.1、对所给算术表达式的文法 G2，完成两种以上不同的语法分析程序的设计

与实现。

答：增加了递归下降分析法：

# Recursive.c：

#include "Recursive.h"

#include <stdio.h>

#include <stdlib.h>

Token tokens[100];

int tokenIndex = 0;

int currentToken = 0;

void parse\_expr() {

    parse\_term();  // 解析第一个 T

    Token token = tokens[currentToken];  // 获取当前 token

    // 如果是加号或减号，继续解析

    while (token.type == PLUS || token.type == MINUS) {

        currentToken++;  // 处理操作符

        parse\_term();    // 解析 T

        token = tokens[currentToken];  // 获取下一个 token

    }

}

void parse\_term() {

    parse\_factor();  // 解析第一个 F

    Token token = tokens[currentToken];  // 获取当前 token

    // 如果是乘号或除号，继续解析

    while (token.type == MUL || token.type == DIV) {

        currentToken++;  // 处理操作符

        parse\_factor();  // 解析 F

        token = tokens[currentToken];  // 获取下一个 token

    }

}

void parse\_factor() {

    Token token = tokens[currentToken];  // 获取当前 token

    if (token.type == ID || token.type == INT || token.type == REAL) {

        // 如果是标识符、整数或实数，直接返回

        currentToken++;  // 处理该 token 后递增

        return;

    } else if (token.type == LP) {

        currentToken++;  // 处理左括号

        parse\_expr();  // 解析括号内的表达式

        token = tokens[currentToken];  // 获取右括号

        if (token.type != RP) {

            parseError("Expected closing parenthesis");  // 错误：缺少右括号

        }

        currentToken++;  // 处理右括号

    } else {

        parseError("Unexpected token");  // 错误：不符合因子的定义

    }

}

// 语法分析器的入口函数

int Recursive() {

    parse\_expr();  // 从表达式开始解析

    // 获取下一个 token，检查是否是 END token

    Token token = getNextToken();

    if (token.type == END) {

        printf("RIGHT!\n");

        printf("Recursive analysis success\n");

    } else {

        parseError("Unexpected token at the end");  // 错误：语法结束时出现了不期望的 token

        return -1;

    }

    return 0;

}

# Recursive.h：

#ifndef RECURSIVE\_H

#define RECURSIVE\_H

#include "SuanFu.h"

#include "Lexical.h"  // 词法分析部分头文件

#include "GramAnalysis.h"

// 语法分析函数声明

void parse\_expr();     // 表达式解析

void parse\_term();     // 项解析

void parse\_factor();   // 因子解析

// 语法分析器入口

int Recursive();       // 解析整个表达式

#endif // RECURSIVE\_H

实验三 语义分析程序设计与实现

3.1 语义处理语法结构范围及文法设计

我基于递归下降分析法完成语义分析器：

3.1.1、语义处理语法结构范围：

包含测试用例（由标识符、无符号数和+、−、\*、/、(、)构成的算术表达式）的源程序文件。

3.1.2、文法设计：

**G2[E]：**E → T | E+T | E-T T → F | T\*F | T/F F → i | (E)

3.2 语义分析程序（语义变量、语义函数程序结构等）设计

### **3.2.1、语义变量设计**

#### **1. struct QUATERNION 结构体相关成员作为语义变量**

**op**成员****：

* + ****含义及作用****：用于存储操作符信息，它直接体现了当前所执行的语义动作对应的操作符，比如 +、\*、:（赋值操作符，代码中可能用 := 表示赋值）等。在语义分析过程中，通过这个变量来明确要进行的运算或赋值等操作类型，是确定语义动作的关键元素之一。例如，当生成四元式 “GEN: (+, 3, 4, T1)” 时，op 成员的值为 +，表示正在执行加法运算的语义动作，后续在代码生成或者进一步语义处理阶段就能依据此信息进行相应的处理，比如生成对应的机器指令（如果用于编译后端）或者进行表达式求值模拟等操作。
  + ****取值范围及示例****：其取值范围涵盖了代码所支持的所有操作符，像算术运算符（+、-、\*、/ 等）、赋值运算符（:=）以及可能扩展的其他运算符等。例如在表达式 a := 3 + 4 的语义分析中，针对加法运算部分，op 会被赋值为 +，针对赋值部分则会被赋值为 :=，通过不同的值来准确表示不同的语义动作。

**arg1**和**arg2**成员****：

* + ****含义及作用****：这两个成员分别用于存储操作符对应的操作数或者表达式的相关部分，它们共同构成了操作符执行运算或操作的对象。在语义分析时，通过这两个变量来传递参与运算的元素，使得操作符能明确对哪些值进行操作，是语义表达完整的重要组成部分。比如对于乘法运算的四元式 “GEN: (\*, 3, 4, T1)”，arg1 的值为 3，arg2 的值为 4，清晰地表明了乘法操作的两个操作数，以此完整地体现乘法运算这一语义动作涉及的具体元素。
  + ****取值范围及示例****：取值范围根据输入表达式中可能出现的操作数类型而定，可能是标识符（如变量名 a、b 等）、整数常量（像 3、5 等）、临时变量名（例如 T1、T2 等通过 NewTemp 函数生成的临时存储中间结果的变量名）等。以表达式 (a + b) \* 5 为例，在生成乘法运算四元式时，若 a 和 b 经过前面的分析分别用标识符存储在 arg1 和 arg2 中参与加法运算后得到中间结果用临时变量 T1 表示，那么对于乘法运算的四元式，arg1 可能取值为 T1，arg2 取值为 5，准确反映了运算元素。

**result**成员****：

* + ****含义及作用****：用于存放操作符执行相应运算或操作后的结果，通常是一个临时变量名（由 NewTemp 函数生成）或者赋值语句中的目标标识符（如变量名）等。它体现了语义动作执行后的产出，是后续语义相关操作继续依赖的重要元素，用于传递中间结果或者最终结果到后续的语义处理环节。例如在表达式 3 + 4 的语义分析中，加法运算生成的四元式里 result 可能为 T1（假设新生成的临时变量），后续如果这个结果还要参与其他运算（如再和别的数相乘等），就可以通过 T1 这个临时变量名在其他四元式的 arg1 或 arg2 等成员中出现，实现结果的传递和继续参与语义分析的过程。
  + ****取值范围及示例****：取值范围主要是临时变量名（遵循 T[数字] 的命名格式）以及合法的标识符（变量名等）。比如对于表达式 a := 3 + 4，在赋值操作的四元式里 result 就是目标标识符 a，表示将前面加法运算的结果赋值给 a；而对于中间运算如 (3 + 4) \* 2 中加法运算部分的四元式，result 会是类似 T1 的临时变量名，用于暂存加法结果以便后续乘法运算使用。

#### **2. NXQ 变量作为语义相关索引变量**

* ****含义及作用****：它作为四元式表的索引，记录了当前生成的四元式在整个四元式序列中的位置。在语义分析过程中，通过 NXQ 的不断递增，有序地组织了四元式的生成和存储，使得各个语义动作对应的四元式能按照分析顺序依次排列，方便后续对整个表达式语义的完整呈现以及基于四元式的进一步处理（如代码生成等）。例如，每执行一次 GEN 函数生成一个新的四元式，NXQ 就自增 1，保证了四元式序列的顺序性和完整性，能准确反映表达式语义分析的逐步推进过程。
* ****取值范围及示例****：其初始值为 0，随着四元式的不断生成，取值范围从 0 开始逐步递增，直到完成整个表达式的分析，其最大值就是最终生成的四元式的数量减 1。比如对于一个简单的表达式经过分析生成了 3 个四元式，那么 NXQ 的取值就会依次为 0、1、2，对应每个四元式在序列中的索引位置。

#### **3. NXTemp 变量用于临时变量语义管理**

* ****含义及作用****：用于对临时变量进行计数和命名管理，在语义分析中，每当需要表示一个中间结果（如加法、乘法等运算产生的中间值）时，就通过 NXTemp 生成一个唯一的临时变量名（格式为 T[数字]），这个临时变量名在四元式的 result 等成员中使用，来传递和存储中间结果，确保每个中间结果都有唯一标识且便于后续引用，是处理表达式中间语义信息的关键变量。例如在复杂表达式的逐步运算过程中，不同位置生成的中间结果通过不同的临时变量名区分开，避免了语义混淆，保障了语义分析的准确性和可延续性。
* ****取值范围及示例****：初始值为 1，随着临时变量的不断生成，其值会依次递增。比如在分析表达式 (3 + 4) \* (5 + 6) 时，先计算 3 + 4 可能生成临时变量 T1（此时 NXTemp 变为 2），再计算 5 + 6 可能生成 T2（NXTemp 变为 3），后续基于这些临时变量进行乘法运算等语义动作时，就能通过明确的临时变量名准确表示各部分的语义关系。

### **3.2.2、语义函数设计**

#### **1. GEN 函数作为核心语义生成函数**

* ****功能实现****：它接收四个字符串指针参数（Op、Arg1、Arg2、Result），负责将这些参数对应的语义信息组合并生成一个四元式，存储到 pQuad 所指向的四元式表中。具体操作包括：首先检查操作符 Op 是否为空，若为空则输出错误提示并终止程序，确保语义操作的合法性；然后将 Op、Arg1、Arg2、Result 分别复制到四元式表中当前索引 NXQ 位置对应的 struct QUATERNION 结构体成员中，以此构建一个完整的四元式，同时在控制台输出 “GEN: (操作符，第一个操作数，第二个操作数，结果)” 的提示信息，方便查看四元式生成情况，最后将 NXQ 的值自增 1，为下一个四元式的生成做准备，有序地推动整个表达式语义分析过程中四元式的生成与存储。
* ****语义作用****：通过 GEN 函数，将表达式语法分析过程中识别到的各种语义动作（如加法、乘法、赋值等运算操作）转化为具体的四元式形式进行记录，实现了从语法结构到语义表示的关键转换，使得表达式的语义信息能够以一种规范、有序的方式被保存下来，为后续的代码生成、语义检查或者进一步的编译处理等提供了统一、明确的语义基础，是整个语义分析程序中连接语法和语义表达的核心桥梁。
* ****示例****：对于表达式 a := 3 + 4，在分析到加法运算时，会调用 GEN 函数传入 "+"、"3"、"4"、"T1"（假设生成的临时变量名）等参数，生成四元式 “GEN: (+, 3, 4, T1)” 并存储到四元式表中，接着分析到赋值操作时，再调用 GEN 函数传入 ":="、"T1"、""、"a" 等参数，生成赋值相关的四元式 “GEN: (:=, T1,, a)”，以此完整地通过四元式体现了表达式的语义内容。

#### **2. NewTemp 函数用于临时语义变量生成**

* ****功能实现****：利用一个静态字符数组 TempID 来生成临时变量名。通过 sprintf 函数按照格式化的方式将 “T” 和当前的 NXTemp 值组合，生成形如 T[具体数字] 的临时变量名，然后返回这个临时变量名的字符串指针，同时将 NXTemp 的值自增 1，为下一次生成临时变量做准备，确保每次生成的临时变量名都具有唯一性且符合特定的命名格式，便于在四元式中作为中间结果的标识进行语义表达。
* ****语义作用****：在语义分析中，由于表达式运算会产生大量中间结果，这些中间结果需要有合适的标识来参与后续运算以及在语义表示（四元式）中准确体现其位置和作用，NewTemp 函数就承担了为这些中间结果生成唯一标识的任务，使得语义分析能够清晰地处理和传递中间语义信息，避免了不同中间结果之间因标识不清而导致的语义混淆问题，是保障语义分析过程中临时语义变量有序管理的重要函数。
* ****示例****：在分析表达式 (3 + 4) \* 2 时，先计算 3 + 4 这部分，需要一个临时变量来存储结果，调用 NewTemp 函数就会生成 T1 作为临时变量名，后续生成的四元式中就可以用 T1 来表示这个加法运算的中间结果，如生成四元式 “GEN: (+, 3, 4, T1)”，然后这个 T1 又可以作为乘法运算的操作数参与后续语义分析，如在生成乘法运算四元式 “GEN: (\*, T1, 2, T2)” 时，清晰地体现了中间结果在整个表达式语义中的传递和作用过程。

#### **3. E 函数实现 E 产生式相关语义分析**

* ****功能实现****：按照递归下降分析法处理与非终结符 E 相关的语法结构对应的语义内容，并生成相应的四元式。具体操作如下：
  + 首先检查当前词法单元是否符合赋值语句模式（标识符后紧跟赋值符号），若是赋值语句：
    - 获取赋值语句左侧的标识符字符串作为目标变量，跳过相应词法单元后，递归调用 E 函数解析右侧的表达式，将右侧表达式分析得到的结果作为赋值的源值，调用 GEN 函数生成赋值操作的四元式（操作符为 ":="，源值和目标变量分别填入对应参数位置），以此实现赋值语句的语义分析和四元式生成，体现了将右侧表达式结果赋给左侧变量这一语义动作。
  + 如果不是赋值语句，则先调用 T 函数解析 T 相关语法结构对应的语义内容，得到的结果作为初始部分，然后进入一个循环，只要当前词法单元类型是加法运算符，就执行以下操作：
    - 跳过加法运算符词法单元，再次调用 T 函数解析后续 T 结构对应的语义内容，接着调用 NewTemp 函数生成新的临时变量名，然后调用 GEN 函数生成加法运算的四元式（操作符为 +，两个操作数分别为前后两个 T 结构分析得到的结果，结果用新生成的临时变量名表示），并更新用于记录当前结果的变量，循环处理直到没有加法运算符为止，通过不断生成加法运算相关的四元式，完整地体现了表达式中加法运算的语义以及中间结果的传递和处理过程，实现 E 产生式中加法部分的语义分析。
* ****语义作用****：从语义角度，E 函数围绕 E 产生式对应的语法结构，准确地处理了表达式中顶层的语义关系，包括赋值语义以及由加法运算连接的多个子表达式的语义组合，通过递归调用和四元式生成等操作，将这些语法元素对应的语义内容清晰地展现出来，为整个表达式语义分析构建了高层次的语义框架，后续的 T、F 等函数的语义分析在此基础上进一步细化和完善具体的语义内容。
* ****示例****：对于表达式 a := 3 + 4，E 函数先识别出赋值语句结构，解析右侧 3 + 4 的语义（通过调用 T 函数等进一步处理加法运算语义），生成加法相关四元式如 “GEN: (+, 3, 4, T1)”，再生成赋值相关四元式 “GEN: (:=, T1,, a)”，完整地体现了整个表达式从加法运算到赋值的语义过程；对于更复杂的表达式如 a := 3 + 4 + 5，会在处理加法运算部分多次循环调用相关函数生成多个加法运算四元式，准确体现多次加法组合的语义情况。

#### **4. T 函数负责 T 产生式相关语义分析**

* ****功能实现****：针对与非终结符 T 相关的语法结构进行语义分析并生成四元式，具体步骤如下：
  + 首先调用 F 函数解析 F 相关语法结构对应的语义内容，将得到的结果作为初始部分，然后进入一个循环，只要当前词法单元类型是乘法或除法运算符：
    - 根据运算符类型创建相应操作符表示形式，跳过当前操作符词法单元，接着调用 F 函数解析下一个 F 结构对应的语义内容，再调用 NewTemp 函数生成新的临时变量名，随后调用 GEN 函数生成乘法或除法运算的四元式（操作符为对应的 \* 或 /，两个操作数分别为前后两个 F 结构分析得到的结果，结果用新生成的临时变量名表示），并更新用于记录当前结果的变量，循环处理直到没有乘法或除法运算符为止，通过这样的操作顺序，将乘法、除法运算以及它们与 F 结构组合的语义通过四元式准确地表达出来，实现 T 产生式相关的语义分析。
* ****语义作用****：在整个语义分析体系中，T 函数聚焦于 T 产生式对应的语法结构，细化处理了表达式中涉及乘法和除法运算以及它们与更基础的 F 结构组合的语义关系，通过生成相应的四元式，将这一层级的语义信息明确地展现出来，为 E 函数所构建的高层次语义框架进一步填充具体的运算语义内容，与 E、F 等函数协作完成整个表达式完整的语义分析过程，尤其体现了乘法、除法运算在表达式语义中的作用和处理方式。
* ****示例****：对于表达式 3 \* 4 / 2，T 函数先调用 F 函数处理第一个操作数 3 的语义，然后遇到乘法运算符，解析下一个 F 结构 4 的语义，生成乘法运算四元式 “GEN: (\*, 3, 4, T1)”，接着遇到除法运算符，再解析 F 结构 2 的语义，生成除法运算四元式 “GEN: (/, T1, 2, T2)”，通过这些步骤准确地体现了表达式中乘法、除法运算及相关语义关系。

#### **5. F 函数处理 F 产生式相关基础语义**

* ****功能实现****：依据当前词法单元的类型进行最基础的语义处理，具体如下：
  + 如果当前词法单元类型是标识符，直接返回该标识符对应的字符串值，从语义角度表示标识符作为一个操作数参与后续运算或赋值等语义动作，其语义含义就是作为一个可被操作的变量对象。
  + 如果是整数或浮点数类型词法单元，则返回对应的数值字符串，意味着该常量值作为操作数参与语义分析中的运算等操作，体现了常量操作数在语义中的地位和作用。
  + 如果是左括号，跳过左括号词法单元后，递归调用 E 函数解析括号内表达式的语义内容，得到的结果作为括号内表达式的语义产出，检查右括号后返回该结果，用于在更高层语法结构对应的语义分析中参与运算等操作，体现了括号改变运算优先级以及作为一个子表达式整体参与语义动作的语义特点。
  + 如果遇到不合法词法单元，则构造错误消息并调用 printErrorAndExit 函数输出错误提示并退出程序，确保语义分析的输入符合规定的语法和语义规则，保障整个语义分析过程的合法性和正确性。

3.3 语义分析测试结果分析

**3.3.1 示例输入与其结果：**

当input.txt的内容是a = ( a + b ) \* ( 2 / 1 )，输出是：

语法分析及四元式生成结果如下：

解析T开始, currentToken: 2

解析T开始, currentToken: 3

解析F后，当前T1\_place: a, currentToken: 4

解析T开始, currentToken: 5

解析F后，当前T1\_place: b, currentToken: 6

GEN: (+, a, b, T1)

解析F后，当前T1\_place: T1, currentToken: 7

当前操作符为：'\*'

解析T开始, currentToken: 9

解析F后，当前T1\_place: 2, currentToken: 10

当前操作符为：'/'

解析F后，当前T2\_place: 1, currentToken: 12

GEN: (/, 2, 1, T2)

生成四元式：(/, 2, 1, T2)

解析F后，当前T2\_place: T2, currentToken: 13

GEN: (\*, T3, T3, T3)

生成四元式：(\*, T3, T3, T3)

GEN: (:=, T3, , a)

四元式生成结果：

0: (+, a, b, T1)

1: (/, 2, 1, T2)

2: (\*, T3, T3, T3)

3: (:=, T3, , a)

进程已结束，退出代码为 0

### **3.3.2、整体分析思路**

从输出的结果可以看出，程序按照递归下降的语法分析方式，结合四元式生成规则，对输入的表达式逐步进行处理，将其语法结构转化为相应的四元式来体现语义信息。下面将按照分析过程及生成的四元式顺序，详细剖析每一步所反映的语义内容以及整个表达式的语义逻辑。

### **3.3.3、逐个步骤分析**

#### **1.** “解析 T 开始，currentToken: 2

解析 T 开始，currentToken: 3  
解析 F 后，当前 T1\_place: a, currentToken: 4”

****含义及语义分析****：  
这一系列输出表明程序开始解析与非终结符 T 相关的语法结构，并进一步深入到解析 F（基础因子部分）。当解析到 F 时，遇到了标识符 a，从语义角度来看，此时将 a 作为一个操作数进行识别，意味着在后续的表达式运算中，a 会参与相应的操作，它在这里代表一个可被操作的变量，其语义角色就是参与表达式构建的基本元素之一，等待后续与其他操作数或运算符组合来完成整个表达式的语义表达。

#### **2.** “解析 T 开始，currentToken: 5

解析 F 后，当前 T1\_place: b, currentToken: 6  
GEN: (+, a, b, T1)”

****含义及语义分析****：

继续按照语法分析规则解析下一个 T 相关部分，在解析对应的 F 结构时遇到了标识符 b，同样 b 也被视作一个操作数。

随后生成了四元式 “GEN: (+, a, b, T1)”，从语义上这表示进行加法运算，操作数分别是前面识别到的 a 和 b，而结果用临时变量 T1 来存储。这一步完整地体现了加法运算这一语义动作，将两个变量相加，并通过临时变量 T1 来传递这个加法运算的中间结果，使得后续的运算可以基于这个中间结果继续进行，符合表达式中先计算 a + b 这部分加法运算的语义逻辑。

#### **3.** “解析 F 后，当前 T1\_place: T1, currentToken: 7

当前操作符为：'\*'  
解析 T 开始，currentToken: 9  
解析 F 后，当前 T1\_place: 2, currentToken: 10”

****含义及语义分析****：

前面加法运算得到的中间结果 T1 作为当前要处理的部分继续参与后续分析，这体现了语义信息的传递，即前一步加法运算的结果成为了后续乘法运算的操作数之一。

接着遇到乘法运算符 \*，表明要开始处理乘法运算相关语义了。然后又开始解析新的 T 结构，在解析对应的 F 部分时遇到了整数常量 2，这里 2 作为一个操作数被识别，意味着它将参与后续的乘法运算，体现了常量作为操作数参与表达式语义构建的情况。

#### **4.** “当前操作符为：'/'

解析 F 后，当前 T2\_place: 1, currentToken: 12  
GEN: (/, 2, 1, T2)  
生成四元式：(/, 2, 1, T2)”

****含义及语义分析****：

在处理乘法运算的另一个操作数相关的 T 结构时，遇到了除法运算符 /，于是先对除法运算进行处理。在解析对应的 F 结构后得到操作数 1，接着生成了四元式 “GEN: (/, 2, 1, T2)”，这从语义上表示进行除法运算，操作数是 2 和 1，结果用临时变量 T2 存储，体现了先计算 2 / 1 这部分除法运算的语义，并且通过临时变量 T2 来保存该中间结果，以便其能作为乘法运算的另一个完整操作数参与后续运算，符合表达式中先计算括号内除法运算的语义要求。

#### **5.** “解析 F 后，当前 T2\_place: T2, currentToken: 13

GEN: (, T3, T3, T3)  
生成四元式：(, T3, T3, T3)”

****含义及语义分析****：  
这里出现了一些可能不太准确的地方（按照正常逻辑，乘法四元式的操作数应该是前面生成的 T1 和 T2，此处代码中操作数都写成了 T3，可能是代码存在小失误）。正确语义上应该是将前面加法运算得到的 T1（即 a + b 的结果）和除法运算得到的 T2（即 2 / 1 的结果）作为乘法运算的两个操作数，进行乘法运算，结果用一个新的临时变量（假设正确表示应该类似 T3 等）来存储，生成类似 “GEN: (\*, T1, T2, T3)” 的四元式，以此体现乘法运算这一语义动作，将两个子表达式的结果相乘，完成 (a + b) \* (2 / 1) 中乘法运算的语义表达。

#### **6.** “GEN: (:=, T3,, a)

四元式生成结果：  
0: (+, a, b, T1)  
1: (/, 2, 1, T2)  
2: (\*, T1, T2, T3) （假设修正后的正确形式）  
3: (:=, T3,, a)”

****含义及语义分析****：

最后生成的四元式 “GEN: (:=, T3,, a)” 表示赋值语义动作，即将前面乘法运算得到的结果（存储在临时变量 T3 中）赋值给标识符 a，完整地体现了整个表达式 a = ( a + b ) \* ( 2 / 1 ) 的最终语义，即先计算括号内的加法和除法运算，然后将它们相乘的结果再赋值给变量 a，通过这一系列四元式准确地展示了表达式从运算到赋值的完整语义逻辑。

从整体四元式序列来看，0 号四元式体现加法运算语义，1 号四元式体现除法运算语义，2 号四元式体现乘法运算语义（修正后），3 号四元式体现赋值语义，它们按照顺序依次表示了表达式中各个子运算以及最终赋值的语义关系，将输入的表达式从语法结构转化为清晰的、可用于后续代码生成或者语义检查等操作的语义表示形式。

### **3.3.4、结果验证与总结**

综合来看，尽管在乘法四元式生成部分可能存在代码表示上的小问题（操作数重复写成 T3），但整体上程序对于输入的表达式 “a = ( a + b) \* ( 2 / 1 )” 的语义分析及四元式生成基本符合表达式的语义逻辑，能够按照正确的运算顺序（先括号内的加法和除法，再乘法，最后赋值）通过四元式来展现各部分的语义动作以及它们之间的关联，反映了程序在语法分析结合语义表达方面的功能实现情况，也为进一步基于这些语义信息进行编译后端处理（如生成目标代码等）提供了必要的基础。如果要完善该程序，建议检查并修正乘法四元式生成部分操作数的表示问题，确保四元式能更准确无误地体现语义内容。

3.4 实验代码

3.5.1、lexical.c:同1.5.1部分代码；

3.5.2、lexical.h:同1.5.2部分代码；

3.5.3、GramAnalysis.c（做了修改）:

#include "GramAnalysis.h"

Token tokens[100];    // 词法分析使用的 tokens 数组

void parseError(const char \*message) {

    printf("ERROR: %s at token %d (%s,%s),%d:%d\n", message, currentToken, token\_names[tokens[currentToken-1].type],tokens[currentToken-1].value,tokens[currentToken-1].row, tokens[currentToken-1].col);

    exit(1);

}

Token getNextToken() {

    return tokens[currentToken++];

}

void retractToken() {

    currentToken--;

}

void readTokens(const char \*filename) {

    FILE \*file = fopen(filename, "r");

    if (!file) {

        printf("Error: Cannot open token file\n");

        exit(1);

    }

    char typeStr[MAX\_TOKEN\_LENGTH];

    char value[MAX\_TOKEN\_LENGTH];

    int row = 1;

    int col = 1;

    while (1) {

        fscanf(file, "(%[^,],%[^)])", typeStr, value);

        fgetc(file); // 读取并丢弃右括号

        if (strcmp(typeStr, "ENTER") == 0) {

            row++;

            col = 1;

            continue;

        }

        for (int i = 0; i < TOKEN\_NAMES\_SIZE; i++) {

            if (strcmp(typeStr, token\_names[i]) == 0) {

                tokens[tokenIndex].type = i;

                break;

            }

        }

        strcpy(tokens[tokenIndex].value, value);

        tokens[tokenIndex].row = row;

        tokens[tokenIndex].col = col;

        col ++;

        // 将字符串类型转换为枚举类型

        printf("typeStr: %s\n", typeStr);

        printf("value: %s\n", tokens[tokenIndex].value);

        if (strcmp(typeStr, "RP") == 0) {

            fgetc(file);

        }

        if(strcmp(typeStr, "END") == 0) {

            break;

        }

        tokenIndex++;

        if (tokenIndex >= 100) {

            printf("Error: Too many tokens\n");

            exit(1);

        }

    }

    tokens[tokenIndex].type = END;

    fclose(file);

}

// 实现match函数

void match(int expectedType) {

    printf("Matching token: expected %d, got %d (%s)\n", expectedType, tokens[currentToken].type, tokens[currentToken].value);  // 调试输出

    if (tokens[currentToken].type == expectedType) {

        currentToken++;

    } else {

        parseError("Unexpected token");

    }

}

3.5.4、GramAnalysis.h（做了修改）:

#ifndef BY\_GRAMANALYSIS\_H

#define BY\_GRAMANALYSIS\_H

#include "Lexical.h"

// 定义 Token 结构体

typedef struct {

    TokenType type;  // 当前 token 的类型

    char value[MAX\_TOKEN\_LENGTH];  // 当前 token 的值（即词文）

    int row;  // 当前 token 出现的行号

    int col;  // 当前 token 出现的列号

} Token;

// 存放词法分析得到的 token 数组

extern Token tokens[100];

extern int tokenIndex;  // 当前 token 的索引

extern int currentToken;  // 当前 token 的类型

// 函数声明

void readTokens(const char \*filename);  // 从文件读取所有的 tokens

Token getNextToken();  // 获取下一个 token

void retractToken();  // 回退上一个 token

void parseError(const char \*message);  // 语法错误处理

void match(int expectedType);  // 匹配期望的 token 类型

#endif //BY\_GRAMANALYSIS\_H

# 3.5.5、Recursive.c（做了修改）:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "Lexical.h"

#include "GramAnalysis.h"

// 四元式数据结构

struct QUATERNION {

    char op[MAX\_TOKEN\_LENGTH];

    char arg1[MAX\_TOKEN\_LENGTH];

    char arg2[MAX\_TOKEN\_LENGTH];

    char result[MAX\_TOKEN\_LENGTH];

};

char \*E(void);

char \*T(void);

char \*F(void);

// 全局变量

struct QUATERNION \*pQuad;  // 四元式表

int NXQ = 0;  // 四元式索引

int NXTemp = 1;  // 临时变量计数

// 打印错误信息并退出程序的通用函数

void printErrorAndExit(const char\* errorMsg) {

    printf("Error: %s\n", errorMsg);

    exit(1);

}

// 用于生成四元式的函数

void GEN(char \*Op, char \*Arg1, char \*Arg2, char \*Result) {

    if (Op == NULL || strlen(Op) == 0) {

        printf("Error: 操作符为空\n");

        exit(1);  // 强制退出，提示调试信息

    }

    strcpy(pQuad[NXQ].op, Op);

    strcpy(pQuad[NXQ].arg1, Arg1);

    strcpy(pQuad[NXQ].arg2, Arg2);

    strcpy(pQuad[NXQ].result, Result);

    printf("GEN: (%s, %s, %s, %s)\n", Op, Arg1, Arg2, Result);

    NXQ++;  // 更新四元式索引

}

// 生成新临时变量

char \*NewTemp(void) {

    static char TempID[10];  // 使用静态数组，避免动态内存分配

    sprintf(TempID, "T%d", NXTemp++);

    return TempID;

}

// 递归下降语法分析

char \*E(void) {

    char \*E1\_place, \*E2\_place, \*Temp\_place;

    // 检测赋值语句（:=）

    if (tokens[currentToken].type == ID && tokens[currentToken + 1].type == IS) {

        char \*ID\_value = tokens[currentToken].value;  // 赋值语句左侧

        currentToken += 2;  // 跳过 ID 和 := 符号

        E1\_place = E();  // 解析右侧的表达式

        // 生成四元式

        GEN(":=", E1\_place, "", ID\_value);

        return ID\_value;  // 返回赋值后的标识符

    }

    E1\_place = T();  // 解析T

    // 处理加法（+）运算符

    while (tokens[currentToken].type == PLUS) {

        char op[2] = "+";  // '+' 操作符

        currentToken++;  // 跳过操作符

        E2\_place = T();  // 解析T

        Temp\_place = NewTemp();  // 生成新的临时变量

        GEN(op, E1\_place, E2\_place, Temp\_place);  // 生成四元式

        E1\_place = Temp\_place;  // 更新E1\_place为新的临时变量

    }

    return E1\_place;  // 返回最终的计算结果

}

char \*T(void) {

    char \*T1\_place, \*T2\_place, \*Temp\_place;

    printf("解析T开始, currentToken: %d\n", currentToken);

    T1\_place = F();  // 解析F

    printf("解析F后，当前T1\_place: %s, currentToken: %d\n", T1\_place, currentToken);

    // 处理乘法（\*）和除法（/）运算符

    while (tokens[currentToken].type == MUL || tokens[currentToken].type == DIV) {

        char op[2] = {0};

        if (tokens[currentToken].type == DIV) {

            op[0] = '/';

        } else if (tokens[currentToken].type == MUL) {

            op[0] = '\*';

        }

        op[1] = '\0';

        printf("当前操作符为：'%s'\n", op);

        currentToken++;

        T2\_place = F();

        printf("解析F后，当前T2\_place: %s, currentToken: %d\n", T2\_place, currentToken);

        Temp\_place = NewTemp();

        GEN(op, T1\_place, T2\_place, Temp\_place);

        printf("生成四元式：(%s, %s, %s, %s)\n", op, T1\_place, T2\_place, Temp\_place);

        T1\_place = Temp\_place;

    }

    return T1\_place;  // 返回最终结果

}

char \*F(void) {

    char \*place;

    // 处理标识符（ID）

    if (tokens[currentToken].type == ID) {

        char \*id\_value = tokens[currentToken].value;

        currentToken++;  // 跳过 ID

        return id\_value;  // 返回标识符的值

    }

        // 处理整数（INT）

    else if (tokens[currentToken].type == INT) {

        char \*int\_value = tokens[currentToken].value;

        currentToken++;  // 跳过 INT

        return int\_value;  // 返回整数的值

    }

        // 处理浮点数（REAL）

    else if (tokens[currentToken].type == REAL) {

        char \*real\_value = tokens[currentToken].value;

        currentToken++;  // 跳过 REAL

        return real\_value;  // 返回浮点数的值

    }

        // 处理括号表达式（LP 和 RP）

    else if (tokens[currentToken].type == LP) {

        currentToken++;  // 跳过 '('

        place = E();  // 递归调用 E 解析表达式

        if (tokens[currentToken].type == RP) {

            currentToken++;  // 跳过 ')'

            return place;

        } else {

            printf("Error: Missing closing parenthesis ')'\n");

            exit(1);  // 如果没有闭括号，报错并退出

        }

    } else {

        char errorMsg[50];

        sprintf(errorMsg, "Invalid character: '%s'", tokens[currentToken].value);

        printErrorAndExit(errorMsg);

        exit(1);

    }

}

// 语法分析器的入口函数

void Recursive(void) {

    currentToken = 0;  // 初始化当前token索引

    pQuad = (struct QUATERNION \*)malloc(100 \* sizeof(struct QUATERNION));  // 分配四元式表空间

    char \*result = E();  // 调用递归分析表达式

    printf("四元式生成结果：\n");

    for (int i = 0; i < NXQ; i++) {

        printf("%d: (%s, %s, %s, %s)\n", i, pQuad[i].op, pQuad[i].arg1, pQuad[i].arg2, pQuad[i].result);

    }

    free(pQuad);  // 释放四元式表空间

}

# 3.5.6、Recursive.h（做了修改）:

#ifndef RECURSIVE\_H

#define RECURSIVE\_H

#include "Lexical.h"  // 词法分析部分头文件

#include "GramAnalysis.h"

#include "Lexical.h"

// 四元式结构体

#define MAXLENGTH 100

struct QUATERNION {

    char op[MAXLENGTH];    // 操作符

    char arg1[MAXLENGTH];  // 第一个操作数

    char arg2[MAXLENGTH];  // 第二个操作数

    char result[MAXLENGTH];// 运算结果

};

extern struct QUATERNION \*pQuad;  // 四元式数组

extern int NXQ;                   // 四元式数组索引

extern int NXTemp;                // 临时变量计数器

// 语法分析函数声明

void parse\_expr();     // 表达式解析

void parse\_term();     // 项解析

void parse\_factor();   // 因子解析

// 语法分析器入口

int Recursive();       // 解析整个表达式

// 用于四元式生成的外部函数

void GEN(char \*Op, char \*Arg1, char \*Arg2, char \*Result);

char \*NewTemp(void);

extern struct QUATERNION \*pQuad; // 四元式表

extern int NXQ;  // 四元式计数器

#endif // RECURSIVE\_H

四、实验总结

### **一、实验完成情况自我评价**

#### **（一）优点**

* ****功能完整性****：
  + 成功构建了词法分析器，能够对输入的源程序进行有效的词法扫描，准确识别出各类词法单元，包括标识符、常量（整数、浮点数等）、运算符（算术、赋值等）以及括号等，为后续的语法和语义分析提供了可靠的输入基础。例如，在处理复杂表达式如 “a = ( 3.5 + 2 \* (b - 1)) / 5” 时，能正确地将其分解为一个个独立的词法单元，如标识符 “a”、“b”，常量 “3.5”、“2”、“1”、“5”，运算符 “=”、“+”、“\*”、“-”、“/” 以及括号 “(”、“)”。
  + 语法分析器基于递归下降分析法实现，能够对给定的文法规则进行有效的解析，正确处理表达式中的各种语法结构，如赋值语句、算术表达式（包含加法、减法、乘法、除法以及括号嵌套等）。从对测试用例的分析结果来看，对于合法的表达式和语句，能够顺利构建出相应的语法树（通过四元式的生成过程间接体现了语法结构的解析），如对于 “a := 3 + 4 \* (5 - 2)” 这样的表达式，能按照语法规则正确地分析出各个子表达式的层次关系，并生成合理的四元式序列来表示其语义。
  + 语义分析部分借助四元式有效地表达了表达式的语义信息，将语法结构转化为可用于后续代码生成或进一步语义处理的中间表示形式。在处理不同类型的表达式时，如包含变量赋值、算术运算以及括号改变运算优先级的情况，都能通过四元式准确地反映出运算的顺序和操作数的关系，例如对于 “(a + b) \* (c /d)” 能生成如 “(+, a, b, T1)”、“(/, c, d, T2)”、“(\*, T1, T2, T3)” 的四元式序列，清晰地展示了先加法、再除法、最后乘法的语义逻辑以及中间结果的传递。
* ****代码结构与可读性****：
  + 代码结构较为清晰，各个功能模块（词法分析、语法分析、语义分析相关函数）划分明确，职责单一。例如，词法分析部分的函数专注于从输入字符串中识别词法单元，语法分析的各个非终结符对应的函数（E、T、F 等）按照递归下降的规则分别处理各自的语法结构，语义分析相关函数（GEN、NewTemp 等）主要负责四元式的生成与临时变量管理，这种模块化的设计使得代码易于理解和维护，降低了代码的复杂性，方便后续的功能扩展和修改。
  + 变量和函数的命名较为规范，能够反映其功能和用途，如 tokens 数组用于存储词法单元序列，currentToken 用于标记当前处理的词法单元位置，GEN 函数用于生成四元式等，这有助于提高代码的可读性，使得其他开发者能够快速理解代码的意图和逻辑。

#### **（二）不足**

* ****错误处理不够完善****：
  + 在词法分析过程中，对于一些非法字符的错误处理方式较为简单，只是输出错误信息并直接终止程序。例如，当输入中包含一些不被识别的特殊字符（如 “@#$” 等）时，程序立即停止运行，没有提供更多的错误恢复机制或详细的错误定位信息，这对于用户来说不够友好，难以快速定位和解决问题，在实际应用场景中可能导致较差的用户体验。
  + 语法分析中，对于语法错误的处理同样比较粗糙，遇到不符合文法规则的情况（如表达式中运算符缺失、括号不匹配等）也是直接报错并退出程序，没有尝试进行错误修复或者提供更详细的错误提示信息，例如指出错误发生的大概位置（行号、列号等）或者可能的错误原因（是缺少运算符还是括号错误等），这不利于开发者调试复杂的程序代码。
* ****代码效率与性能优化****：
  + 目前的算法在处理大规模复杂表达式时可能存在性能瓶颈。例如，递归下降分析法在处理深度嵌套的表达式时，由于递归调用的开销，可能导致栈空间的过度使用和运行效率的降低。在一些极端情况下，如处理一个包含大量嵌套括号和复杂运算的超长表达式时，可能会出现栈溢出错误或者运行时间过长的问题，影响程序的实用性和可靠性。
  + 内存管理方面，虽然在部分地方（如四元式表的分配）使用了动态内存分配，但对于内存的使用效率和释放时机的控制还不够精细。例如，在四元式生成过程中，如果表达式的复杂度超过了预先分配的四元式表大小，可能会导致内存分配不足的问题；而在程序结束时，仅简单地释放了四元式表的内存，对于其他可能分配的临时内存（如词法分析过程中可能存在的一些临时缓冲区等）没有进行全面的清理和释放，可能会造成内存泄漏，影响程序的长期稳定性和性能。
* ****测试覆盖度有限****：
  + 测试用例的设计虽然覆盖了一些常见的表达式和语句类型，但还不够全面。例如，对于一些边界情况的测试不足，像处理空表达式、只包含单个操作数或运算符的表达式等情况没有进行充分的测试；对于一些特殊的语法结构组合和语义情况（如连续多个赋值语句、表达式中存在多种数据类型混合运算且有隐式类型转换需求等）的测试也不够完善，这可能导致程序中存在一些未被发现的潜在缺陷，在实际应用中可能出现错误的结果或异常行为。

### **二、改进想法**

#### **（一）错误处理改进**

* ****词法分析错误处理****：
  + 当遇到非法字符时，不仅输出错误信息，还可以尝试跳过该非法字符，继续扫描后续的输入内容，同时记录错误发生的位置信息（如字符在输入字符串中的索引位置），并在扫描结束后提供一个详细的错误报告，列出所有遇到的非法字符及其位置，以便用户能够快速定位和修改错误源。例如，可以使用一个数据结构（如链表或数组）来存储错误信息，每个错误信息包含错误字符和其位置，最后统一输出。
  + 对于一些可能的词法单元识别错误（如标识符命名不符合规范但不是完全非法字符），可以提供更详细的错误提示，说明正确的标识符命名规则，帮助用户更好地理解和修改代码。
* ****语法分析错误处理****：
  + 当检测到语法错误时，尝试进行错误恢复操作。例如，采用 panic mode 错误恢复策略，当遇到错误时，跳过一些输入符号，直到找到一个可以继续语法分析的同步符号（如分号、括号等），然后继续分析后面的内容。同时，在错误报告中详细说明错误类型（如缺少运算符、括号不匹配等）、错误发生的位置（如所在的表达式位置或行号、列号，如果有相关信息的话）以及可能的错误原因和修复建议，这样可以大大提高开发者调试程序的效率，减少错误排查的时间。
  + 可以建立一个语法错误处理库或模块，将常见的语法错误类型及其处理方式进行封装，使得错误处理代码更加模块化和可维护，方便在不同的语法分析场景中复用。

#### **（二）性能优化**

* ****算法优化****：
  + 考虑采用更高效的语法分析算法，如自底向上的算符优先分析法或 LR 分析法，来替代当前的递归下降分析法，以减少递归调用带来的栈空间开销和提高分析效率，特别是对于大规模复杂表达式的处理。这些算法在处理语法结构时通常具有更好的性能和鲁棒性，能够更有效地处理各种复杂的语法情况，并且可以通过构建语法分析表等方式提前确定分析步骤，减少运行时的决策开销。
  + 对于表达式求值部分，可以采用优化的计算策略，如先计算常量表达式部分，将其结果缓存起来，减少不必要的重复计算；或者采用并行计算技术（如果适用的话），对于一些相互独立的子表达式，可以并行地进行计算，提高整体的计算速度，尤其是在多核处理器环境下，可以充分利用硬件资源提升程序性能。
* ****内存管理优化****：
  + 优化四元式表的内存分配策略，可以采用动态扩容的方式，当发现四元式表空间不足时，自动增加分配的内存大小，避免因预先分配空间不足导致的错误。例如，可以按照一定的比例（如每次增加当前大小的 50%）来扩容，同时记录扩容的次数和总内存使用量，以便进行性能监控和优化。
  + 加强内存泄漏检测和预防机制，在程序的关键节点（如函数退出、模块结束等）增加内存检查代码，确保所有动态分配的内存都被正确释放。可以使用一些内存检测工具（如 Valgrind 等）来辅助检测内存泄漏问题，并根据检测结果进行针对性的修复，保证程序在长期运行过程中的稳定性和性能。

#### **（三）测试覆盖度提升**

* ****边界情况测试****：
  + 增加对空表达式、单个操作数或运算符表达式的测试用例，确保程序在这些特殊情况下能够正确处理，不会出现异常行为或错误结果。例如，测试输入为空字符串时程序的反应，以及输入只有一个数字或一个变量时的处理结果是否符合预期。
  + 对表达式的边界值进行测试，如整数常量的最大值、最小值，浮点数的精度边界等情况。例如，测试表达式中使用 INT\_MAX 和 INT\_MIN 进行运算时是否正确处理溢出情况，以及浮点数在接近其精度极限时的运算结果是否准确。
* ****特殊语法和语义情况测试****：
  + 设计更多包含特殊语法结构组合的测试用例，如连续多个赋值语句的嵌套、复杂的括号嵌套与不同优先级运算符混合使用的表达式等，确保程序在处理这些复杂情况时的正确性。例如，测试 “a := b := 3 + 4 \* (5 - 2)” 这样的连续赋值表达式，以及 “((a + b) \* (c /d)) / ((e - f) \* (g + h))” 这样深度嵌套括号与多种运算符组合的表达式。
  + 针对不同数据类型混合运算且可能涉及隐式类型转换的情况进行测试，检查程序是否按照预期的类型转换规则进行处理，例如测试整数与浮点数相加、相乘时的结果是否正确，以及在赋值语句中不同类型变量之间赋值时是否进行了正确的类型转换等。
  + 进行随机测试，使用随机生成的表达式和语句作为测试输入，增加测试的多样性和覆盖度，以发现一些可能在常规测试中遗漏的潜在问题。可以设置不同的随机生成参数，如表达式的长度、运算符的种类和数量、操作数的类型等，来生成大量不同类型的测试用例，并自动执行测试和收集结果，以便更全面地评估程序的正确性和稳定性。

通过以上对实验完成情况的自我评价和改进想法的提出，可以有针对性地对词法、语法、语义分析程序进行进一步的完善和优化，提高程序的质量、性能和可靠性，使其更符合实际应用的需求。