设计任务名称		SeuLex&SeuYacc		
完成时间		2020…	验收时间	
本组成员情况				
学　号	姓　名	承　担　的　任　务		成　绩
xxx	xxx	Lex：消除复杂语法的正则式转 NFA、NFA 转 DFA、DFA 最小化，生成词法分析器，编写总控程序。 Yacc:生成 LR(1) DFA，DFA 生成分析表，编写总控程序。		
xxx	xxx	Lex:对正则式初步编译，进行代换消除复杂语法。 Yacc：对 LR(1) DFA 合并同心集，生成 LALR。测试验证程序正确性。		
xxx	xxx	Lex:解析词法规则文件。 Yacc:解析语法规则文件，并生成产生式和终结符非终结符表 。		

1 编译对象与编译功能

1.1 编译对象

详见本报告目录下基于 c99.l 改写的词法规则文件文件 littlec.l 以及基于 c99.y（基本未改动）的语法规则文件 littlec.y

1.2 编译功能

1．词法分析器 SeuLex
   1) 读取词法规则文件　yylParser.h
   2) 对正则式进行代换，消除复杂语法　RegFilter.h
   3) 正则转 NFA　NFA.h，NFA.cpp
   4) NFA 转 DFA　DFA.h，DEA.cpp
   5) DFA 最小化　RegToDFA.h
   6) 生成词法分析器代码　lexSource.h, lexSource.cpp, SwitchBuilder.h
   7) 调用上述模块实现把词法规则文件转换成词法分析程序　main.cpp

2．语法分析器 SeuYacc
   1) 读取读取语法规则文件并生成产生式和终结符非终结符表 yaccParser.h
   2) 存储产生式和语法符号表，并提供生成和访问的接口 yaccProducer.h, yaccProducer.cpp
   3) 生成 LR(1) DFA，合并同心集转化为 LALR DFA yaccUtil.h, yaccDFA.cpp
   4) 根据 DFA 生成分析表 LRTable.h
   5) 代码生成模块 yyWrite.h, yaccSource.h, yaccSource.cpp
   6) 总控程序 main.cpp

2. 主要特色

1. 功能较完整的正则支持

对规则做了一些使分析程序便于设计的修改，但是对表达能力没有根本影响。

支持 ASCII 码 9(‘\t’)~126(‘~’)中的字符作为源代码文本，未考虑兼容其他类型字符（多字节字符会作单字节字符处理）

2. 词法语法全集支持

可接受适应规则修改的 c99.l 和原样的 c99.y 作为输入，即可对 C 语言全集（仅考虑常规字符集）进行词法和语法分析

3. 出错处理

词法分析结果带单词的行号列号定位；语法分析也能定位每个语法单元；报错时给出出错位置，语法报错还会建议可能需要的终结符类型。

简单的恐慌模式错误恢复，重复抛弃引起错误的符号，直到下个符号能让分析继续进行。

具体规则文件语法：

(1) Lex 段划分：用%%划分为定义段、规则段和程序段，不可以缺少%%号，此外还有 %{和%}包裹的一个程序定义段，我们允许此段在定义段和规则段的任意行之间插入；%%、%{、%}号都必须独占一行并且前后无空格，文件里也不建议包括有带空格的空行；
(2) lex 定义段：支持把不含无空格和转义字符的任意字符串作标志，允许任意复杂的定义和后面的定义再引用前面已作的定义，只要用定义的目标替换进对应引用的正则式

中成为一个合法的正则式就可以；

(3) lex 规则段：

1) 事实上允许源代码包含的字符集仅为 ASCII9(‘\t’)-126(‘~’)正则式支持的正则语法符号包括{}[]()|+?*.（都与一般的正则式规则一致），正则式不能含有空格，如要表示空格，用我们定义的转义方式‘\p’表示；其他非输入字符的转义还包括\r \n \t \v \f;
2) 双引号必须完整地包括整个正则式才起到说明整个式子是原始字符串的作用，除此之外，局部出现的双引号都当作普通字符；
3) 正则式语法符号{}[]()|+?*.以及\需要在前面加\转义，此外为便于使用，我们允许在任意字符前加\来表明是一个原始字符而不引起错误，上文提到的\n\p 等情形除外
4) []中第一个字符为^时，不是表示它本身而是表示反转，由[]中字符任意一个的含义变为[]中字符任意一个的含义；^只有出现在开头才有此含义，因此该字符不需要被转义；[]中仅有-和\需要被转义，x-y 表示一个从 x 字符到 y 字符的范围（ASCII 编码连续）
5) 每个正则式后用至少一个空格或 tab 分开，然后书写相应执行动作的代码，须以{}包裹，每个正则式及其匹配必须占据且占据一行；先定义的规则优先级高于后定义的。
6) 要表示匹配了一个有意义的单词，应该返回一个标记，是一个整数（通常用 yacc 生成的 y.tab.h 做宏定义）；

(4) Lex 程序段：用户定义的程序段都将原样放入生成代码，我们提供的可引用的变量和函数包括 yytext,yyleng,yylineno,column,input(),unput()，还有宏 ECHO 和 error(str),此外,必须定义一个 yywrap(),由于还不提供多个文件的联合分析功能,这个函数需要始终返回 1;

(5) Yacc 段划分：用%%划分为定义段、规则段和程序段，我们的 yacc 还不支持用户自定义任何程序，但是规则文件仍然至少要有第二个%%来指示结尾；

(6) Yacc 定义段：仅支持%token 和%start 定义，并且必须用%start 指示一个唯一的文法开始符号，%token 终结符要与 lex 规则文件中的返回标记配合；

(7) Yacc 规则段：依次描述所有产生式——对于每个产生式，第一行只包括左部，第二行以冒号开始，然后用空格分开产生式右部每一项，之后每行用|开始，列出其他相同左部的产生式的右部，最后用单独一行;来结束；可以把符号规约为 ε，只要右部空着不写即可；支持在规则段隐含使用单个字符的终结符，用单引号包括即可，但是这个字符不能是\n 等带转义符号的字符；由于我们的定义段不支持定义结合性与优先级等，因此相应产生式的写法要采用以不同语法符号来区分优先级的方法；冲突处理是移进优先于规约、先声明的规约优先于后声明的，因此 if-else 等经典结构可以直接表达

3 概要设计与详细设计

3.1 概要设计

1. SeuLex 的设计

yylParser：读取词法规则文件，提供一个读取指定规则文件并构造存储文件中规则内容的对象的构造函数；

RegFilter：对正则式进行代换，消除复杂语法，提供把规则定义和复杂形式正则式转换成只包含简单规则的正则式的函数；

NFA、DFA、RegToDFA：包含简化后的正则转 NFA、NFA 转 DFA、DFA 最小化的紧凑流程，以及一系列对外提供的封装后的接口，允许在按顺序输入正则式后，生成对应的最小化 DFA；

lexSource、switchBuilder：根据最小化 DFA 以及预先设定的代码和用户提供的代码，生成词法分析器代码；

main.cpp 引用上述模块，完成从读取词法规则文件，到解析出 DFA，再到代码生成输出到文件 lex.yy.c 的步骤。

2. SeuYacc 的设计

yaccParser ：读取语法规则文件，生成产生式和终结符非终结符表

yaccProducer：存储产生式和语法符号表，并提供生成和访问的接口；此外也提供导出 y.tab.h 的函数；

yaccDFA：紧密集成了基于当前的产生式生成 LR(1) DFA 对象的步骤；LR(1) DFA 对象还可以进一步调用其方法，合并同心集转化为 LALR DFA；

LRTable：输入 LR(1)/LALR DFA 对象，生成分析表对象；还包括把分析表生成为代码的成员函数；

yyWrite：生成语法分析器所需产生式等语法信息的初始化过程；

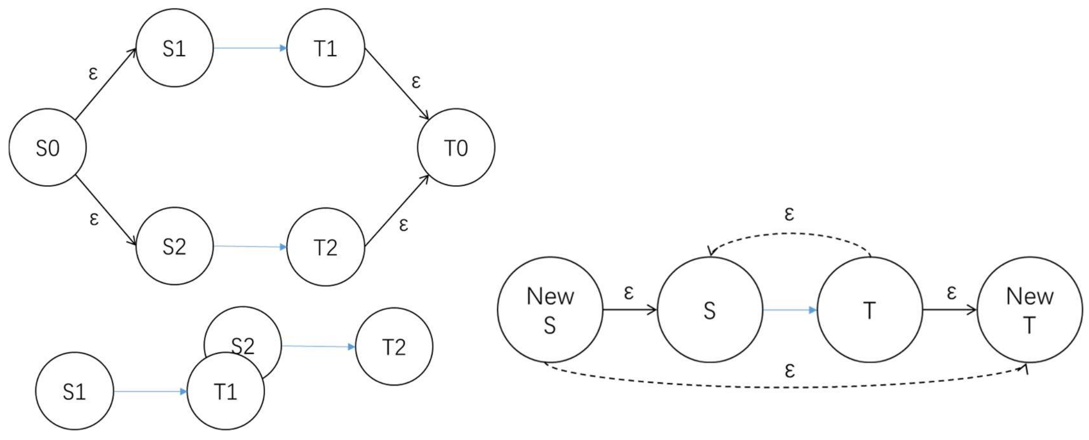
ySource：生成语法分析器包含的代码；

main.cpp 引用上述模块，完成从读取语法规则文件，到生成相应语法符号并导出 y.tab.h，到生成 DFA，再到生成分析表，最后输出代码文件 y.tab.c 的步骤。

3.2 详细设计

1. yylParser 读取词法文件：每次读取文件的一行，根据对读取到%%, %{和%}的记录可以很容易确知当前在哪一段，然后分别采取不同的解析：对于程序段我们简单把字符串保存下来；而对于定义段和规则段，由于采用了禁止空格出现在定义串/正则式中的做法，我们使用一个直接按空白字符分割的函数 lnToPair()就可以把分割出的第一个串与后面的内容分离开来。
2. RegFilter 提供了一系列 string 映射到 string 的函数，可以逐步地把我们的正则式中的复杂规则除去，替换成只包含简单规则的字符串。首先，我们对所有规则定义的目标字符串和正则式应用 spcharFilter()，这个函数将从头到尾扫描其中的非输入字符的转义形式并转换成实际字符，例如字符串成分"\\n" 两个字符转化为一个字符'\n'，此外还把我们之后要用到的转换符号{}[]用特殊的允许字符集以外的字符替换，方便和转义字符区分开来；之后，我们应用 replaceBracePairs()，把正则式中的{}定义引用以字符串替换的形式代换（此前大括号已被 spcharFilter 代换为特殊符号，因此可用一个简单的检索函数来实现这个过程），由于定义可以嵌套，替换循环进行直到没有可代换的内容；之后我们再应用 setDots()，把字符串中不是在[]里的非转义符号的.用字符串[\t-~]替换（[]应该为 spcharFilter 提到的特殊字符，\t 和~分别是我们容许解析的字符的开始和结束），这也可以通过从左到右扫描实现；最后，应用 replaceBracketPairs()，它负责把[]转换成用()和|符号表达的形式，这可以通过一边扫描并设置一个符号包含表来实现，转换时还要对基本正则式中仍然要用的语法符号(|)+?*以及\前面加\来转义。除此之外，用 quoteFilter 特殊处理引号包裹的串，我们直接除去引号，然后对其中(|)+?*\符号前面加上\转义。
3. 正则转 NFA：采用 Thompson 算法，并且只用一遍从左到右扫描的同时就隐含分析，构造出 NFA。具体来说，转义符号可在从左到右扫描时自然处理，我们主要用到的结构包括运算符栈和结点栈，并初始化结点栈第一个元素为只有一个 NFA 结点的空

NFA——结点栈每个元素只需要只有对应 NFA 的起点和终点的指针就可以在元素间执行各种运算；扫描的过程中遇到单目运算符直接更新结点栈顶，双目运算符则把栈顶相同和更高优先级的运算符处理完后，压入运算符栈，然后把一个新的空 NFA 入栈；遇到非运算符时，实际上隐含一个连接运算，先处理这个连接运算符（我们用’\0’表示），入栈连接运算符后，再把一个只有两个结点的代表单个字符的 NFA 入栈；遇到右括号时，则弹出运算符栈并计算直到栈顶为左括号，然后用一个连接运算符代替左括号。正则串结束时的处理类似右括号，最后在清空运算符栈时，结点栈剩余的最后一个元素就是该正则式所对应 NFA。由于上述 NFA 生成过程是基于递归的，因此 NFA 只有下图几种基本形状的构造组合，每个结点最多两条出边，由此我们的 NFA 是用固定数据结构——编号+两条边的符号与指向——实现的。



此外，还要把多个正则式分别生成 NFA 合并，这相当于|运算，这里我们也递归地把第 2 个以后的 NFA 合并到前面去，但是要分别标记不同的终点来区分匹配到不同的产生式，这是通过一个数组 int FA_flag[]标记每个结点是否为终态以及终态编号来实现的。

4. NFA 转 DFA：int NFAtoDFA(NFANode * startNode);实现把 NFA 数据 zhuand DFA 结构中，具体方法为模拟子集构造法：使用结构 set<int> state_set[]和 map<char, set<int> > state_table[]来存储当前的转换表，然后把起点的 ε-closure 作为 0 号状态即第一个 DFA 结点，进行拓展，拓展出的状态均求 ε-closure 后跟已有状态对比确认是否为新状态，新状态将放到表尾等待之后被拓展。最后所有状态按顺序转换为 DFA 结点，此时遍历其中具体包含的 NFA 结点以根据优先级规则确定该 DFA 结点是哪个规则的终态，或者是非终态。生成的 DFA 结点结构如下：

```
struct DFANode {
    int id;
    int flag = 0;
    std::map<char, DFANode*> ptrs;
};
```

5. DFA 最小化：经典的子集分割法，这里是把 DFA 划分为 n+1 个集合，n 为匹配不同规则的种类，每种终态都是一个独立集合，不能合并；采用结构 std::vector< std::set<int> > DFASets 和 int DFANodeIdx[]来维护当前子集划分，反复遍历，对于每个集合，对比其中第一个 DFA 结点和其他结点，如果对于相同字符可接受性不同或转移到不同集合的状态，就把这些结点分离出来作为一个新的集合；当一次遍历不生成新集合，算法结束。

6. DFA 生成词法分析器代码：包括三个需要的生成的函数

int DFAPush(char c); —— 接受字符，然后修改全局变量 DFAState，已经不接受该字符时返回 1，正常接受返回 0

int DFAExec(void); —— 执行匹配上正则式后的对应程序段

int DFATry(void); —— 确定当前状态是否为终态

7. 词法分析中的错误处理：词法分析程序一次读入整个文件后把字符串模拟为一个可以取出字符(input)和放回字符(unput)的输入流，在 input()和 unput()中内置对 yylineno 的计算；每次 yylex()调用是一个不断 input()直到字符不可接受为止，然后 unput()直到回到一个终态。如果直到把 yytext 的内容全部 unput 都没有一个可匹配的前缀，则发生了一个词法分析错误，报错，然后采取恐慌模式，空 input 下个字符来把它从输入流中移出，从第二个字符开始继续解析。
8. YaccProducer 提供便于协作的基础设施，列表如下：

```
std::map<int, std::string> map_idx_idname;
std::map<std::string, int> map_idname_idx;
std::set<int> terminators;
using Producer = std::pair< int, std::vector<int> >;
std::vector<Producer> producers;                    //产生式
int idnameToIdx(std::string producerItemName); //语法符号名称->编号
std::string idxToIdname(int producerItemIdx);   //语法符号编号->名称
void appendNewSymbol(int idx, std::string name, bool isTerminator);
                        /* 添加一个新发现的语法符号 */
bool isTerminator(int);
void writeTab(FILE * file, int(*print_)(FILE *, const char *, ...));
//writeTab 是基于终结符表生成 y.tab.h 的函数
```

9. yaccParser 解析语法规则文件，并调用上面提到的通用 API 生成把产生式和语法符号信息导入程序；由于不包含对执行动作的解析，读取的操作比较简单。主要的额外操作是生成符号信息和拓广文法增加符号和产生式，考虑到给 ASCII 码表示的终结符预留空间，我们从 300 开始编号终结符，把拓广文法开始符号记为__PROGRAM__并编号为 999，原文法开始符号编号为 1000，之后的其他符号依次递增，并把__PROGRAM__推导出原文法开始符号的产生式添加为 0 号产生式。
10. 生成 LR(1)DFA：这是通过确定相关的基本数据结构和拓展成员函数实现的，基本上是对 LR(1)状态 DFA 手工生成方法的模拟：

First 集合

```
std::map<int, std::set<int> > firsts;
void initFirsts();
```

产生式项目：不同预测符分开存储

```
struct LRItem {
 int position = 0;          //点的位置
 int gramarIdx = -1;        //产生式编号
 int predictiveSymbol;      //预测符
 bool operator < (const LRItem & t) const;
     //重载比较运算，使 LRItem 可以放进 std::set 并自动去重
}
struct LRState {
 int idx = -1;                      //状态号
```

```
    std::map<int, int> edgesMap; //<发出边上符号，状态号>
    std::set<LRItem> LRItemsSet; //状态内项目的集合
    void extend();                  //状态内拓展
   }
   struct LRDFA {
    int startState = 0;
    std::vector< LRState > statesVec;//store allLRState
    LRDFA();     // 构造即初始化，直接完成 LR(1)DFA
   }
```

1) 求 First 集——初始化 std::map<int, std::set<int> > firsts;

   1> 先提取所有终结符，初始化其 First 集为本身的单元素集合；其余均空

   2> 不断循环遍历所有产生式，尝试更新产生式左部的符号的 First 集：

   遍历右部符号

   把第一个右部符号的 First 集合中左部符号还未包含的都加入，ε 除外

   如果 ε 在这个符号的 First 集中，再继续看下一个符号的 First 集，否则结束遍历

   如果遍历右部符号 First 集都有 ε（或右部长度为 0），说明左部符号可规约为 ε，把 ε 加到它的 First 集

   3> 直到某次遍历所有产生式，都未能更新任何一个符号的 First 集，结束

2) 状态间拓展

   1> 把第一个产生式（S’-> • S, $）加到状态 0

   2> 状态 0 状态内拓展

   3> 遍历已生成的状态：

   状态间拓展：遍历状态内的产生式，尝试把点移进，得到新的状态

   若点在产生式的最后，说明是规约项，跳过

   否则右移点，得到一个移进后的新产生式，记录要接受的相应符号 c,加入 newStates[c] 中。 (std::map<int, LRItem> newStates)

   对于每个新的状态，状态内拓展，然后与已生成的状态判重

   对于相同的状态，把出边连接到对应已有状态上

   不重复的状态将添加为新的状态，然后连接一个指向该状态的边

   4> 算法在遍历完所有生成状态（无新状态）后结束

3) 状态内拓展

   1> 新建空队列，并将状态已有的产生式加入队列

   2> 当队列不为空时，取出下一个产生式进行处理：

   如果点在产生式最右部，跳过

   如果点右侧的符号是一个终结符，也跳过

   确定预测符，方法是看原产生式点右移后的剩余项，剩余项的第一项的 First 集全部加入预测符集合，如果含 ε 则不是加入 ε 而是把下一项的 First 集合也加入，直到如果最后一项的 First 集还含有 ε（或点右移后无剩余项），把原产生式的所有预测符也加入

   遍历所有产生式，找到那些左部为点右侧的非终结符的：把这些产生式分别设置为预测符集合中的每一个预测符后，查看状态集合中是否已有该产生式，是新产生式项目就加入状态并入队列

11. LR(1)合并同心集转化为 LALR：

   给 struct LRDFA 添加一个成员函数，在已经生成 LR(1)DFA 后合并它的同心集，用新

的 LALR 的 DFA 替换，LALRDFA 可以用相同的数据结构表达。

具体过程我们采用分阶段的方式进行，使用一个编号映射表 int stateBelongs[]，遍历 LR(1)的状态，两两比较是否为同心集；我们实际采用一个效率较低但不失可行性的方法——遍历 A 状态的项目，搜索 B 状态尝试对每个项目在 B 中找到一个同心项，然后反过来执行一次，均成功则表明两个状态同心；最终所有同心集都找到各自集合当中编号最小的那个作为代表，然后把所有项目都汇入代表结点，它们就是 LALR 结点；之后紧缩编号，并重新映射出边的指向，LALRDFA 就转化成功了。

12. 生成分析表并转化为代码

因为每一种状态实际能接受的符号通常不多，因此 LRTable 的每一行是一个映射而不是完整数组：

```
struct LRTableSide{
 int type;    // 0-reduce                        1-shift / GOTO
 int data;    // reduce: producerId       shift: stateId
};
using LRTableRow = std::map<int, LRTableSide>;
struct    LRTable{
 std::vector< LRTableRow > table;
 LRTable(const LRDFA & dfa);      //构造即初始化
 std::string      code_dump(std::string      state_name,      std::string      symbol_name)
// 代码生成
};
```

移进是 LRDFA 中本身直接包含的信息，容易实现

规约则是通过遍历状态内项目实现的

对于每个产生式项目，如果点在最右侧，说明是个规约项

预测符即代表向右看到该符号后要用此产生式规约

但是还有冲突的问题，对于同一个状态，这次规约可能和已有的移进的移入符号和其他规约的预测符冲突，解决方式如下：

移进优先，规约无法覆盖移进——这是对 if-else 冲突的一般解决方案

规约-规约冲突时，根据产生式在规则文件中出现的顺序确定优先程度，先出现的表达式被优先规约，对应在代码实现中，我们的产生式编号就是按出现顺序进行的，因此这可归结为规约到的产生式编号较小的优先

代码生成：LRTable 将翻译成一个二级跳转的函数，根据当前状态和下一个语法符号确定是返回值，约定移进时返回值是下个状态的编号，规约时返回值是-1-producerId，不接受时返回 65536

13. 语法分析过程和出错处理：

LRTable 生成的跳转函数需要一个状态栈控制函数配合，移进时把下个状态入栈，之后基于新的栈顶状态控制，规约时根据应用的产生式右部长度退栈，移进规约时还配合生成语法树结点。由于拓广文法符号规约后无法移进，因此只有在语法分析正确结束时，才会在规约时导致非终结符 GOTO 失败，把这作为语法分析正常退出的标志；而终结符移进失败则要报错。报错时我们遍历终结符表，提示出所有其他可能接续的终结符。然后我们采取循环移出下个终结符，直到有一个终结符能接着分析的简易恐慌模式，除非直到程序终止未能处理时给出第二次提示，不再反复报错。

语法树结点也用栈来保存，移进终结符时加入结点，规约时退栈并用规约出的符号作为它们的父结点入栈，语法分析成功结束时栈中最后留下语法树的根，是规约出的__PROGRAM__符号，如果出错一直到程序结束未能处理，则将留下一片未处理完的语法森林，仍然可以观察。

4 使用说明

4.1 **SeuLex** 使用说明

输入相配合的 littlec.l 词法规则文件；

Lex 程序接受 littlec.l，输出 lex.yy.c 代码文件，该文件依赖终结符表 y.tab.h（人工编写或由相配合的 yacc 生成）

lex.yy.c 编译得到的词法分析器程序可接受符合 littlec.l 词法规则的程序源代码文件，输出词法分析结果文件；

词法分析程序接受源代码(默认 test.c)，生成词法分析结果(默认 test.lo)。

4.2 SeuYacc 使用说明

输入相配合的 littlec.y 语法规则文件；

Yacc 程序接受 littlec.y，输出 y.tab.c 和 y.tab.h 代码文件；

y.tab.c 编译得到的语法分析器程序可接受符合 littlec.y 语法规则的程序源代码文件，输出语法分析结果；

语法分析程序接受词法分析结果（默认 test.lo），输出语法分析结果（默认输出到控制台）。

4.3 可执行文件的使用方法

最佳用法为在命令行状态下启动，源代码文件与可执行文件在同一文件夹为宜；（否则可能只能看到一闪而过的语法分析结果）

在命令行中调用词法分析程序，可默认吧同目录下的 test.c 源代码生成 test.lo 词法分析结果，再调用语法分析程序把对这个文件的语法结果输出到命令行环境中；

此外，如果正确提供命令行参数，将可以改为使用其他文件，示例如下：

（以 PowerShell 环境，词法程序取名 lexical.exe,语法程序取名 grammar.exe，源代码为 std.c 为例）

在该文件夹下进入 Powershell，输入命令

```
./lexical.exe std.c std.lo
```

将生成词法分析结果文件 std.lo

再输入命令

```
./grammar.exe std.lo std.go
```

将生成语法分析结果文件 std.go

任何情况下文件中只保存生成结果的信息，如果发生错误，错误信息会发到控制台使用户直接看到，而结果文件只包括错误恢复后尽力分析的结果。

5 测试用例与结果分析

测试一：

输入测试文件 test.c：

```
char cc[996] = " Hello world";
int main(){
    double h = 7.66e8f;
char g = 'h';
    return 0;
}
```

输出词法分析：

```
333 < 1 , 0 > char
300 < 1 , 5 > cc
91 < 1 , 7 > [
301 < 1 , 8 > 996
93 < 1 , 11 > ]
61 < 1 , 13 > =
302 < 1 , 15 > "Hello world"
59 < 1 , 28 > ;
335 < 2 , 0 > int
300 < 2 , 4 > main
40 < 2 , 8 > (
41 < 2 , 9 > )
123 < 2 , 10 > {
340 < 3 , 8 > double
300 < 3 , 15 > h
61 < 3 , 17 > =
301 < 3 , 19 > 7.66e8f
59 < 3 , 26 > ;
333 < 4 , 0 > char
300 < 4 , 5 > g
61 < 4 , 7 > =
301 < 4 , 9 > 'h'
59 < 4 , 12 > ;
362 < 5 , 8 > return
301 < 5 , 15 > 0
59 < 5 , 16 > ;
125 < 6 , 0 > }
```

输出语法分析：

```
__PROGRAM__ : => translation_unit    <1, 0> : char cc [ 996 ] = "Hello world" ; int main ( ) { double h = 7.66e8f ; char g = 'h' ; return 0 ; }
 translation_unit : => translation_unit    external_declaration    <1, 0> : char cc [ 996 ] = "Hello world" ; int main ( ) { double h = 7.66e8f ; char g = 'h' ; return 0 ; }
  translation_unit : => external_declaration    <1, 0> : char cc [ 996 ] = "Hello world" ;
   external_declaration : => declaration    <1, 0> : char cc [ 996 ] = "Hello world" ;
    declaration : => declaration_specifiers    init_declarator_list    ';'    <1, 0> : char cc [ 996 ] = "Hello world" ;
     declaration_specifiers : => type_specifier    <1, 0> : char
      type_specifier : => CHAR    <1, 0> : char
       CHAR <1, 0> : char
     init_declarator_list : => init_declarator    <1, 5> : cc [ 996 ] = "Hello world"
      init_declarator : => declarator    '='    initializer    <1, 5> : cc [ 996 ] = "Hello
world"
       declarator : => direct_declarator    <1, 5> : cc [ 996 ]
        direct_declarator : => direct_declarator    '['    assignment_expression    ']'
<1, 5> : cc [ 996 ]
         direct_declarator : => IDENTIFIER    <1, 5> : cc
          IDENTIFIER <1, 5> : cc
         '[' <1, 7> : [
         assignment_expression : => conditional_expression    <1, 8> : 996
          conditional_expression : => logical_or_expression    <1, 8> : 996
           logical_or_expression : => logical_and_expression    <1, 8> : 996
            logical_and_expression : => inclusive_or_expression    <1, 8> : 996
             inclusive_or_expression : => exclusive_or_expression    <1, 8> :
996
              exclusive_or_expression : => and_expression    <1, 8> : 996
               and_expression : => equality_expression    <1, 8> : 996
                equality_expression : => relational_expression    <1, 8> : 996
                 relational_expression : => shift_expression    <1, 8> : 996
                  shift_expression : => additive_expression    <1, 8> : 996
                   additive_expression : => multiplicative_expression    <1,
8> : 996
                    multiplicative_expression : => cast_expression    <1, 8> :
996
                     cast_expression : => unary_expression    <1, 8> : 996
                      unary_expression : => postfix_expression    <1, 8> : 996
                       postfix_expression : => primary_expression    <1, 8> :
996
                        primary_expression : => CONSTANT    <1, 8> : 996
                         CONSTANT <1, 8> : 996
         ']' <1, 11> : ]
       '=' <1, 13> : =
       initializer : => assignment_expression    <1, 15> : "Hello world"
        assignment_expression : => conditional_expression    <1, 15> : "Hello
world"
         conditional_expression : => logical_or_expression    <1, 15> : "Hello
world"
          logical_or_expression : => logical_and_expression    <1, 15> : "Hello
world"
           logical_and_expression : => inclusive_or_expression    <1, 15> :
"Hello world"
            inclusive_or_expression : => exclusive_or_expression    <1, 15> :
"Hello world"
             exclusive_or_expression : => and_expression    <1, 15> : "Hello
```

```
world"
                              and_expression : => equality_expression    <1, 15> : "Hello world"
                               equality_expression : => relational_expression    <1, 15> : "Hello
world"
                                relational_expression : => shift_expression    <1, 15> : "Hello
world"
                                 shift_expression : => additive_expression    <1, 15> : "Hello
world"
                                  additive_expression : => multiplicative_expression    <1,
15> : "Hello world"
                                   multiplicative_expression : => cast_expression    <1, 15> :
"Hello world"
                                    cast_expression : => unary_expression    <1, 15> : "Hello
world"
                                     unary_expression : => postfix_expression    <1, 15> :
"Hello world"
                                      postfix_expression : => primary_expression    <1, 15> :
"Hello world"
                                       primary_expression : => STRING_LITERAL    <1,
15> : "Hello world"
                                        STRING_LITERAL <1, 15> : "Hello world"
     ';' <1, 28> : ;
  external_declaration : => function_definition    <2, 0> : int main ( ) { double h =
7.66e8f ; char g = 'h' ; return 0 ; }
   function_definition : => declaration_specifiers declarator
compound_statement    <2, 0> : int main ( ) { double h = 7.66e8f ; char g = 'h' ; return
0 ; }
    declaration_specifiers : => type_specifier    <2, 0> : int
     type_specifier : => INT    <2, 0> : int
      INT <2, 0> : int
    declarator : => direct_declarator    <2, 4> : main ( )
     direct_declarator : => direct_declarator    '('    ')'    <2, 4> : main ( )
      direct_declarator : => IDENTIFIER    <2, 4> : main
       IDENTIFIER <2, 4> : main
      '(' <2, 8> : (
      ')' <2, 9> : )
    compound_statement : => '{'    block_item_list    '}'    <2, 10> : { double h =
7.66e8f ; char g = 'h' ; return 0 ; }
     '{' <2, 10> : {
     block_item_list : => block_item_list    block_item    <3, 8> : double h =
7.66e8f ; char g = 'h' ; return 0 ;
      block_item_list : => block_item_list    block_item    <3, 8> : double h =
7.66e8f ; char g = 'h' ;
       block_item_list : => block_item    <3, 8> : double h = 7.66e8f ;
```

```
         block_item : => declaration    <3, 8> : double h = 7.66e8f ;
          declaration : => declaration_specifiers   init_declarator_list   ';'   <3, 8> : double h = 7.66e8f ;
           declaration_specifiers : => type_specifier    <3, 8> : double
            type_specifier : => DOUBLE    <3, 8> : double
             DOUBLE <3, 8> : double
           init_declarator_list : => init_declarator    <3, 15> : h = 7.66e8f
            init_declarator : => declarator    '='    initializer    <3, 15> : h = 7.66e8f
             declarator : => direct_declarator    <3, 15> : h
              direct_declarator : => IDENTIFIER    <3, 15> : h
               IDENTIFIER <3, 15> : h
             '=' <3, 17> : =
             initializer : => assignment_expression    <3, 19> : 7.66e8f
              assignment_expression : => conditional_expression    <3, 19> : 7.66e8f
               conditional_expression : => logical_or_expression    <3, 19> : 7.66e8f
                logical_or_expression : => logical_and_expression    <3, 19> : 7.66e8f
                 logical_and_expression : => inclusive_or_expression    <3, 19> : 7.66e8f
                  inclusive_or_expression : => exclusive_or_expression    <3, 19> : 7.66e8f
                   exclusive_or_expression : => and_expression    <3, 19> : 7.66e8f
                    and_expression : => equality_expression    <3, 19> : 7.66e8f
                     equality_expression : => relational_expression    <3, 19> : 7.66e8f
                      relational_expression : => shift_expression    <3, 19> : 7.66e8f
                       shift_expression : => additive_expression    <3, 19> : 7.66e8f
                        additive_expression : => multiplicative_expression    <3, 19> : 7.66e8f
                         multiplicative_expression : => cast_expression    <3, 19> : 7.66e8f
                          cast_expression : => unary_expression    <3, 19> : 7.66e8f
                           unary_expression : => postfix_expression    <3, 19> : 7.66e8f
                            postfix_expression : => primary_expression    <3, 19> : 7.66e8f
                             primary_expression : => CONSTANT    <3, 19> : 7.66e8f
                             CONSTANT <3, 19> : 7.66e8f
          ';' <3, 26> : ;
       block_item : => declaration    <4, 0> : char g = 'h' ;
        declaration : => declaration_specifiers    init_declarator_list    ';'    <4, 0> :
char g = 'h' ;
         declaration_specifiers : => type_specifier    <4, 0> : char
          type_specifier : => CHAR    <4, 0> : char
           CHAR <4, 0> : char
         init_declarator_list : => init_declarator    <4, 5> : g = 'h'
          init_declarator : => declarator    '='    initializer    <4, 5> : g = 'h'
           declarator : => direct_declarator    <4, 5> : g
            direct_declarator : => IDENTIFIER    <4, 5> : g
             IDENTIFIER <4, 5> : g
           '=' <4, 7> : =
           initializer : => assignment_expression    <4, 9> : 'h'
            assignment_expression : => conditional_expression    <4, 9> : 'h'
             conditional_expression : => logical_or_expression    <4, 9> : 'h'
              logical_or_expression : => logical_and_expression    <4, 9> : 'h'
               logical_and_expression : => inclusive_or_expression    <4, 9> :
'h'
                inclusive_or_expression : => exclusive_or_expression    <4, 9> :
'h'
                 exclusive_or_expression : => and_expression    <4, 9> : 'h'
                  and_expression : => equality_expression    <4, 9> : 'h'
                   equality_expression : => relational_expression    <4, 9> : 'h'
                    relational_expression : => shift_expression    <4, 9> : 'h'
                     shift_expression : => additive_expression    <4, 9> : 'h'
                      additive_expression : => multiplicative_expression    <4,
9> : 'h'
                       multiplicative_expression : => cast_expression    <4,
9> : 'h'
                        cast_expression : => unary_expression    <4, 9> : 'h'
                         unary_expression : => postfix_expression    <4, 9> :
'h'
                          postfix_expression : => primary_expression    <4,
9> : 'h'
                           primary_expression : => CONSTANT    <4, 9> :
'h'
                            CONSTANT <4, 9> : 'h'
         ';' <4, 12> : ;
      block_item : => statement    <5, 8> : return 0 ;
       statement : => jump_statement    <5, 8> : return 0 ;
        jump_statement : => RETURN    expression    ';'    <5, 8> : return 0 ;
         RETURN <5, 8> : return
```

```
          expression : => assignment_expression    <5, 15> : 0
           assignment_expression : => conditional_expression    <5, 15> : 0
            conditional_expression : => logical_or_expression    <5, 15> : 0
             logical_or_expression : => logical_and_expression    <5, 15> : 0
              logical_and_expression : => inclusive_or_expression    <5, 15> : 0
               inclusive_or_expression : => exclusive_or_expression    <5, 15> :
0
                exclusive_or_expression : => and_expression    <5, 15> : 0
                 and_expression : => equality_expression    <5, 15> : 0
                  equality_expression : => relational_expression    <5, 15> : 0
                   relational_expression : => shift_expression    <5, 15> : 0
                    shift_expression : => additive_expression    <5, 15> : 0
                     additive_expression : => multiplicative_expression    <5,
15> : 0
                      multiplicative_expression : => cast_expression    <5, 15> :
0
                       cast_expression : => unary_expression    <5, 15> : 0
                        unary_expression : => postfix_expression    <5, 15> : 0
                         postfix_expression : => primary_expression    <5,
15> : 0
                          primary_expression : => CONSTANT    <5, 15> : 0
                           CONSTANT <5, 15> : 0
          ';' <5, 16> : ;
      '}' <6, 0> : }
```

测试二：

输入 test.c(有错):

```
char cc[996] = "Hello world";
int main(){
    double h = ;
char g = 'h';
    return 0;
}
```

词法分析：

```
333 < 1 , 0 > char
300 < 1 , 5 > cc
91 < 1 , 7 > [
301 < 1 , 8 > 996
93 < 1 , 11 > ]
61 < 1 , 13 > =
302 < 1 , 15 > "Hello world"
59 < 1 , 28 > ;
335 < 2 , 0 > int
300 < 2 , 4 > main
```

```
40 < 2 , 8 > (
41 < 2 , 9 > )
123 < 2 , 10 > {
340 < 3 , 8 > double
300 < 3 , 15 > h
61 < 3 , 17 > =
59 < 3 , 19 > ;
333 < 4 , 0 > char
300 < 4 , 5 > g
61 < 4 , 7 > =
301 < 4 , 9 > 'h'
59 < 4 , 12 > ;
362 < 5 , 8 > return
301 < 5 , 15 > 0
59 < 5 , 16 > ;
125 < 6 , 0 > }
```

语法分析：

```
line<3>, col<19>: grammar parsing error!
                Expected - '!'
                Expected - '&'
                Expected - '('
                Expected - '*'
                Expected - '+'
                Expected - '-'
                Expected - '{'
                Expected - '~'
                Expected - IDENTIFIER
                Expected - CONSTANT
                Expected - STRING_LITERAL
                Expected - SIZEOF
                Expected - INC_OP
                Expected - DEC_OP
                Unexpected : ';'

__PROGRAM__ : => translation_unit    <1, 0> : char cc [ 996 ] = "Hello world" ; int main ( ) { double h = g = 'h' ; return 0 ; }
  translation_unit : => translation_unit    external_declaration    <1, 0> : char cc [ 996 ] = "Hello world" ; int main ( ) { double h = g = 'h' ; return 0 ; }
    translation_unit : => external_declaration    <1, 0> : char cc [ 996 ] = "Hello world" ;
      external_declaration : => declaration    <1, 0> : char cc [ 996 ] = "Hello world" ;
        declaration : => declaration_specifiers    init_declarator_list    ';'    <1, 0> : char cc [ 996 ] = "Hello world" ;
          declaration_specifiers : => type_specifier    <1, 0> : char
```

```
       type_specifier : => CHAR    <1, 0> : char
        CHAR <1, 0> : char
      init_declarator_list : => init_declarator    <1, 5> : cc [ 996 ] = "Hello world"
       init_declarator : => declarator    '='    initializer    <1, 5> : cc [ 996 ] = "Hello
world"
        declarator : => direct_declarator    <1, 5> : cc [ 996 ]
         direct_declarator : => direct_declarator    '['    assignment_expression    ']'
<1, 5> : cc [ 996 ]
          direct_declarator : => IDENTIFIER    <1, 5> : cc
           IDENTIFIER <1, 5> : cc
          '[' <1, 7> : [
          assignment_expression : => conditional_expression    <1, 8> : 996
           conditional_expression : => logical_or_expression    <1, 8> : 996
            logical_or_expression : => logical_and_expression    <1, 8> : 996
             logical_and_expression : => inclusive_or_expression    <1, 8> : 996
              inclusive_or_expression : => exclusive_or_expression    <1, 8> :
996
               exclusive_or_expression : => and_expression    <1, 8> : 996
                and_expression : => equality_expression    <1, 8> : 996
                 equality_expression : => relational_expression    <1, 8> : 996
                  relational_expression : => shift_expression    <1, 8> : 996
                   shift_expression : => additive_expression    <1, 8> : 996
                    additive_expression : => multiplicative_expression    <1,
8> : 996
                     multiplicative_expression : => cast_expression    <1, 8> :
996
                      cast_expression : => unary_expression    <1, 8> : 996
                       unary_expression : => postfix_expression    <1, 8> : 996
                        postfix_expression : => primary_expression    <1, 8> :
996
                         primary_expression : => CONSTANT    <1, 8> : 996
                          CONSTANT <1, 8> : 996
          ']' <1, 11> : ]
        '=' <1, 13> : =
        initializer : => assignment_expression    <1, 15> : "Hello world"
         assignment_expression : => conditional_expression    <1, 15> : "Hello
world"
          conditional_expression : => logical_or_expression    <1, 15> : "Hello
world"
           logical_or_expression : => logical_and_expression    <1, 15> : "Hello
world"
            logical_and_expression : => inclusive_or_expression    <1, 15> :
"Hello world"
             inclusive_or_expression : => exclusive_or_expression    <1, 15> :
```

```
"Hello world"
             exclusive_or_expression : => and_expression   <1, 15> : "Hello
world"
              and_expression : => equality_expression   <1, 15> : "Hello world"
               equality_expression : => relational_expression   <1, 15> : "Hello
world"
                relational_expression : => shift_expression   <1, 15> : "Hello
world"
                 shift_expression : => additive_expression   <1, 15> : "Hello
world"
                  additive_expression : => multiplicative_expression   <1,
15> : "Hello world"
                   multiplicative_expression : => cast_expression   <1, 15> :
"Hello world"
                    cast_expression : => unary_expression   <1, 15> : "Hello
world"
                     unary_expression : => postfix_expression   <1, 15> :
"Hello world"
                      postfix_expression : => primary_expression   <1, 15> :
"Hello world"
                       primary_expression : => STRING_LITERAL   <1,
15> : "Hello world"
                        STRING_LITERAL <1, 15> : "Hello world"
     ';' <1, 28> : ;
  external_declaration : => function_definition   <2, 0> : int main ( ) { double h =
g = 'h' ; return 0 ; }
   function_definition : => declaration_specifiers declarator
compound_statement   <2, 0> : int main ( ) { double h = g = 'h' ; return 0 ; }
    declaration_specifiers : => type_specifier   <2, 0> : int
     type_specifier : => INT   <2, 0> : int
      INT <2, 0> : int
    declarator : => direct_declarator   <2, 4> : main ( )
     direct_declarator : => direct_declarator   '('   ')'   <2, 4> : main ( )
      direct_declarator : => IDENTIFIER   <2, 4> : main
       IDENTIFIER <2, 4> : main
      '(' <2, 8> : (
      ')' <2, 9> : )
    compound_statement : => '{'   block_item_list   '}'   <2, 10> : { double h = g
= 'h' ; return 0 ; }
     '{' <2, 10> : {
     block_item_list : => block_item_list   block_item   <3, 8> : double h = g =
'h' ; return 0 ;
      block_item_list : => block_item   <3, 8> : double h = g = 'h' ;
       block_item : => declaration   <3, 8> : double h = g = 'h' ;
```

```
        declaration : => declaration_specifiers    init_declarator_list    ';'    <3, 8> :
double h = g = 'h' ;
         declaration_specifiers : => type_specifier    <3, 8> : double
          type_specifier : => DOUBLE    <3, 8> : double
           DOUBLE <3, 8> : double
         init_declarator_list : => init_declarator    <3, 15> : h = g = 'h'
          init_declarator : => declarator    '='    initializer    <3, 15> : h = g = 'h'
           declarator : => direct_declarator    <3, 15> : h
            direct_declarator : => IDENTIFIER    <3, 15> : h
             IDENTIFIER <3, 15> : h
           '=' <3, 17> : =
           initializer : => assignment_expression    <4, 5> : g = 'h'
            assignment_expression : => unary_expression    assignment_operator
assignment_expression    <4, 5> : g = 'h'
             unary_expression : => postfix_expression    <4, 5> : g
              postfix_expression : => primary_expression    <4, 5> : g
               primary_expression : => IDENTIFIER    <4, 5> : g
                IDENTIFIER <4, 5> : g
             assignment_operator : => '='    <4, 7> : =
              '=' <4, 7> : =
             assignment_expression : => conditional_expression    <4, 9> : 'h'
              conditional_expression : => logical_or_expression    <4, 9> : 'h'
               logical_or_expression : => logical_and_expression    <4, 9> : 'h'
                logical_and_expression : => inclusive_or_expression    <4, 9> :
'h'
                 inclusive_or_expression : => exclusive_or_expression    <4,
9> : 'h'
                  exclusive_or_expression : => and_expression    <4, 9> : 'h'
                   and_expression : => equality_expression    <4, 9> : 'h'
                    equality_expression : => relational_expression    <4, 9> : 'h'
                     relational_expression : => shift_expression    <4, 9> : 'h'
                      shift_expression : => additive_expression    <4, 9> : 'h'
                       additive_expression : => multiplicative_expression
<4, 9> : 'h'
                        multiplicative_expression : => cast_expression    <4,
9> : 'h'
                         cast_expression : => unary_expression    <4, 9> : 'h'
                          unary_expression : => postfix_expression    <4, 9> :
'h'
                           postfix_expression : => primary_expression    <4,
9> : 'h'
                            primary_expression : => CONSTANT    <4, 9> :
'h'
                             CONSTANT <4, 9> : 'h'
         ';' <4, 12> : ;
      block_item : => statement    <5, 8> : return 0 ;
       statement : => jump_statement    <5, 8> : return 0 ;
        jump_statement : => RETURN    expression    ';'    <5, 8> : return 0 ;
         RETURN <5, 8> : return
         expression : => assignment_expression    <5, 15> : 0
          assignment_expression : => conditional_expression    <5, 15> : 0
           conditional_expression : => logical_or_expression    <5, 15> : 0
            logical_or_expression : => logical_and_expression    <5, 15> : 0
             logical_and_expression : => inclusive_or_expression    <5, 15> : 0
              inclusive_or_expression : => exclusive_or_expression    <5, 15> : 0
               exclusive_or_expression : => and_expression    <5, 15> : 0
                and_expression : => equality_expression    <5, 15> : 0
                 equality_expression : => relational_expression    <5, 15> : 0
                  relational_expression : => shift_expression    <5, 15> : 0
                   shift_expression : => additive_expression    <5, 15> : 0
                    additive_expression : => multiplicative_expression    <5, 15> : 0
                     multiplicative_expression : => cast_expression    <5, 15> : 0
                      cast_expression : => unary_expression    <5, 15> : 0
                       unary_expression : => postfix_expression    <5, 15> : 0
                        postfix_expression : => primary_expression    <5, 15> : 0
                         primary_expression : => CONSTANT    <5, 15> : 0
                          CONSTANT <5, 15> : 0
         ';' <5, 16> : ;
     '}' <6, 0> : }
```

测试三：

输入 test.c（功能：调用函数实现两个数交换）：

```
void swap(int* a, int* b) {
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(){
    int a=3, b=4;
    printf("After：");
    swap(&a, &b);
    system("pause");
    return 0;
}
```

词法分析：

```
343 < 1 , 0 > void
300 < 1 , 5 > swap
40 < 1 , 9 > (
335 < 1 , 10 > int
42 < 1 , 13 > *
300 < 1 , 15 > a
44 < 1 , 16 > ,
335 < 1 , 18 > int
42 < 1 , 21 > *
300 < 1 , 23 > b
41 < 1 , 24 > )
123 < 1 , 26 > {
335 < 2 , 8 > int
300 < 2 , 12 > tmp
59 < 2 , 15 > ;
300 < 3 , 8 > tmp
61 < 3 , 12 > =
42 < 3 , 14 > *
300 < 3 , 15 > a
59 < 3 , 16 > ;
42 < 4 , 8 > *
300 < 4 , 9 > a
61 < 4 , 11 > =
42 < 4 , 13 > *
300 < 4 , 14 > b
59 < 4 , 15 > ;
42 < 5 , 8 > *
300 < 5 , 9 > b
61 < 5 , 11 > =
300 < 5 , 13 > tmp
59 < 5 , 16 > ;
125 < 6 , 0 > }
335 < 8 , 0 > int
300 < 8 , 4 > main
40 < 8 , 8 > (
41 < 8 , 9 > )
123 < 8 , 10 > {
335 < 9 , 8 > int
300 < 9 , 12 > a
61 < 9 , 13 > =
301 < 9 , 14 > 3
44 < 9 , 15 > ,
300 < 9 , 17 > b
61 < 9 , 18 > =
301 < 9 , 19 > 4
59 < 9 , 20 > ;
300 < 10 , 8 > printf
40 < 10 , 14 > (
300 < 10 , 15 > After
41 < 10 , 20 > )
59 < 10 , 21 > ;
300 < 11 , 8 > swap
40 < 11 , 12 > (
38 < 11 , 13 > &
300 < 11 , 14 > a
44 < 11 , 15 > ,
38 < 11 , 17 > &
300 < 11 , 18 > b
41 < 11 , 19 > )
59 < 11 , 20 > ;
300 < 12 , 8 > system
40 < 12 , 14 > (
302 < 12 , 15 > "pause"
41 < 12 , 22 > )
59 < 12 , 23 > ;
362 < 13 , 8 > return
301 < 13 , 15 > 0
59 < 13 , 16 > ;
125 < 14 , 0 > }
```

语法分析:

```
__PROGRAM__ : => translation_unit   <1, 0> : void swap ( int * a , int * b ) { int tmp ; tmp = * a ; * a = * b ; * b = tmp ; } int main ( ) { int a = 3 , b = 4 ; printf ( After ) ; swap ( & a , & b ) ; system ( "pause" ) ; return 0 ; }
 translation_unit : => translation_unit external_declaration   <1, 0> : void swap ( int * a , int * b ) { int tmp ; tmp = * a ; * a = * b ; * b = tmp ; } int main ( ) { int a = 3 , b = 4 ; printf ( After ) ; swap ( & a , & b ) ; system ( "pause" ) ; return 0 ; }
  translation_unit : => external_declaration   <1, 0> : void swap ( int * a , int * b ) { int tmp ; tmp = * a ; * a = * b ; * b = tmp ; }
   external_declaration : => function_definition   <1, 0> : void swap ( int * a , int * b ) { int tmp ; tmp = * a ; * a = * b ; * b = tmp ; }
    function_definition : => declaration_specifiers declarator compound_statement   <1, 0> : void swap ( int * a , int * b ) { int tmp ; tmp = * a ; * a = * b ; * b = tmp ; }
     declaration_specifiers : => type_specifier   <1, 0> : void
      type_specifier : => VOID   <1, 0> : void
      VOID <1, 0> : void
    declarator : => direct_declarator   <1, 5> : swap ( int * a , int * b )
     direct_declarator : => direct_declarator   '('   parameter_type_list   ')'   <1, 5> : swap ( int * a , int * b )
      direct_declarator : => IDENTIFIER   <1, 5> : swap
       IDENTIFIER <1, 5> : swap
      '(' <1, 9> : (
      parameter_type_list : => parameter_list   <1, 10> : int * a , int * b
       parameter_list : => parameter_list   ','   parameter_declaration   <1, 10> : int * a , int * b
        parameter_list : => parameter_declaration   <1, 10> : int * a
         parameter_declaration : => declaration_specifiers   declarator   <1, 10> : int * a
          declaration_specifiers : => type_specifier   <1, 10> : int
           type_specifier : => INT   <1, 10> : int
            INT <1, 10> : int
          declarator : => pointer   direct_declarator   <1, 13> : * a
           pointer : => '*'   <1, 13> : *
            '*' <1, 13> : *
           direct_declarator : => IDENTIFIER   <1, 15> : a
            IDENTIFIER <1, 15> : a
        ',' <1, 16> : ,
        parameter_declaration : => declaration_specifiers   declarator   <1, 18> : int * b
         declaration_specifiers : => type_specifier   <1, 18> : int
          type_specifier : => INT   <1, 18> : int
           INT <1, 18> : int
         declarator : => pointer   direct_declarator   <1, 21> : * b
          pointer : => '*'   <1, 21> : *
           '*' <1, 21> : *
          direct_declarator : => IDENTIFIER   <1, 23> : b
           IDENTIFIER <1, 23> : b
      ')' <1, 24> : )
    compound_statement : => '{'   block_item_list   '}'   <1, 26> : { int tmp ; tmp = * a ; * a = * b ; * b = tmp ; }
     '{' <1, 26> : {
     block_item_list : => block_item_list   block_item   <2, 8> : int tmp ; tmp = * a ; * a = * b ; * b = tmp ;
      block_item_list : => block_item_list   block_item   <2, 8> : int tmp ; tmp = * a ; * a = * b ;
       block_item_list : => block_item_list   block_item   <2, 8> : int tmp ; tmp = * a ;
        block_item_list : => block_item   <2, 8> : int tmp ;
         block_item : => declaration   <2, 8> : int tmp ;
```

```
           declaration : => declaration_specifiers    init_declarator_list    ';'    <2,
8> : int tmp ;
            declaration_specifiers : => type_specifier    <2, 8> : int
             type_specifier : => INT    <2, 8> : int
              INT <2, 8> : int
            init_declarator_list : => init_declarator    <2, 12> : tmp
             init_declarator : => declarator    <2, 12> : tmp
              declarator : => direct_declarator    <2, 12> : tmp
               direct_declarator : => IDENTIFIER    <2, 12> : tmp
                IDENTIFIER <2, 12> : tmp
            ';' <2, 15> : ;
         block_item : => statement    <3, 8> : tmp = * a ;
          statement : => expression_statement    <3, 8> : tmp = * a ;
           expression_statement : => expression    ';'    <3, 8> : tmp = * a ;
            expression : => assignment_expression    <3, 8> : tmp = * a
             assignment_expression           :           =>           unary_expression
assignment_operator    assignment_expression    <3, 8> : tmp = * a
              unary_expression : => postfix_expression    <3, 8> : tmp
               postfix_expression : => primary_expression    <3, 8> : tmp
                primary_expression : => IDENTIFIER    <3, 8> : tmp
                 IDENTIFIER <3, 8> : tmp
              assignment_operator : => '='    <3, 12> : =
               '=' <3, 12> : =
              assignment_expression : => conditional_expression    <3, 14> : * a
               conditional_expression : => logical_or_expression    <3, 14> : * a
                logical_or_expression : => logical_and_expression    <3, 14> : *
a
                 logical_and_expression  :  =>  inclusive_or_expression     <3,
14> : * a
                  inclusive_or_expression  :  =>  exclusive_or_expression    <3,
14> : * a
                   exclusive_or_expression : => and_expression    <3, 14> : * a
                    and_expression : => equality_expression    <3, 14> : * a
                     equality_expression : => relational_expression    <3, 14> :
* a
                      relational_expression : => shift_expression    <3, 14> : *
a
                       shift_expression : => additive_expression    <3, 14> : *
a
                        additive_expression   :   =>   multiplicative_expression
<3, 14> : * a
                         multiplicative_expression : => cast_expression    <3,
14> : * a
                          cast_expression : => unary_expression    <3, 14> : *
```

```
a
                              unary_expression    :    =>    unary_operator
cast_expression   <3, 14> : * a
                               unary_operator : => '*'   <3, 14> : *
                                '*' <3, 14> : *
                               cast_expression  :  =>  unary_expression    <3,
15> : a
                                unary_expression : => postfix_expression   <3,
15> : a
                                 postfix_expression  :  =>  primary_expression
<3, 15> : a
                                  primary_expression : => IDENTIFIER   <3,
15> : a
                                   IDENTIFIER <3, 15> : a
            ';' <3, 16> : ;
        block_item : => statement   <4, 8> : * a = * b ;
         statement : => expression_statement   <4, 8> : * a = * b ;
          expression_statement : => expression   ';'   <4, 8> : * a = * b ;
           expression : => assignment_expression   <4, 8> : * a = * b
            assignment_expression : => unary_expression   assignment_operator
assignment_expression   <4, 8> : * a = * b
             unary_expression : => unary_operator   cast_expression   <4, 8> :
* a
              unary_operator : => '*'   <4, 8> : *
               '*' <4, 8> : *
              cast_expression : => unary_expression   <4, 9> : a
               unary_expression : => postfix_expression   <4, 9> : a
                postfix_expression : => primary_expression   <4, 9> : a
                 primary_expression : => IDENTIFIER   <4, 9> : a
                  IDENTIFIER <4, 9> : a
             assignment_operator : => '='   <4, 11> : =
              '=' <4, 11> : =
             assignment_expression : => conditional_expression   <4, 13> : * b
              conditional_expression : => logical_or_expression   <4, 13> : * b
               logical_or_expression : => logical_and_expression   <4, 13> : *
b
                logical_and_expression : => inclusive_or_expression   <4, 13> :
* b
                 inclusive_or_expression  :  =>  exclusive_or_expression    <4,
13> : * b
                  exclusive_or_expression : => and_expression   <4, 13> : * b
                   and_expression : => equality_expression   <4, 13> : * b
                    equality_expression : => relational_expression   <4, 13> :
* b
```

```
                     relational_expression : => shift_expression    <4, 13> : * b
                      shift_expression : => additive_expression    <4, 13> : * b
                       additive_expression : => multiplicative_expression
<4, 13> : * b
                        multiplicative_expression : => cast_expression    <4,
13> : * b
                         cast_expression : => unary_expression    <4, 13> : *
b
                          unary_expression : => unary_operator
cast_expression    <4, 13> : * b
                           unary_operator : => '*'    <4, 13> : *
                            '*' <4, 13> : *
                           cast_expression : => unary_expression    <4, 14> :
b
                            unary_expression : => postfix_expression    <4,
14> : b
                             postfix_expression : => primary_expression
<4, 14> : b
                              primary_expression : => IDENTIFIER    <4,
14> : b
                               IDENTIFIER <4, 14> : b
           ';' <4, 15> : ;
       block_item : => statement    <5, 8> : * b = tmp ;
        statement : => expression_statement    <5, 8> : * b = tmp ;
         expression_statement : => expression    ';'    <5, 8> : * b = tmp ;
          expression : => assignment_expression    <5, 8> : * b = tmp
           assignment_expression : => unary_expression    assignment_operator
assignment_expression    <5, 8> : * b = tmp
            unary_expression : => unary_operator    cast_expression    <5, 8> : *
b
             unary_operator : => '*'    <5, 8> : *
              '*' <5, 8> : *
             cast_expression : => unary_expression    <5, 9> : b
              unary_expression : => postfix_expression    <5, 9> : b
               postfix_expression : => primary_expression    <5, 9> : b
                primary_expression : => IDENTIFIER    <5, 9> : b
                 IDENTIFIER <5, 9> : b
            assignment_operator : => '='    <5, 11> : =
             '=' <5, 11> : =
            assignment_expression : => conditional_expression    <5, 13> : tmp
             conditional_expression : => logical_or_expression    <5, 13> : tmp
              logical_or_expression : => logical_and_expression    <5, 13> : tmp
               logical_and_expression : => inclusive_or_expression    <5, 13> :
tmp
```

```
                inclusive_or_expression : => exclusive_or_expression    <5,
13> : tmp
                 exclusive_or_expression : => and_expression    <5, 13> : tmp
                  and_expression : => equality_expression    <5, 13> : tmp
                   equality_expression : => relational_expression    <5, 13> :
tmp
                    relational_expression : => shift_expression    <5, 13> : tmp
                     shift_expression : => additive_expression    <5, 13> : tmp
                      additive_expression : => multiplicative_expression    <5,
13> : tmp
                       multiplicative_expression : => cast_expression    <5,
13> : tmp
                        cast_expression : => unary_expression    <5, 13> : tmp
                         unary_expression : => postfix_expression    <5, 13> :
tmp
                          postfix_expression : => primary_expression    <5,
13> : tmp
                           primary_expression : => IDENTIFIER    <5, 13> :
tmp
                            IDENTIFIER <5, 13> : tmp
          ';' <5, 16> : ;
      '}' <6, 0> : }
  external_declaration : => function_definition    <8, 0> : int main ( ) { int a = 3 , b
= 4 ; printf ( After ) ; swap ( & a , & b ) ; system ( "pause" ) ; return 0 ; }
   function_definition : => declaration_specifiers declarator
compound_statement    <8, 0> : int main ( ) { int a = 3 , b = 4 ; printf ( After ) ; swap
( & a , & b ) ; system ( "pause" ) ; return 0 ; }
    declaration_specifiers : => type_specifier    <8, 0> : int
     type_specifier : => INT    <8, 0> : int
      INT <8, 0> : int
    declarator : => direct_declarator    <8, 4> : main ( )
     direct_declarator : => direct_declarator    '('    ')'    <8, 4> : main ( )
      direct_declarator : => IDENTIFIER    <8, 4> : main
       IDENTIFIER <8, 4> : main
      '(' <8, 8> : (
      ')' <8, 9> : )
    compound_statement : => '{'    block_item_list    '}'    <8, 10> : { int a = 3 , b =
4 ; printf ( After ) ; swap ( & a , & b ) ; system ( "pause" ) ; return 0 ; }
     '{' <8, 10> : {
     block_item_list : => block_item_list    block_item    <9, 8> : int a = 3 , b = 4 ;
printf ( After ) ; swap ( & a , & b ) ; system ( "pause" ) ; return 0 ;
      block_item_list : => block_item_list    block_item    <9, 8> : int a = 3 , b =
4 ; printf ( After ) ; swap ( & a , & b ) ; system ( "pause" ) ;
       block_item_list : => block_item_list    block_item    <9, 8> : int a = 3 , b =
4 ; printf ( After ) ; swap ( & a , & b ) ;
        block_item_list : => block_item_list    block_item    <9, 8> : int a = 3 , b =
4 ; printf ( After ) ;
         block_item_list : => block_item    <9, 8> : int a = 3 , b = 4 ;
          block_item : => declaration    <9, 8> : int a = 3 , b = 4 ;
           declaration : => declaration_specifiers    init_declarator_list    ';'    <9,
8> : int a = 3 , b = 4 ;
            declaration_specifiers : => type_specifier    <9, 8> : int
             type_specifier : => INT    <9, 8> : int
              INT <9, 8> : int
            init_declarator_list : => init_declarator_list    ','    init_declarator
<9, 12> : a = 3 , b = 4
             init_declarator_list : => init_declarator    <9, 12> : a = 3
              init_declarator : => declarator    '='    initializer    <9, 12> : a = 3
               declarator : => direct_declarator    <9, 12> : a
                direct_declarator : => IDENTIFIER    <9, 12> : a
                 IDENTIFIER <9, 12> : a
               '=' <9, 13> : =
               initializer : => assignment_expression    <9, 14> : 3
                assignment_expression : => conditional_expression    <9, 14> :
3
                 conditional_expression : => logical_or_expression    <9, 14> :
3
                  logical_or_expression : => logical_and_expression    <9,
14> : 3
                   logical_and_expression : => inclusive_or_expression    <9,
14> : 3
                    inclusive_or_expression : => exclusive_or_expression
<9, 14> : 3
                     exclusive_or_expression : => and_expression    <9, 14> :
3
                      and_expression : => equality_expression    <9, 14> : 3
                       equality_expression : => relational_expression    <9,
14> : 3
                        relational_expression : => shift_expression    <9, 14> :
3
                         shift_expression : => additive_expression    <9, 14> :
3
                          additive_expression : => multiplicative_expression
<9, 14> : 3
                           multiplicative_expression : => cast_expression
<9, 14> : 3
                            cast_expression : => unary_expression    <9,
14> : 3
```

```
                             unary_expression : => postfix_expression    <9,
14> : 3
                              postfix_expression  :  =>  primary_expression
<9, 14> : 3
                               primary_expression  :  =>  CONSTANT    <9,
14> : 3
                                CONSTANT <9, 14> : 3
             ',' <9, 15> : ,
             init_declarator : => declarator    '='    initializer    <9, 17> : b = 4
              declarator : => direct_declarator    <9, 17> : b
               direct_declarator : => IDENTIFIER    <9, 17> : b
                IDENTIFIER <9, 17> : b
              '=' <9, 18> : =
              initializer : => assignment_expression    <9, 19> : 4
               assignment_expression : => conditional_expression    <9, 19> : 4
                conditional_expression : => logical_or_expression    <9, 19> : 4
                 logical_or_expression : => logical_and_expression    <9, 19> :
4
                  logical_and_expression  :  =>  inclusive_or_expression    <9,
19> : 4
                   inclusive_or_expression : => exclusive_or_expression    <9,
19> : 4
                    exclusive_or_expression : => and_expression    <9, 19> : 4
                     and_expression : => equality_expression    <9, 19> : 4
                      equality_expression  :  =>  relational_expression    <9,
19> : 4
                       relational_expression : => shift_expression    <9, 19> :
4
                        shift_expression : => additive_expression    <9, 19> :
4
                         additive_expression  :  =>  multiplicative_expression
<9, 19> : 4
                          multiplicative_expression   :   =>   cast_expression
<9, 19> : 4
                           cast_expression : => unary_expression    <9, 19> :
4
                            unary_expression  :  =>  postfix_expression    <9,
19> : 4
                             postfix_expression   :   =>   primary_expression
<9, 19> : 4
                              primary_expression  :  =>  CONSTANT    <9,
19> : 4
                               CONSTANT <9, 19> : 4
            ';' <9, 20> : ;
```

```
          block_item : => statement    <10, 8> : printf ( After ) ;
           statement : => expression_statement    <10, 8> : printf ( After ) ;
            expression_statement : => expression    ';'    <10, 8> : printf ( After ) ;
             expression : => assignment_expression    <10, 8> : printf ( After )
              assignment_expression  :  =>  conditional_expression    <10,  8>  :
printf ( After )
               conditional_expression  :  =>  logical_or_expression    <10,  8>  :
printf ( After )
                logical_or_expression  :  =>  logical_and_expression    <10,  8>  :
printf ( After )
                 logical_and_expression : => inclusive_or_expression    <10, 8> :
printf ( After )
                  inclusive_or_expression  :  =>  exclusive_or_expression    <10,
8> : printf ( After )
                   exclusive_or_expression  :  =>  and_expression    <10,  8>  :
printf ( After )
                    and_expression  :  =>  equality_expression    <10,  8>  :  printf
( After )
                     equality_expression : => relational_expression    <10, 8> :
printf ( After )
                      relational_expression  :  =>  shift_expression    <10,  8>  :
printf ( After )
                       shift_expression  :  =>  additive_expression    <10,  8>  :
printf ( After )
                        additive_expression    :    =>    multiplicative_expression
<10, 8> : printf ( After )
                         multiplicative_expression : => cast_expression    <10,
8> : printf ( After )
                          cast_expression  :  =>  unary_expression    <10,  8>  :
printf ( After )
                           unary_expression  :  =>  postfix_expression    <10,
8> : printf ( After )
                            postfix_expression  :  =>  postfix_expression    '('
argument_expression_list    ')'    <10, 8> : printf ( After )
                             postfix_expression    :    =>    primary_expression
<10, 8> : printf
                              primary_expression  :  =>  IDENTIFIER    <10,
8> : printf
                               IDENTIFIER <10, 8> : printf
                             '(' <10, 14> : (
                             argument_expression_list              :              =>
assignment_expression    <10, 15> : After
                              assignment_expression              :              =>
conditional_expression    <10, 15> : After
```

```
                                            conditional_expression          :          =>
logical_or_expression   <10, 15> : After
                                             logical_or_expression          :          =>
logical_and_expression   <10, 15> : After
                                              logical_and_expression        :          =>
inclusive_or_expression   <10, 15> : After
                                               inclusive_or_expression       :         =>
exclusive_or_expression   <10, 15> : After
                                                exclusive_or_expression      :         =>
and_expression   <10, 15> : After
                                                 and_expression             :          =>
equality_expression   <10, 15> : After
                                                  equality_expression        :         =>
relational_expression   <10, 15> : After
                                                   relational_expression     :         =>
shift_expression   <10, 15> : After
                                                    shift_expression         :         =>
additive_expression   <10, 15> : After
                                                     additive_expression     :         =>
multiplicative_expression   <10, 15> : After
                                                      multiplicative_expression   :    =>
cast_expression   <10, 15> : After
                                                       cast_expression       :         =>
unary_expression   <10, 15> : After
                                                        unary_expression     :         =>
postfix_expression   <10, 15> : After
                                                         postfix_expression    :       =>
primary_expression   <10, 15> : After
                                                          primary_expression   :       =>
IDENTIFIER   <10, 15> : After
                                                           IDENTIFIER  <10,  15>  :
After
                                         ')' <10, 20> : )
                         ';' <10, 21> : ;
                     block_item : => statement   <11, 8> : swap ( & a , & b ) ;
                      statement : => expression_statement   <11, 8> : swap ( & a , & b ) ;
                       expression_statement : => expression   ';'   <11, 8> : swap ( & a , & b ) ;
                        expression : => assignment_expression   <11, 8> : swap ( & a , & b )
                         assignment_expression : => conditional_expression   <11, 8> : swap
( & a , & b )
                          conditional_expression : => logical_or_expression   <11, 8> : swap
( & a , & b )
                           logical_or_expression  :  =>  logical_and_expression   <11,  8>  :
swap ( & a , & b )
```

```
                          logical_and_expression : => inclusive_or_expression    <11, 8> :
swap ( & a , & b )
                            inclusive_or_expression  :  =>  exclusive_or_expression    <11,
8> : swap ( & a , & b )
                              exclusive_or_expression : => and_expression    <11, 8> : swap
( & a , & b )
                                and_expression : => equality_expression    <11, 8> : swap ( &
a , & b )
                                  equality_expression  :  =>  relational_expression    <11,  8>  :
swap ( & a , & b )
                                    relational_expression  :  =>  shift_expression    <11,  8>  :
swap ( & a , & b )
                                      shift_expression  :  =>  additive_expression    <11,  8>  :
swap ( & a , & b )
                                        additive_expression    :    =>    multiplicative_expression
<11, 8> : swap ( & a , & b )
                                          multiplicative_expression  :  =>  cast_expression    <11,
8> : swap ( & a , & b )
                                            cast_expression  :  =>  unary_expression    <11,  8>  :
swap ( & a , & b )
                                              unary_expression : => postfix_expression    <11, 8> :
swap ( & a , & b )
                                                postfix_expression  :  =>  postfix_expression    '('
argument_expression_list    ')'    <11, 8> : swap ( & a , & b )
                                                  postfix_expression    :    =>    primary_expression
<11, 8> : swap
                                                    primary_expression  :  =>  IDENTIFIER    <11,
8> : swap
                                                      IDENTIFIER <11, 8> : swap
                                                  '(' <11, 12> : (
                                                  argument_expression_list              :              =>
argument_expression_list    ','    assignment_expression    <11, 13> : & a , & b
                                                    argument_expression_list              :              =>
assignment_expression    <11, 13> : & a
                                                      assignment_expression              :              =>
conditional_expression    <11, 13> : & a
                                                        conditional_expression              :              =>
logical_or_expression    <11, 13> : & a
                                                          logical_or_expression              :              =>
logical_and_expression    <11, 13> : & a
                                                            logical_and_expression            :            =>
inclusive_or_expression    <11, 13> : & a
                                                              inclusive_or_expression            :            =>
exclusive_or_expression    <11, 13> : & a
```

```
                                    exclusive_or_expression        :        =>
and_expression   <11, 13> : & a
                                     and_expression            :           =>
equality_expression   <11, 13> : & a
                                      equality_expression         :         =>
relational_expression   <11, 13> : & a
                                       relational_expression        :        =>
shift_expression   <11, 13> : & a
                                        shift_expression          :          =>
additive_expression   <11, 13> : & a
                                         additive_expression        :        =>
multiplicative_expression   <11, 13> : & a
                                          multiplicative_expression   :   =>
cast_expression   <11, 13> : & a
                                           cast_expression         :         =>
unary_expression   <11, 13> : & a
                                            unary_expression       :       =>
unary_operator   cast_expression   <11, 13> : & a
                                             unary_operator : => '&'   <11,
13> : &
                                              '&' <11, 13> : &
                                             cast_expression       :       =>
unary_expression   <11, 14> : a
                                              unary_expression     :     =>
postfix_expression   <11, 14> : a
                                               postfix_expression   :   =>
primary_expression   <11, 14> : a
                                                primary_expression  :  =>
IDENTIFIER   <11, 14> : a
                                                 IDENTIFIER <11, 14> :
a
                                 ',' <11, 15> : ,
                                 assignment_expression             :           =>
conditional_expression   <11, 17> : & b
                                  conditional_expression            :          =>
logical_or_expression   <11, 17> : & b
                                   logical_or_expression            :          =>
logical_and_expression   <11, 17> : & b
                                    logical_and_expression           :         =>
inclusive_or_expression   <11, 17> : & b
                                     inclusive_or_expression          :        =>
exclusive_or_expression   <11, 17> : & b
                                      exclusive_or_expression         :        =>
and_expression   <11, 17> : & b
```

```
                                                   and_expression : => equality_expression
<11, 17> : & b
                                                    equality_expression          :          =>
relational_expression   <11, 17> : & b
                                                     relational_expression         :         =>
shift_expression   <11, 17> : & b
                                                      shift_expression            :           =>
additive_expression   <11, 17> : & b
                                                       additive_expression         :          =>
multiplicative_expression   <11, 17> : & b
                                                        multiplicative_expression   :    =>
cast_expression   <11, 17> : & b
                                                         cast_expression          :          =>
unary_expression   <11, 17> : & b
                                                          unary_expression        :         =>
unary_operator   cast_expression   <11, 17> : & b
                                                           unary_operator : => '&'   <11,
17> : &
                                                            '&' <11, 17> : &
                                                           cast_expression        :          =>
unary_expression   <11, 18> : b
                                                            unary_expression      :        =>
postfix_expression   <11, 18> : b
                                                             postfix_expression    :     =>
primary_expression   <11, 18> : b
                                                              primary_expression   :    =>
IDENTIFIER   <11, 18> : b
                                                               IDENTIFIER <11, 18> : b
                                             ')' <11, 19> : )
                          ';' <11, 20> : ;
                      block_item : => statement   <12, 8> : system ( "pause" ) ;
                       statement : => expression_statement   <12, 8> : system ( "pause" ) ;
                        expression_statement : => expression   ';'   <12, 8> : system ( "pause" ) ;
                         expression : => assignment_expression   <12, 8> : system ( "pause" )
                          assignment_expression : => conditional_expression   <12, 8> : system
( "pause" )
                           conditional_expression : => logical_or_expression   <12, 8> : system
( "pause" )
                            logical_or_expression  :  =>  logical_and_expression   <12,  8>  :
system ( "pause" )
                             logical_and_expression  :  =>  inclusive_or_expression   <12,  8>  :
system ( "pause" )
                              inclusive_or_expression : => exclusive_or_expression   <12, 8> :
system ( "pause" )
```

```
                              exclusive_or_expression : => and_expression   <12, 8> : system
( "pause" )
                               and_expression : => equality_expression   <12, 8> : system
( "pause" )
                                equality_expression : => relational_expression   <12, 8> :
system ( "pause" )
                                 relational_expression : => shift_expression   <12, 8> :
system ( "pause" )
                                  shift_expression : => additive_expression   <12, 8> :
system ( "pause" )
                                   additive_expression : => multiplicative_expression   <12,
8> : system ( "pause" )
                                    multiplicative_expression : => cast_expression   <12,
8> : system ( "pause" )
                                     cast_expression : => unary_expression   <12, 8> :
system ( "pause" )
                                      unary_expression : => postfix_expression   <12, 8> :
system ( "pause" )
                                       postfix_expression : => postfix_expression   '('
argument_expression_list   ')'   <12, 8> : system ( "pause" )
                                        postfix_expression : => primary_expression   <12,
8> : system
                                         primary_expression : => IDENTIFIER   <12, 8> :
system
                                          IDENTIFIER <12, 8> : system
                                        '(' <12, 14> : (
                                        argument_expression_list         :         =>
assignment_expression   <12, 15> : "pause"
                                         assignment_expression           :         =>
conditional_expression   <12, 15> : "pause"
                                          conditional_expression         :         =>
logical_or_expression   <12, 15> : "pause"
                                           logical_or_expression         :         =>
logical_and_expression   <12, 15> : "pause"
                                            logical_and_expression       :         =>
inclusive_or_expression   <12, 15> : "pause"
                                             inclusive_or_expression     :         =>
exclusive_or_expression   <12, 15> : "pause"
                                              exclusive_or_expression    :         =>
and_expression   <12, 15> : "pause"
                                               and_expression : => equality_expression
<12, 15> : "pause"
                                                equality_expression      :         =>
relational_expression   <12, 15> : "pause"
```

```
                                                    relational_expression          :          =>
shift_expression    <12, 15> : "pause"
                                                     shift_expression             :           =>
additive_expression    <12, 15> : "pause"
                                                      additive_expression          :          =>
multiplicative_expression    <12, 15> : "pause"
                                                       multiplicative_expression     :     =>
cast_expression    <12, 15> : "pause"
                                                        cast_expression           :           =>
unary_expression    <12, 15> : "pause"
                                                         unary_expression          :          =>
postfix_expression    <12, 15> : "pause"
                                                          postfix_expression        :        =>
primary_expression    <12, 15> : "pause"
                                                           primary_expression      :      =>
STRING_LITERAL    <12, 15> : "pause"
                                                            STRING_LITERAL       <12,
15> : "pause"
                                         ')' <12, 22> : )
                         ';' <12, 23> : ;
      block_item : => statement    <13, 8> : return 0 ;
       statement : => jump_statement    <13, 8> : return 0 ;
        jump_statement : => RETURN    expression    ';'    <13, 8> : return 0 ;
         RETURN <13, 8> : return
         expression : => assignment_expression    <13, 15> : 0
          assignment_expression : => conditional_expression    <13, 15> : 0
           conditional_expression : => logical_or_expression    <13, 15> : 0
            logical_or_expression : => logical_and_expression    <13, 15> : 0
             logical_and_expression : => inclusive_or_expression    <13, 15> : 0
              inclusive_or_expression : => exclusive_or_expression    <13, 15> :
0
               exclusive_or_expression : => and_expression    <13, 15> : 0
                and_expression : => equality_expression    <13, 15> : 0
                 equality_expression : => relational_expression    <13, 15> : 0
                  relational_expression : => shift_expression    <13, 15> : 0
                   shift_expression : => additive_expression    <13, 15> : 0
                    additive_expression  :  =>  multiplicative_expression    <13,
15> : 0
                     multiplicative_expression  :  =>  cast_expression     <13,
15> : 0
                      cast_expression : => unary_expression    <13, 15> : 0
                       unary_expression : => postfix_expression    <13, 15> :
0
                        postfix_expression  :  =>  primary_expression    <13,
15> : 0
                                        primary_expression : => CONSTANT    <13, 15> : 0
                                          CONSTANT <13, 15> : 0
              ';' <13, 16> : ;
        '}' <14, 0> : }
```

6 课程设计总结

这次的课程设计，进一步加深了我们对编译原理知识的理解，对词法分析和语法分析的切身体会更加深刻，还掌握了之前非重点的内容——LALR 分析、词法分析和语法分析的错误恢复。

通过把大的词法规则分析和语法规则分析分解为多个相对较独立的模块，我们发现实现一个通用的基本的词法规则解析 lex 和语法规则解析 yacc 并不困难，只需要对各个我们已经在编译原理中学习到的转化步骤透彻理解，然后挑选适当的数据结构，转化为代码形式。因此，程序设计的实现切忌畏难的空谈，反复地实践、特别是代码实践可以逐步地解决我们的疑难。

错误报告和恢复是我们的一个重要拓展内容，我们拓展学习了错误恢复的基本方法，并且为分析程序添加了简单的处理步骤。

需要改进的内容：

1. 我们的程序的模块划分还比较粗糙，这和我们一开始时着急于进度，基于功能目标编码优先于模块划分有关，组内协作是基于一人主导的代码的手动合并，没有引入 git 等控制系统，因此我们的代码模块间耦合比较紧，并且有些划分存在模糊，还需要进一步划分模块；
2. LR(1)DFA 转 LALR 步骤写的比较粗略，从测试来看耗时约一分钟多，可以考虑再优化算法，改进使用体验；
3. 我们对正则式写法的一些限制虽然没限制根本表达能力，但使正则式写起来变得麻烦了一些，后续可引入成熟度更高的分析方法来支持更完善的正则特性；
4. 我们对 lex 和 yacc 规则文件编写的容错率不高，在许多地方不允许出现多余的空格，并且在规则编写有问题时会静默出错，较难确知哪里写得有问题，因此需要进一步细化我们的分析过程，并且考虑分析规则失败时的情况写错误处理代码；
5. 我们的词法分析中间结果是文件形式，这已经不太符合现今一般编译器的做法，输出到文件再读取这一转换也确实存在一些麻烦，进一步改变应该让 lex 与 yacc 合作，词法分析程序提供一个让语法分析程序直接获取下一个单词的接口；
6. 我们的程序还不支持中文字符串等多语言编码拓展，这需要改革词法处理的流程；
7. 我们的语法分析错误恢复实用性还不高，后续可改进语法分析器结构，允许用户定义出错情况。
8. 还未添加语义分析等后续编译过程，不过在我们保存了完整语法树的基础上应该容易继续进行后续工作。