東南大學

编译原理课程设计

SeuYacc 报告

东南大学计算机科学与工程学院

二〇一九年六月

1 编译对象与编译功能

1.1 编译对象

本设计中使用了 1999 年的 ANSI 标准 C 文法。未做任何修改。

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME
```

所有的赋值语句中的终结符

```
%token TYPEDEF EXTERN STATIC AUTO REGISTER INLINE RESTRICT
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST
VOLATILE VOID
%token BOOL COMPLEX IMAGINARY
%token STRUCT UNION ENUM ELLIPSIS
```

所有的关键字对应的终结符

```
%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK
RETURN
```

所有跳转语句中需要的终结符

此处定义了文法中需要的所有可能终结符，与 lex 的词法相对应，含义见 SeuLex 设计报告，在此不再赘述。

```
%start translation_unit
```

此处选定一个翻译单元为一个 yacc 处理的单位

```
%%

primary_expression

: IDENTIFIER
| CONSTANT
| STRING_LITERAL
| '(' expression ')'
;
```

基础表达式，包含了表达式中可能出现的单一元素

```
postfix_expression

: primary_expression
| postfix_expression '[' expression ']'
| postfix_expression '(' ')'
| postfix_expression '(' argument_expression_list ')'
| postfix_expression '.' IDENTIFIER
| postfix_expression PTR_OP IDENTIFIER
| postfix_expression INC_OP
| postfix_expression DEC_OP
| '(' type_name ')' '{' initializer_list '}'
| '(' type_name ')' '{' initializer_list ',' '}'
;
```

后缀表达式，列举了在基础表达式的后面可能出现的所有文法结构

```
argument_expression_list

: assignment_expression
| argument_expression_list ',' assignment_expression
;
```

参数表达式列表，表示用逗号分隔的一系列赋值表达式

```
unary_expression

: postfix_expression
| INC_OP unary_expression
| DEC_OP unary_expression
| unary_operator cast_expression
| SIZEOF unary_expression
| SIZEOF '(' type_name ')'
;
```

一元表达式，表示所有一元操作符的文法

```
unary_operator

: '&'
| '*'
| '+'
| '-'
| '~'
| '!'
;
```
一元操作符，列举了所有单符号的一元操作符

```
cast_expression

: unary_expression
| '(' type_name ')' cast_expression
;
```
显式类型转换表达式，表示了显式类型转换的文法

```
multiplicative_expression

: cast_expression
| multiplicative_expression '*' cast_expression
| multiplicative_expression '/' cast_expression
| multiplicative_expression '%' cast_expression
;
```
乘除表达式，表达了乘除和取余操作的文法

```
additive_expression

: multiplicative_expression
| additive_expression '+' multiplicative_expression
| additive_expression '-' multiplicative_expression
;
```
加减表达式，表达了加减操作的文法

```
shift_expression

: additive_expression
| shift_expression LEFT_OP additive_expression
| shift_expression RIGHT_OP additive_expression
;
```

移位表达式，表达了移位操作的文法

```
relational_expression

: shift_expression
| relational_expression '<' shift_expression
| relational_expression '>' shift_expression
| relational_expression LE_OP shift_expression
| relational_expression GE_OP shift_expression
;
```

关系运算表达式，表达了大于，大于等于，小于，小于等于关系运算的文法

```
equality_expression

: relational_expression
| equality_expression EQ_OP relational_expression
| equality_expression NE_OP relational_expression
;
```

相等表达式，表达了相等和不等关系的文法

```
and_expression

: equality_expression
| and_expression '&' equality_expression
;
```

与表达式，表达了与运算的文法

```
exclusive_or_expression

: and_expression
| exclusive_or_expression '^' and_expression
;
```

非表达式，表达了非运算的文法

```
inclusive_or_expression

: exclusive_or_expression
```

```
| inclusive_or_expression '|' exclusive_or_expression
;
```
或表达式，表达了或运算的文法

```
logical_and_expression

: inclusive_or_expression
| logical_and_expression AND_OP inclusive_or_expression
;
```
逻辑与表达式，表达了逻辑与运算的文法

```
logical_or_expression

: logical_and_expression
| logical_or_expression OR_OP logical_and_expression
;
```
逻辑或表达式，表达了逻辑或运算的文法

```
conditional_expression

: logical_or_expression
| logical_or_expression '?' expression ':' conditional_expression
;
```
条件表达式，表达了三元条件运算的文法

```
assignment_expression

: conditional_expression
| unary_expression assignment_operator assignment_expression
;
```
赋值表达式，表达了赋值表达式的文法

```
assignment_operator

: '='
| MUL_ASSIGN
| DIV_ASSIGN
| MOD_ASSIGN
| ADD_ASSIGN
| SUB_ASSIGN
| LEFT_ASSIGN
| RIGHT_ASSIGN
| AND_ASSIGN
| XOR_ASSIGN
| OR_ASSIGN
;
```

赋值符号，所有赋值运算符

```
expression

: assignment_expression
| expression ',' assignment_expression
;
```

表达式，由多个赋值表达式用逗号分隔

```
constant_expression

: conditional_expression
;
```

常表达式，条件表达式

```
declaration

: declaration_specifiers ';'
| declaration_specifiers init_declarator_list ';'
;
```

声明，声明以分号结束

```
declaration_specifiers

: storage_class_specifier
| storage_class_specifier declaration_specifiers
| type_specifier
| type_specifier declaration_specifiers
| type_qualifier
| type_qualifier declaration_specifiers
| function_specifier
| function_specifier declaration_specifiers
;
```
声明的标识符，可以是存储类标识符、类型标识符或函数标识符开头

```
init_declarator_list

: init_declarator
| init_declarator_list ',' init_declarator
;
```
初始声明符列表，之间用逗号分隔

```
init_declarator

: declarator
| declarator '=' initializer
;
```
初始声明符，表示了一个连续赋值语句

```
storage_class_specifier

: TYPEDEF
| EXTERN
| STATIC
| AUTO
| REGISTER
;
```
存储类指示符，表示需要特别开辟内存空间的特殊指示符

```
type_specifier

: VOID
| CHAR
| SHORT
| INT
| LONG
| FLOAT
| DOUBLE
| SIGNED
| UNSIGNED
| BOOL
| COMPLEX
| IMAGINARY
| struct_or_union_specifier
| enum_specifier
| TYPE_NAME
;
```

类型指示符，表示所有类型指示符

```
struct_or_union_specifier

: struct_or_union IDENTIFIER '{' struct_declaration_list '}'
| struct_or_union '{' struct_declaration_list '}'
| struct_or_union IDENTIFIER
;
```

结构体或集合的标识符

```
struct_or_union

: STRUCT
| UNION
;
```

结构体或集合的关键字

```
struct_declaration_list

: struct_declaration
| struct_declaration_list struct_declaration
;
```

结构体的语句列表

```
struct_declaration

: specifier_qualifier_list struct_declarator_list ';'
;
```

结构体的声明，以分号结束

```
specifier_qualifier_list

: type_specifier specifier_qualifier_list
| type_specifier
| type_qualifier specifier_qualifier_list
| type_qualifier
;
```

指示符和资格符号的列表

```
struct_declarator_list

: struct_declarator
| struct_declarator_list ',' struct_declarator
;
```

结构体声明符列表，用逗号分隔

```
struct_declarator

: declarator
| ':' constant_expression
| declarator ':' constant_expression
;
```

结构体声明符，分号后接常表达式

```
enum_specifier

: ENUM '{' enumerator_list '}'
| ENUM IDENTIFIER '{' enumerator_list '}'
| ENUM '{' enumerator_list ',' '}'
| ENUM IDENTIFIER '{' enumerator_list ',' '}'
| ENUM IDENTIFIER
;
```
枚举指示符，后接枚举列表，用花括号括起来

```
enumerator_list

: enumerator
| enumerator_list ',' enumerator
;
```
枚举列表，用逗号分隔枚举元素

```
enumerator

: IDENTIFIER
| IDENTIFIER '=' constant_expression
;
```
枚举元素，可以就是一个标识符，或者一个带等号赋值的标识符

```
type_qualifier

: CONST
| RESTRICT
| VOLATILE
;
```
类型资格符

```
function_specifier

: INLINE
;
```
函数指示符，可以是 INLINE

```
declarator

: pointer direct_declarator
| direct_declarator
```

```
;
```

声明符，可以是一个直接声明符，或者前面加上指针

```
direct_declarator

: IDENTIFIER
| '(' declarator ')'
| direct_declarator '[' type_qualifier_list assignment_expression
']'
| direct_declarator '[' type_qualifier_list ']'
| direct_declarator '[' assignment_expression ']'
| direct_declarator '[' STATIC type_qualifier_list
assignment_expression ']'
| direct_declarator '[' type_qualifier_list STATIC
assignment_expression ']'
| direct_declarator '[' type_qualifier_list '*' ']'
| direct_declarator '[' '*' ']'
| direct_declarator '[' ']'
| direct_declarator '(' parameter_type_list ')'
| direct_declarator '(' identifier_list ')'
| direct_declarator '(' ')'
;
```

直接声明符，有很多种形式

```
pointer

: '*'
| '*' type_qualifier_list
| '*' pointer
| '*' type_qualifier_list pointer
;
```

指针，可以是单独的星号，或其后加上类型资格符列表或指针

```
type_qualifier_list

: type_qualifier
| type_qualifier_list type_qualifier
```

```
;
类型资格符列表，多个类型资格符

parameter_type_list

: parameter_list
| parameter_list ',' ELLIPSIS
;
参数类型列表，后面可接逗号加省略号

parameter_list

: parameter_declaration
| parameter_list ',' parameter_declaration
;
参数列表，参数声明用逗号分隔

parameter_declaration

: declaration_specifiers declarator
| declaration_specifiers abstract_declarator
| declaration_specifiers
;
参数声明，声明指示符加上声明或抽象声明

identifier_list

: IDENTIFIER
| identifier_list ',' IDENTIFIER
;
标识符列表，之间用逗号分隔

type_name

: specifier_qualifier_list
| specifier_qualifier_list abstract_declarator
;
```

类型名称，可以是指示符状态符列表，或后再接抽象声明符

```
abstract_declarator

: pointer
| direct_abstract_declarator
| pointer direct_abstract_declarator
;
```

抽象声明符，直接抽象声明符或前面加指针

```
direct_abstract_declarator

: '(' abstract_declarator ')'
| '[' ']'
| '[' assignment_expression ']'
| direct_abstract_declarator '[' ']'
| direct_abstract_declarator '[' assignment_expression ']'
| '[' '*' ']'
| direct_abstract_declarator '[' '*' ']'
| '(' ')'
| '(' parameter_type_list ')'
| direct_abstract_declarator '(' ')'
| direct_abstract_declarator '(' parameter_type_list ')'
;
```

直接抽象声明符，有多种形式，用中括号或小括号包含赋值表达式

```
initializer

: assignment_expression
| '{' initializer_list '}'
| '{' initializer_list ',' '}'
;
```

初始化块，用中括号包裹初始化列表

```
initializer_list

: initializer
```

```
| designation initializer
| initializer_list ',' initializer
| initializer_list ',' designation initializer
;
```

初始化列表，用逗号分隔初始化块

```
designation

: designator_list '='
;
```

指定块，指定符后接等号

```
designator_list

: designator
| designator_list designator
;
```

指定列表，多个指定符

```
designator

: '[' constant_expression ']'
| '.' IDENTIFIER
;
```

指定符，后接中括号寻址或点加标识符

```
statement

: labeled_statement
| compound_statement
| expression_statement
| selection_statement
| iteration_statement
| jump_statement
;
```

一个语句，可能是很多种语句

```
labeled_statement

: IDENTIFIER ':' statement
| CASE constant_expression ':' statement
| DEFAULT ':' statement
;
```

标签语句，标签名称加冒号

```
compound_statement

: '{' '}'
| '{' block_item_list '}'
;
```

混合语句，块列表外套花括号

```
block_item_list

: block_item
| block_item_list block_item
;
```

块列表，多个块项组成

```
block_item

: declaration
| statement
;
```

块项，声明或语句

```
expression_statement

: ';'
| expression ';'
;
```

表达式语句，表达式加分号

```
selection_statement

: IF '(' expression ')' statement
| IF '(' expression ')' statement ELSE statement
| SWITCH '(' expression ')' statement
;
```

选择语句，ifelse 或 switch 语句

```
iteration_statement

: WHILE '(' expression ')' statement
| DO statement WHILE '(' expression ')' ';'
| FOR '(' expression_statement expression_statement ')' statement
| FOR '(' expression_statement expression_statement expression ')'
statement
| FOR '(' declaration expression_statement ')' statement
| FOR '(' declaration expression_statement expression ')' statement
;
```

迭代语句，for 或 while 或 do while 循环

```
jump_statement

: GOTO IDENTIFIER ';'
| CONTINUE ';'
| BREAK ';'
| RETURN ';'
| RETURN expression ';'
;
```

跳转语句，多种跳转

```
translation_unit

: external_declaration
| translation_unit external_declaration
;
```

翻译单元，由多个外部声明构成

```
external_declaration

: function_definition
| declaration
;
```

外部声明，是函数定义或者声明

```
function_definition

: declaration_specifiers declarator declaration_list
compound_statement
| declaration_specifiers declarator compound_statement
;
```

函数定义，是否带参数的两种函数定义格式

```
declaration_list

: declaration
| declaration_list declaration
;
```

声明列表，多个声明的组合

1.2 编译功能

- Yacc 输入文件解析：ParseYacc.h
- 求 First 集：First.h
- 依照上下文无关文法生成 LR(1)DFA：CFGToLRDFA.h
- LR(1) DFA 转化为 LALR(1) DFA：LR1ToLALR.h
- LALR DFA 转化为 LALR 分析表：LR1ToLALR.h
- SEU_Yacc 代码生成：CodeGeneration.h
- 总控程序：SEU_Yacc.cpp
- 数据结构定义和全局变量的定义：helper.h

2. 主要特色

1. 本设计中实现了 LR(1)到 LALR(1)的映射，在 LALR(1)的基础上构造了预测分析表；
2. 本设计对 ANSI 1999 标准 C 文法做到了完全一致，未做任何删减；
3. 本设计实现了对 SEU_Lex 的输出兼容，可以直接将 SEU_Lex 的输出作为 SEU_Yacc 的输入；
4. 本设计中利用 STL 巧妙地设计了大量数据结构，并利用了编译器的优化，在保证内存安全的同时提高了效率，项目运行仅需不到 3 秒。
5. 结果呈现美观，在终端里也能很好地将 AST 打印出来，并且有一张可视化 PDF 版本的 AST，更加直观明了。

3 概要设计与详细设计

3.1 概要设计

- Parse_Yacc.h
  - ○ `int Parse_Yacc(const string& filename,vector<Symbol>& symbolvec,`
```
          ProductionVec& production,
          ProducerVec& production_int,
          vector<string>&FuncVec)
```

读取并分析.y 文件，将其分隔成不同部分，存于特殊形式用于程序后续处理。返回 1 表示正确，返回 0 表示错误。

- First.h
  - ○ `calc_first()`

对所有读入的产生式中的非终结符/终结符，求他们的 first 集。

○ `first_string(set<int>&, const vector<int>&)`

对一个符号串，求其 first 集，用于求解 LR(1)DFA 时计算预测符

○ `intersection(set<int>&, const set<int>&)`

将一个集合中的内容加入另一个集合中

○ `first_symbol(set<int>&, int&, set<int>&)`

求解一个符号的 first 集

- CFGToLRDFA.h
  - ○ `CFGToLRDFA(Collection&)`

依照所有读入的产生式以及 First 集，生成一个 LR(1)DFA。

○ `epsilon_clousure(const ItemSet&, ItemSet&)`

对于一个给定的 LR 项集(仅包含内核项)，求其状态内扩展获得完整的 LR 项集并放入另一个 ItemSet 中用来产生后继状态。

○ `stateEdge_construct(const vector<Item> &, map<int, ItemSet >& )`

对于一个给定的 LR 项集(完整项集，同时包括内核项和非内核项)，求其状态间扩展，对应出边和扩展状态放入 map 中存储

- LR1ToLALR.h
  - ○ `void LR1ToLALR( Collection& LRcollection,`
`Collection& LALRcollection)`

合并同心项，将 LR(1) DFA 转化成对应的 LALR(1) DFA。

○ `void LR1ToTable`
`(Collection& LRcollection,Parse_Table& Parsing_table)`

依照 LALR(1) DFA 生成对应的预测分析表。

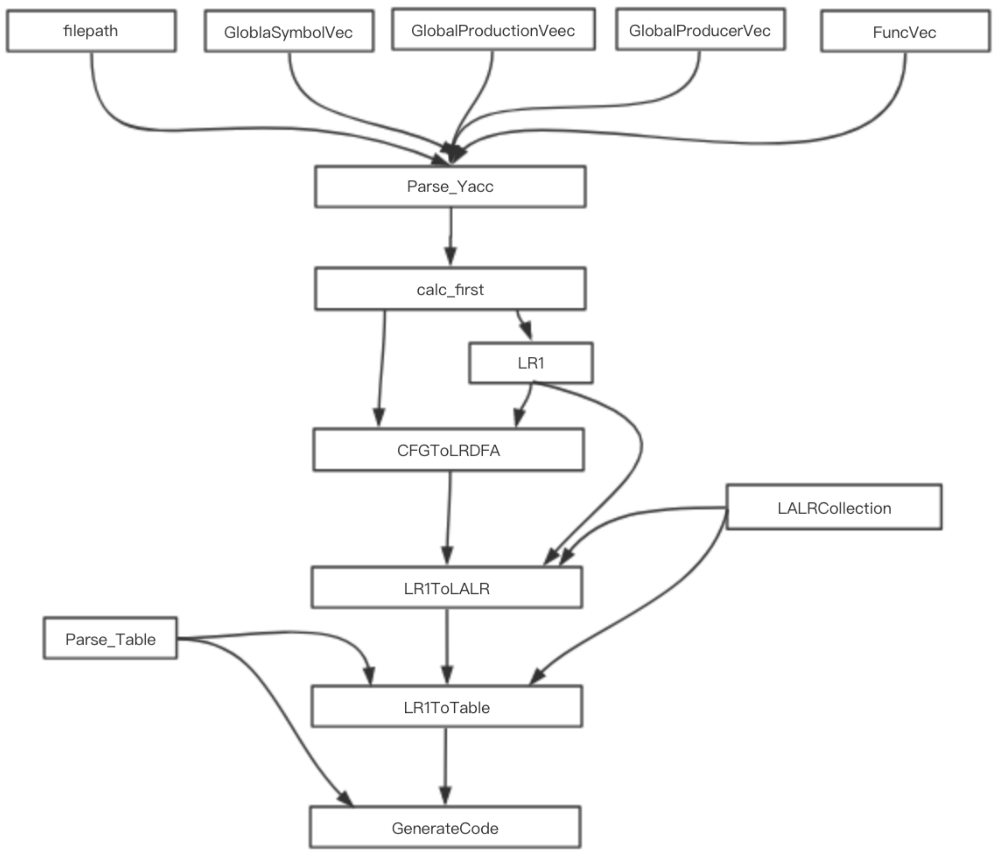
- CodeGeneration.h
  - `void GenerateCode(Parse_Table& p2 )`

根据分析表来，生成 Parser 的代码。

- SEU_Yacc.h
  - `void Parsing()`

总控程序的执行

各个模块之间基本上为顺序执行关系，由 Main.cpp 中的 main (int, char**)调用，其数据流图如下(由于空间限制只展示重要的数据，一些辅助用的变量省略)。



3.2 详细设计

数据结构

- 存储 LR(1)文法产生式标号以及此时点的位置的结构体

```
typedef struct Item {
    int dot_positionInt = 0;       //表示·的位置
    int productionrInt = -1;       //表示表达式文法标号
    set<int> prediction;         //表示预测符标号的 set
    inline bool operator==(const Item& item)const {  //便于判等
        if (dot_positionInt != item.dot_positionInt ||
            productionrInt != item.productionrInt ||
            prediction != item.prediction
                )
            return false;
        else
            return true;
    }
    Item& operator=(const Item&item) {//便于 Item 的直接复制
        dot_positionInt = item.dot_positionInt;
        productionrInt = item.productionrInt;
        prediction.clear();
        prediction.insert(item.prediction.cbegin(),
item.prediction.cend());
        return *this;
    }
    bool operator<(const Item& rhs) const  //进行记录的比较
    {
        if (productionrInt < rhs.productionrInt) {
            return true;
        }
        else {
            if (productionrInt == rhs.productionrInt) {
                if (dot_positionInt < rhs.dot_positionInt) {
                    return true;
                }
            }
        }
        return false;
    }
    Item() {
    }
    Item(int a, int b, set<int>c) {
        dot_positionInt = a;
        productionrInt = b;
        prediction = c;
    }

}Item;
```

我们定义了这个结构体来表示状态中的产生式和对应的终结符以及点的位置，并用集合记录的预测符，方便合并相同产生式、相同点的位置的结构体对象，用序号来表示产生式，节省了空间，提高了效率。

- 存储 LR(1)文法状态的结构体(*仅存储内核项)

```
typedef struct ItemSet {
    int stateInt = -1;                  //状态号
    unordered_map<int, int> edgeMap;     //<字符标号，状态号>
    vector<Item> itemSet;         //项目集内各项目
    inline bool operator==(const ItemSet& BSet)const { //通过内核项判断两个 LR(1)项集是否相同
        int la = itemSet.size(), lb = BSet.itemSet.size();
        if (la != lb)
            return false;
        for (int i = 0; i < la; i++) {
            int j = 0;
            for (; j < lb; j++)
                if (itemSet[i] == BSet.itemSet[j])break;
            if (j == lb)return false;
        }
        return true;
    }
}ItemSet;
```

该结构体用于存储状态，状态通过状态号来区分，其中的 edgesMap 用于表示发出边，对于一个状态来说，每个类型的字符只可能有一种移进，所以使用普通的 unordered_map。而对于 Item，我们用了比较直观的逐项比较的方法来判断是否两个项集相同。

- 存储状态的、用于表示 LR（1）DFA 的 vector

```
typedef vector<ItemSet> Collection;
```

我们默认 0 号状态为初始状态，放入 vector 的 DFA 状态均不相同。

算法

- 读入解析 yacc.y:

yacc.y 由定义部分和规则部分组成:

- ○ token 的定义部分，每行由%token 开始，后面跟上 token 名，如下：

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
```

- ○ %left 表明了运算符号的左结合性，如下：

```
%left '+' '-'
%left '*' '/'
```

表明'*'和'/'和优先级比'+'和'-'的优先级高

- ○ 开始符由%start 开始，表示规则的开始符号，如下：

```
%start translation_unit
```

- ○ %%单独为一行表示规则部分的开始，如下：

```
%%
primary_expression
: IDENTIFIER
| CONSTANT
| STRING_LITERAL
| '(' expression ')'
;
```

在 yacc 程序的规则部分，备用规则是用'|'隔开的，其中的元素有可能是非终结符，token 和用单引号包裹在内的单个字符。由于本设计的目标编译.y 文件中无语义动作，故本设计不处理语义动作。

根据语法规则编写程序将.y 文件的以下信息进行存储：

1. token 符号
2. 结合性信息
3. 开始符号
4. 产生式
5. 附加子程序

- 求 first 集算法

first 的求解过程分为两种情况，一种是求某个符号的 first，另一种是字符串序列的 first。

- ○ 求符号 s 的 first 集 first(s)

1. 如果 s 是终结符，直接返回{s}

2. 如果 s 为 epsilon，返回{-1}（表示 epsilon）

3. 如果 s 是非终结符，对于每一个 s 在左部的产生式 s->$x_1x_2…x_n$

a)遍历产生式右部 s->$x_1x_2…x_n$，对于不是最后一个符号的 x_i ，利用 first_symbol 递归求解 first(x_i)，如果其中包含 epsilon(-1)，去掉它，与原来的 first(s)求并;若不包含 epsilon 则后面的 x_i 不用再计算 first

b)若遍历到最后一个符号 x_n，其中包含 epsilon，将 epsilon(-1)加入 first(s)

○ 求序列 $s_1s_2s_3…s_n$ 的 first 集 first($s_1s_2s_3…s_n$)

1. 遍历 $s_1s_2s_3…s_n$，对于不是最后一个符号的 s_i 求 first(s_i)，如果包含 epsilon(-1)，去掉它，与原来的 first($s_1s_2s_3…s_{i-1}$)求并。
2. 当遍历到最后一个符号 s_n 时，如果如果其中包含 epsilon，将 epsilon(-1)加入 first($s_1s_2s_3…s_n$)。

因为求解 first 的时候利用了递归算法，所以可能会出现因为左递归而导致的栈溢出的问题，我们这里利用的一个集合来记录函数调用路径上的正在处理的非终结符，若当前要求的非终结符已经在 set 中出现便不再进行递归处理，因此防止了左递归的出现，解决了栈溢出的问题。

- 依照上下文无关文法生成 LR(1) DFA

在该算法中，我们仅利用 LR(1)项集的内核项来区分不同的 LR(1)项集，因此我们在设计 LR(1)项的数据结构时，规定一个 LR 项(1)包含所有预测符，这样我们可以通过对点的位置和产生式编号的 pair 的 map 来确定 LR(1)项在 LR(1)项集中的编号，而且保证点的位置加上产生式可以在一个 LR(1)项集中唯一确定一个 LR(1)项，则可以实现判断 LR(1)项集(LR(1)状态)是否相等。

1. 由规定 startProduction 记录了第一个产生式，将其作为初始 LR(1)状态(0 号状态)的内核项，文法结束符号”$r”的序号规定为符号表中最后一个。
2. 将状态号 0 入队，当队列不为空时，从队列中取出未处理状态 s，对其进行如下处理：

a)状态内扩展

1. 新建空队列，并将状态 s 的所有 LR(1)项的标号加入队列
2. 当队列不为空时，取出队列头部的标号，进行如下操作：

a）如果点在 r 最右部，将 r 弹出队列不做任何处理

b）否则取出 r 点右侧的符号 w，若 w 为终结符，将 r 弹出队列不做任何处理

c）否则若 w 为非终结符，则将产生式右侧 s->…r*w..，在 w 及其后的符号放入新

的字符串中，求解该字符串的 first_string 函数调用结果,若 first 集合中存在 epsilono(-1)，则将该编号对应的产生式的所有预测符也加入 first 运算的结果中，作为该非终结符 w 的所有产生式的预测符，并判断相应产生式是否已经在项集中，所在项集中则更新预测符集合，不在项集中则加入项集。

b)状态间扩展

对于一个传入的状态 s，遍历其中所有的 LR 项 r

- a）如果 r 的点在产生式的最后，不做任何操作
- b）否则查看是否处理过点后的符号 s
    - i. 如果处理过 s，将 r 的点右移后并入相应的 LR 项集中
    - ii. 如果没处理过 s，则新建 LR 项集，将 r 的点右移后并入

- 将 LR (1)DFA 转换为 LALR(1)DFA

将 LR（1）文法转化为 LALR（1）文法本质上就是合并同心项。主要算法如下：

1. 找到同心项：
    - a）对于每一个状态：
        - 提取产生式和点的位置作为同心项。
        - 对已有的产生式集合进行遍历，如果该状态同心项与集合中的相同，则直接加入集合，否则新建该类型同心项。
2. 合并同心项：
    - a）根据 1 中得到的同心项集合，遍历每个同心项，将同心项对应的所有产生式取并集。创建一个新的 LR(1)DFA 状态。记录新旧状态号的映射关系。
    - b）以新旧状态号的映射关系对边和初态标号进行映射。

- LALR(1)DFA 预测分析表的生成

本部分的算法思路较为简单，利用同心集的概念，在 LR (1)的 DFA 基础上进行合并。在此就不在赘述。我们生成了一张.csv 格式的分析表，具体可详见程序运行结果。

- yyparse.cpp 文件生成

之后直接向 yyparse 中输出利用预测分析表进行语法分析的总控算法，如下：

1. 定义符号栈和状态栈;

2.　　初始状态压入状态栈，初始符号$r 压入符号栈；

3.　　读取一个 token;

4.　　执行下列步骤直到结束

a)令 item 为根据当前栈顶和读头下 token 查分析表得到的项
b)如果 item>0 即表示移进

i.若 item 为 INT_MAX，则报错。否则继续。

ii.将 item 压入状态栈；

iii.将 token 压入符号栈；

iv.读取一个 token 号；

v.跳转到步骤 a);

c)如果 item<=0 即表示归约

i.判断 p 是否为-1，若为 0 则代表规约 0 号产生式，分析结束。

ii.根据 item 查表得到产生式 p

iii.从符号栈栈顶弹出产生式右部符号数量个符号；

iv.从状态栈栈顶弹出同样数量个状态；

v.把 p 左部和依照 p 左部和栈顶元素查表的结果分别压入符号栈和状态栈；

vi.输出产生式 p；

vii.跳转到步骤 a)；

d)如果 item 为 INT_MAX 表示无法继续分析，报错处理。

- AST 文件生成

1）控制台 AST 的输出：多叉树的前序遍历，利用栈将每个节点的最后一个叶子结点标记，控制输出节点所在的行与行之间的层次关系。

2）PDF 版本 AST 的输出：生成 dot 文件，利用 grapviz 工具自动生成。

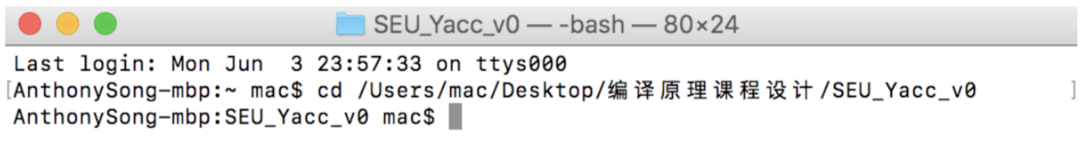
4 使用说明

4.1 SEU_Yacc 使用说明

这里，我们为了缩短编译时间和跨平台下的使用方便，我们为我们的 SEU_Yacc 项目编写了 MakeFile，从而利用 CMake 工具方便了编译运行，缩小了 SEU_Yacc 工具在占的磁盘空间，方便了用户的调试和使用。

下面详细阐述 SEU_Yacc 的使用步骤。

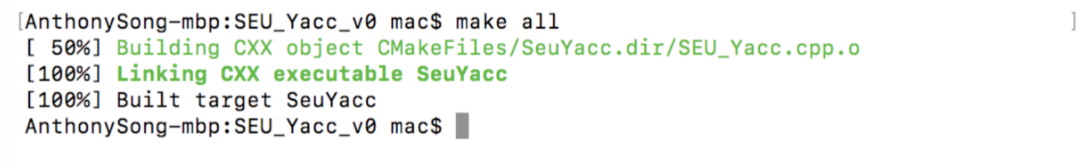
1. 打开终端，切换当前工作目录。如图 4-1-1。



```
Last login: Mon Jun  3 23:57:33 on ttys000
AnthonySong-mbp:~ mac$ cd /Users/mac/Desktop/编译原理课程设计/SEU_Yacc_v0
AnthonySong-mbp:SEU_Yacc_v0 mac$ 
```

图 4-1-1:SEU_Yacc 使用步骤 1

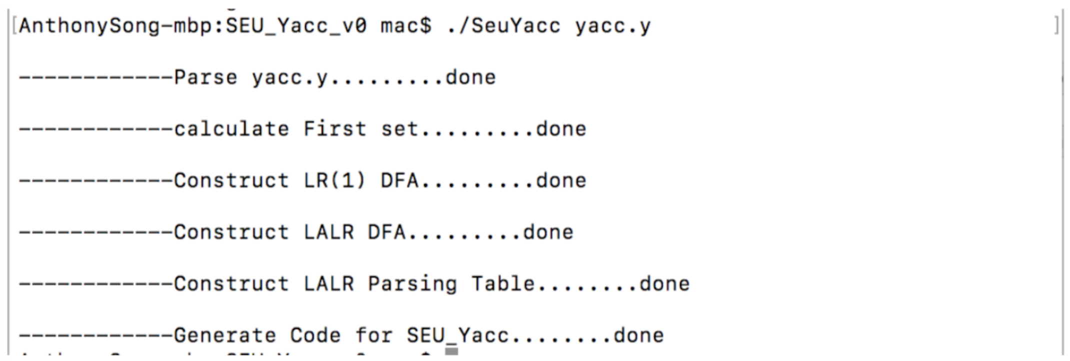
2. 输入 make all，编译项目。



```
AnthonySong-mbp:SEU_Yacc_v0 mac$ make all
[ 50%] Building CXX object CMakeFiles/SeuYacc.dir/SEU_Yacc.cpp.o
[100%] Linking CXX executable SeuYacc
[100%] Built target SeuYacc
AnthonySong-mbp:SEU_Yacc_v0 mac$ 
```

图 4-1-2:SEU_Yacc 使用步骤 2

3. 输入./Seu_Yacc yacc.y，根据 yacc.y 文件，产生语法分析器 **SEU_Yacc** 的 C++代码文件 yyparse.cpp。

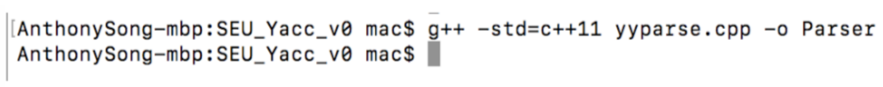


```
AnthonySong-mbp:SEU_Yacc_v0 mac$ ./SeuYacc yacc.y

------------Parse yacc.y.........done

------------calculate First set.........done

------------Construct LR(1) DFA.........done

------------Construct LALR DFA.........done

------------Construct LALR Parsing Table........done

------------Generate Code for SEU_Yacc........done
```

图 4-1-3:SEU_Yacc 使用步骤 3

4. 输入 g++ -std=c++11 yyparse.cpp -o Parser，编译生成语法分析器

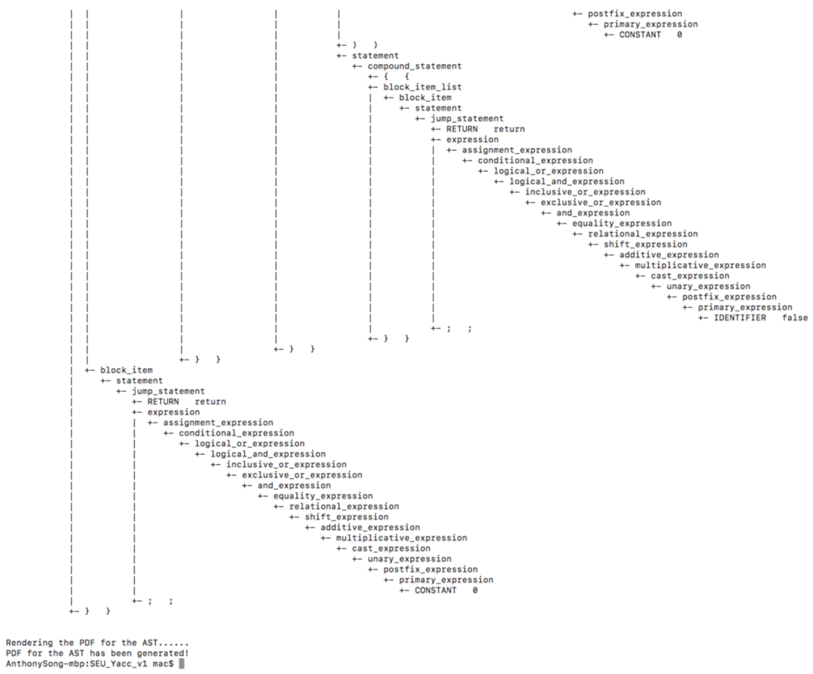
SEU_Yacc 的可执行文件。



```
AnthonySong-mbp:SEU_Yacc_v0 mac$ g++ -std=c++11 yyparse.cpp -o Parser
AnthonySong-mbp:SEU_Yacc_v0 mac$ 
```

图 4-1-4:SEU_Yacc 使用步骤 4

5. 输入 `./Parser Lex_Tokens.txt`，对测试代码文件经过 **SEU_Lex** 得到的 Token 序列进行语法分析，测试 **SEU_Yacc** 的表现。可在控制台看到规约产生式序列和 AST 的可视化表达。并且生成了一张 PDF 版本的 AST，更加直观。这里由于篇幅限制，仅展示部分。



```
        | |           |           |       |                             +- postfix_expression
        | |           |           |       |                               +- primary_expression
        | |           |           |       |                                 +- CONSTANT   0
        | |           |           |       +- )   )
        | |           |           |       +- statement
        | |           |           |         +- compound_statement
        | |           |           |           +- {   {
        | |           |           |           +- block_item_list
        | |           |           |           |   +- block_item
        | |           |           |           |     +- statement
        | |           |           |           |       +- jump_statement
        | |           |           |           |         +- RETURN   return
        | |           |           |           |         +- expression
        | |           |           |           |         | +- assignment_expression
        | |           |           |           |         |   +- conditional_expression
        | |           |           |           |         |     +- logical_or_expression
        | |           |           |           |         |       +- logical_and_expression
        | |           |           |           |         |         +- inclusive_or_expression
        | |           |           |           |         |           +- exclusive_or_expression
        | |           |           |           |         |             +- and_expression
        | |           |           |           |         |               +- equality_expression
        | |           |           |           |         |                 +- relational_expression
        | |           |           |           |         |                   +- shift_expression
        | |           |           |           |         |                     +- additive_expression
        | |           |           |           |         |                       +- multiplicative_expression
        | |           |           |           |         |                         +- cast_expression
        | |           |           |           |         |                           +- unary_expression
        | |           |           |           |         |                             +- postfix_expression
        | |           |           |           |         |                               +- primary_expression
        | |           |           |           |         |                                 +- IDENTIFIER   false
        | |           |           |           |         +- ;   ;
        | |           |           |           +- }   }
        | |           |           +- }   }
        | |           +- }   }
        | +- block_item
        |   +- statement
        |     +- jump_statement
        |       +- RETURN   return
        |       +- expression
        |       | +- assignment_expression
        |       |   +- conditional_expression
        |       |     +- logical_or_expression
        |       |       +- logical_and_expression
        |       |         +- inclusive_or_expression
        |       |           +- exclusive_or_expression
        |       |             +- and_expression
        |       |               +- equality_expression
        |       |                 +- relational_expression
        |       |                   +- shift_expression
        |       |                     +- additive_expression
        |       |                       +- multiplicative_expression
        |       |                         +- cast_expression
        |       |                           +- unary_expression
        |       |                             +- postfix_expression
        |       |                               +- primary_expression
        |       |                                 +- CONSTANT   0
        |       +- ;   ;
        +- }   }

Rendering the PDF for the AST......
PDF for the AST has been generated!
AnthonySong-mbp:SEU_Yacc_v1 mac$ 
```

图 4-1-5: SEU_Yacc 使用步骤 5

图 4-1-6: SEU_Yacc 使用步骤 5

5 测试用例与结果分析

测试用例：

测试代码：

```
int t=0,j=10,tot,ans;
int a[10];

typedef struct node{
unsigned int q;
struct node* next;
}Node;



float add(int x){
static int p1=1;
static double p2=2;
static float p3=3;
p1+=x;
p2*=x;
p3-=x;
do{
        Node* ptrp1;
        if(ptr->next != 0 && 3>=2.0){
                (*ptr).q %= 2;
        }
        else{
                p1 = p2 < 9 ? p2 : p1 * p2;
                continue;
        }
}while(i != ++j);
return 22.2;
}



int foo(int* a,const int *b){
if(a == b || a!= -0.2 && b << 1){
        return sizeof(int);
```

```
}
else{
        foo(&a-&b,&b);
        return ;
}
}

enum week{ Mon = 1, Tues = 2, Wed = 3, Thurs = 4, Fri = 5, Sat = 6, Sun = 7 };

int main(int argc, char const *argv[]){
tot=10;

char a[10]={"a","b","c","a","b","c","a","b","c","a","b","c"};

unsigned a[12];

int (*(*func[7][8][9])(int*))[5];

for(int j = 0 ; j< 12; ++j){
        a[j] += j << 1;
}

for(int i=1;i<=10;i+=1){
        t=t+i;
        while((j>0&&j<i) || i<8)
                tot=tot*t-j/i;
        add(i);
        tot/=2;
}
if(tot==tot&&!(tot!=tot))
        add(tot,para1,& para2);
switch(j){
        case 3:{
                foo(&a,&b);
                break;
        }
        case 4:{
                if(tot==tot&&!(tot!=tot))
```

```
                        add(tot,para1,& para2);
                        break;
            }
            case 5:break;
            default:break;
}

int a[3][4] = {  {0, 1, 2, 3} , {4, 5, 6, 7} ,   {8, 9, 10, 11}   };

printf("%d, %d, %d, %d, %d", sizeof(enum week), sizeof(day), sizeof(Mon), sizeof(Wed),
sizeof(int) );
for (j=0;j<n2;j++)
       {
            (*p)[j].stu=(char **)malloc (sizeof(char)*3);
            if ((*p)[j].stu == 0)
                        {
                        return false;
                        }
            for (k=0;k<3;k++)
            {
                        (*p)[j].stu[k]=(char *)malloc (sizeof(char)*20);
                        if ((*p)[j].stu[k] == 0)
            {
            return false;
            }
            }
       }
return 0;
}
```

SEU_Yacc 分析得到的产生式序列：

```
type_specifier -> INT
declaration_specifiers -> type_specifier
direct_declarator -> IDENTIFIER
declarator -> direct_declarator
primary_expression -> CONSTANT
postfix_expression -> primary_expression
```

```
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
initializer -> assignment_expression
init_declarator -> declarator = initializer
init_declarator_list -> init_declarator
direct_declarator -> IDENTIFIER
declarator -> direct_declarator
primary_expression -> CONSTANT
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
initializer -> assignment_expression
init_declarator -> declarator = initializer
init_declarator_list -> init_declarator_list , init_declarator
```

```
direct_declarator -> IDENTIFIER
declarator -> direct_declarator
init_declarator -> declarator
init_declarator_list -> init_declarator_list , init_declarator
direct_declarator -> IDENTIFIER
declarator -> direct_declarator
init_declarator -> declarator
init_declarator_list -> init_declarator_list , init_declarator
declaration -> declaration_specifiers init_declarator_list ;
external_declaration -> declaration
translation_unit -> external_declaration
type_specifier -> INT
declaration_specifiers -> type_specifier
direct_declarator -> IDENTIFIER
primary_expression -> CONSTANT
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
direct_declarator -> direct_declarator [ assignment_expression ]
declarator -> direct_declarator
init_declarator -> declarator
init_declarator_list -> init_declarator
declaration -> declaration_specifiers init_declarator_list ;
external_declaration -> declaration
translation_unit -> translation_unit external_declaration
storage_class_specifier -> TYPEDEF
```

```
struct_or_union -> STRUCT
type_specifier -> UNSIGNED
type_specifier -> INT
specifier_qualifier_list -> type_specifier
specifier_qualifier_list -> type_specifier specifier_qualifier_list
direct_declarator -> IDENTIFIER
declarator -> direct_declarator
struct_declarator -> declarator
struct_declarator_list -> struct_declarator
struct_declaration -> specifier_qualifier_list struct_declarator_list ;
struct_declaration_list -> struct_declaration
struct_or_union -> STRUCT
struct_or_union_specifier -> struct_or_union IDENTIFIER
type_specifier -> struct_or_union_specifier
specifier_qualifier_list -> type_specifier
pointer -> *
direct_declarator -> IDENTIFIER
declarator -> pointer direct_declarator
struct_declarator -> declarator
struct_declarator_list -> struct_declarator
struct_declaration -> specifier_qualifier_list struct_declarator_list ;
struct_declaration_list -> struct_declaration_list struct_declaration
struct_or_union_specifier -> struct_or_union IDENTIFIER { struct_declaration_list }
type_specifier -> struct_or_union_specifier
declaration_specifiers -> type_specifier
declaration_specifiers -> storage_class_specifier declaration_specifiers
direct_declarator -> IDENTIFIER
declarator -> direct_declarator
init_declarator -> declarator
init_declarator_list -> init_declarator
declaration -> declaration_specifiers init_declarator_list ;
external_declaration -> declaration
translation_unit -> translation_unit external_declaration
type_specifier -> FLOAT
declaration_specifiers -> type_specifier
direct_declarator -> IDENTIFIER
type_specifier -> INT
declaration_specifiers -> type_specifier
```

```
direct_declarator -> IDENTIFIER
declarator -> direct_declarator
parameter_declaration -> declaration_specifiers declarator
parameter_list -> parameter_declaration
parameter_type_list -> parameter_list
direct_declarator -> direct_declarator ( parameter_type_list )
declarator -> direct_declarator
storage_class_specifier -> STATIC
type_specifier -> INT
declaration_specifiers -> type_specifier
declaration_specifiers -> storage_class_specifier declaration_specifiers
direct_declarator -> IDENTIFIER
declarator -> direct_declarator
primary_expression -> CONSTANT
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
initializer -> assignment_expression
init_declarator -> declarator = initializer
init_declarator_list -> init_declarator
declaration -> declaration_specifiers init_declarator_list ;
block_item -> declaration
block_item_list -> block_item
storage_class_specifier -> STATIC
type_specifier -> DOUBLE
declaration_specifiers -> type_specifier
declaration_specifiers -> storage_class_specifier declaration_specifiers
direct_declarator -> IDENTIFIER
declarator -> direct_declarator
primary_expression -> CONSTANT
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
initializer -> assignment_expression
init_declarator -> declarator = initializer
init_declarator_list -> init_declarator
declaration -> declaration_specifiers init_declarator_list ;
block_item -> declaration
block_item_list -> block_item_list block_item
storage_class_specifier -> STATIC
type_specifier -> FLOAT
declaration_specifiers -> type_specifier
declaration_specifiers -> storage_class_specifier declaration_specifiers
direct_declarator -> IDENTIFIER
declarator -> direct_declarator
primary_expression -> CONSTANT
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
```

```
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
initializer -> assignment_expression
init_declarator -> declarator = initializer
init_declarator_list -> init_declarator
declaration -> declaration_specifiers init_declarator_list ;
block_item -> declaration
block_item_list -> block_item_list block_item
primary_expression -> IDENTIFIER
postfix_expression -> primary_expression
unary_expression -> postfix_expression
assignment_operator -> ADD_ASSIGN
primary_expression -> IDENTIFIER
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
assignment_expression -> unary_expression assignment_operator assignment_expression
expression -> assignment_expression
expression_statement -> expression ;
```

```
statement -> expression_statement
block_item -> statement
block_item_list -> block_item_list block_item
primary_expression -> IDENTIFIER
postfix_expression -> primary_expression
unary_expression -> postfix_expression
assignment_operator -> MUL_ASSIGN
primary_expression -> IDENTIFIER
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
assignment_expression -> unary_expression assignment_operator assignment_expression
expression -> assignment_expression
expression_statement -> expression ;
statement -> expression_statement
block_item -> statement
block_item_list -> block_item_list block_item
primary_expression -> IDENTIFIER
postfix_expression -> primary_expression
unary_expression -> postfix_expression
assignment_operator -> SUB_ASSIGN
primary_expression -> IDENTIFIER
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
```

```
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
……
```

(由于规约的产生式序列过多，这里仅展示前面和后面的序列，详细结果可参见程序运行结果。)

```
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
assignment_expression -> unary_expression assignment_operator assignment_expression
expression -> assignment_expression
expression_statement -> expression ;
statement -> expression_statement
block_item -> statement
block_item_list -> block_item
unary_operator -> *
primary_expression -> IDENTIFIER
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
unary_expression -> unary_operator cast_expression
cast_expression -> unary_expression
```

```
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
expression -> assignment_expression
primary_expression -> ( expression )
postfix_expression -> primary_expression
primary_expression -> IDENTIFIER
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
expression -> assignment_expression
postfix_expression -> postfix_expression [ expression ]
postfix_expression -> postfix_expression . IDENTIFIER
primary_expression -> IDENTIFIER
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
```

```
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
expression -> assignment_expression
postfix_expression -> postfix_expression [ expression ]
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
primary_expression -> CONSTANT
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> equality_expression EQ_OP relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
expression -> assignment_expression
```

```
primary_expression -> IDENTIFIER
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
expression -> assignment_expression
jump_statement -> RETURN expression ;
statement -> jump_statement
block_item -> statement
block_item_list -> block_item
compound_statement -> { block_item_list }
statement -> compound_statement
selection_statement -> IF ( expression ) statement
statement -> selection_statement
block_item -> statement
block_item_list -> block_item_list block_item
compound_statement -> { block_item_list }
statement -> compound_statement
iteration_statement -> FOR ( expression_statement expression_statement expression ) statement
statement -> iteration_statement
block_item -> statement
block_item_list -> block_item_list block_item
compound_statement -> { block_item_list }
statement -> compound_statement
iteration_statement -> FOR ( expression_statement expression_statement expression ) statement
statement -> iteration_statement
block_item -> statement
```

```
block_item_list -> block_item_list block_item
primary_expression -> CONSTANT
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
and_expression -> equality_expression
exclusive_or_expression -> and_expression
inclusive_or_expression -> exclusive_or_expression
logical_and_expression -> inclusive_or_expression
logical_or_expression -> logical_and_expression
conditional_expression -> logical_or_expression
assignment_expression -> conditional_expression
expression -> assignment_expression
jump_statement -> RETURN expression ;
statement -> jump_statement
block_item -> statement
block_item_list -> block_item_list block_item
compound_statement -> { block_item_list }
function_definition -> declaration_specifiers declarator compound_statement
external_declaration -> function_definition
translation_unit -> translation_unit external_declaration
```

6 **课程设计总结**（包括设计的总结和需要改进的内容）
本课程设计很好地完成了语法分析程序的所有基本功能。本项目的最大特色在于继承了我们 SEU_Lex 的广而全的特点，对 ANSI 标准 C 文法未做一字修改，并且在测试文件里很好地展示了这种几乎囊括了所有 C 文法的特性的优势，如指针的嵌套（函数指针、指针数组等）、数组下标的嵌套、函数的声明、定义、调用、嵌套等以及 struct、enum 等复合数据结构等等。另外，我们充分考虑到了 SEU_Lex 和 SEU_Yacc 的输入输出的一致性和兼容性，能够将用户编写的任意符合 1999 ANSI 标准 C 文法的程序，作为我们 Lexer 的输入，完成词法分析，得到的输出 Token 序列文件再次直接作为我们 SEU_Yacc 的输入，从而方便了用户，达到了词法分析和语法分析的高度统一结合。 在各个模块到开发伊始，我们商讨了各个部分所需要到数据结构，在参考了学长所定义的数据结构之后，我们充分考虑到程序运行的效率和所消耗的内存空间大小，在做到一定的权衡利弊之后，我们最终确定了我们所认为的最优的数据结构定义，这不但为我们程序的在可接受时间内执行完成提供了保障，也为我们后期的整体调试整合提供了方便。最终，我们将整个 SEU_Yacc 生成 Parser 的运行时间控制在 2 到 3 秒钟。 在最后整合阶段，由于前期对分析表中各数字代表的含义（移进、规约、错误、接收）未讨论充分，结果导致对编号为 0 的产生式始终无法规约，这样的教训是惨痛的，这充分使我们意识到先期讨论的重要性。另外，对一些特殊情形，也要事先充分考虑，以防最后在 debug 的过程中迷失方向。 对于未来改进方向，我们尝试了中间代码生成，但是由于始终无法找到 ANSI 1999 的标准 C 文法的对应的语义动作，所以我们始终摸不着头脑，在网上查找各钟对文法语言的语义动作的有关文献，但是由于时间原因，我们无法很好地完成该部分，所以最终打算将该部分放在课程结束之后，继续开展。
7 **教 师 评 语**
签名：

附：包含源程序、可运行程序、输入数据文件、输出数据文件、答辩所做 PPT 文件、本设计报告等一切可放入光盘的内容的光盘。