编译原理 实验四 文法设计实验

07111802 王梓丞 1120180488

实验目的

本次实验的主要目的是了解程序设计语言的演化过程和相关标准的制定过 程，深入理解与编译实现有关的形式语言理论，熟练掌握文法及其相关的概念， 并能够使用文法对给定的语言进行描述，为后面的词法分析和语法分析做准备。

实验内容

选定 C 语言子集，并使用 BNF 表示方法文法进行描述，要求至少包括 表达式、赋值语句、分支语句和循环语句；或者设计一个新的程序设计语言，并 使用文法对该语言的词法规则和文法规则进行描述。 以上语言定义首先要给出所使用的字母表，在此基础上使用 2 型文法描述语法规则。

个人理解

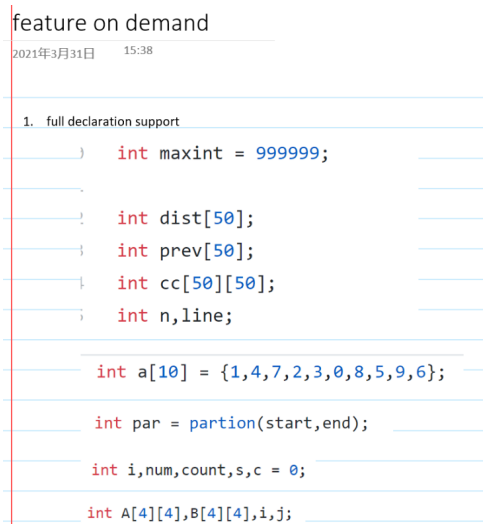
我认为语法规范是程序化语言中最为重要的一环。只有指定了明确的规范并将其写成文档，一个语言的特性、书写方法才能被他人准确的理解。在我看来，文法分析器则是通过规范定义的法则将程序结构化解析，正确解析每一语句“意思”的工具。

实验过程

文法设计

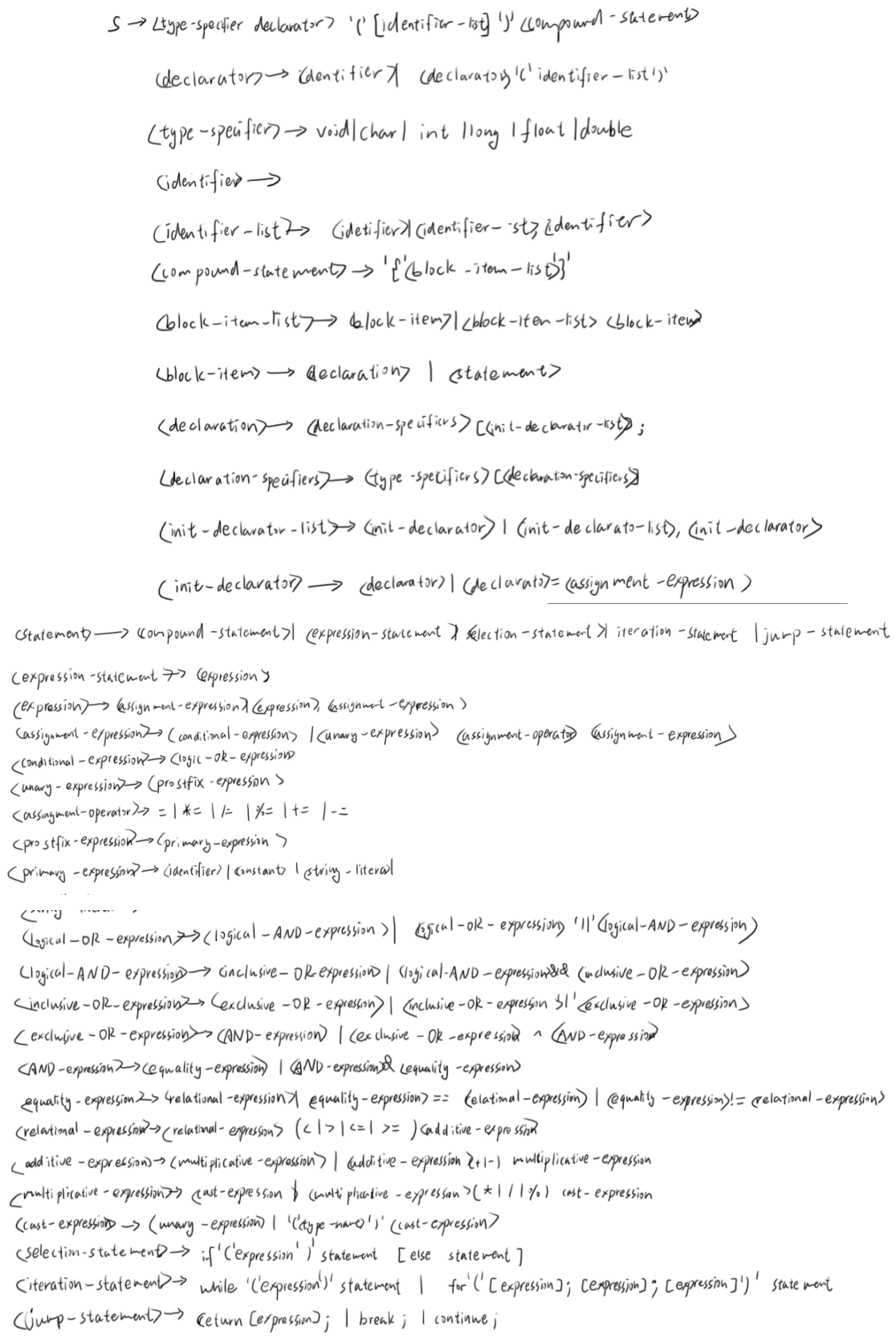
语法特性选择

首先，我通过分析10个C代码的源文件分析出了C语言常见的语句，选定将要实现的语法特性。比如以下声明的特性：



BNF表达式书写

在上一步初步选取了C语言的子集，因此下一步我将更为具体的将法则通过二型文法表示出来：



ANTLR表述的实现

为了验证上述法则是否正确，我将其转换为ANTLR的语法形式将其运行。在运行过程中确实发现了一些需要修正的情况（比如左递归等问题）。经过修正后的ANTLR版的完整表述如下。

表达式 Expression

在表达式这里，通过法则的多级调用可以体现出运算符的优先级顺序

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

Primary Expression

字符串等

```
primary_expression:
    identifier
    |constant
    |string_literal
    |'(' expression ')'
    ;
```

Postfix operators 后缀表达式

```
postfix_expression:
    primary_expression postfix_expression_r?
    | '(' type_name ')' '{' initializer_list '}' postfix_expression_r?
    | '(' type_name ')' '{' initializer_list ',' '}' postfix_expression_r?
    ;
postfix_expression_r:
    | '[' expression ']' postfix_expression?
    | '(' argument_expression_list? ')' postfix_expression?
    | '.' identifier postfix_expression?
    | '->' identifier postfix_expression?
    | '++' postfix_expression?
    | '--' postfix_expression?
    ;
argument_expression_list:
    assignment_expression
    | assignment_expression (',' argument_expression_list)?
    ;
```

Unary operators 单目运算

其中`unary-operator`直接使用`('&'|'*'|'+'|'-'|'~'|'!')`表示

```
postfix_expression
    | '++' unary_expression
    | '--' unary_expression
    | ('&'|'*'|'+'|'-'|'~'|'!') cast_expression
    | 'sizeof' unary_expression
    | 'sizeof' '(' type_name ')'
    ;
```

Cast operators 类型转换

```
cast_expression:
    unary_expression
    | '(' type_name ')' cast_expression
    ;
```

Multicative expression 乘法表达式

```
multiplicative_expression:
    cast_expression
    | cast_expression '*' multiplicative_expression
    | cast_expression '/' multiplicative_expression
    | cast_expression '%' multiplicative_expression
    ;
```

Additive expression 加法表达式

```
additive_expression:
    multiplicative_expression
    | multiplicative_expression '+' additive_expression
    | multiplicative_expression '-' additive_expression
    ;
```

Shift expression 位移表达式

```
shift_expression:
   additive_expression
   | additive_expression '<<' shift_expression
   | additive_expression '>>' shift_expression
    ;
```

Relational expression 关系表达式

```
relational_expression:
    shift_expression
    | shift_expression '<' relational_expression
    | shift_expression '>' relational_expression
    | shift_expression '<=' relational_expression
    | shift_expression '>=' relational_expression
    ;
```

Equality expression 相等关系表达式

```
equality_expression:
    relational_expression
    | relational_expression '==' equality_expression
    | relational_expression '!=' equality_expression
    ;
```

位运算

```
and_expression:
    equality_expression
    | equality_expression '&' and_expression
    ;
exclusive_or_expression:
    and_expression
    | and_expression '^' exclusive_or_expression
    ;
inclusive_or_expression:
    exclusive_or_expression
    | exclusive_or_expression '|' inclusive_or_expression
    ;xxxxxxxxxx and_expression:    equality_expression    | equality_expression
'&' and_expression    ;and_expression:    equality_expression    |
equality_expression '&' and_expression    ;exclusive_or_expression:
and_expression    | and_expression '^' exclusive_or_expression
;inclusive_or_expression:    exclusive_or_expression    | exclusive_or_expression
'|' inclusive_or_expression    ;
```

逻辑运算

```
logical_and_expression:
    inclusive_or_expression
    | inclusive_or_expression '&&' logical_and_expression
    ;
logical_or_expression:
    logical_and_expression
    | logical_and_expression '||' logical_or_expression
    ;
conditional_expression:
    | logical_or_expression
    | logical_or_expression '?' expression ':' conditional_expression
    ;
```

赋值运算

```
assignment_expression:
    conditional_expression
    | unary_expression ('='|'*='|'/='|'%='|'+='|'-='|'<<='|'>>='|'&='|'^='|'|=')
assignment_expression
    ;
```

声明 Declaration

通过分析，我认为通过`const`等关键字对变量实现限制的`type-specifier`、规定变量存储类型的`storage-class-specifier`和`alignment-specifier`这些类型并不常用，因此在`declaration_specifiers`只选用了部分语法进行实现。

`declaration`模块使用antlr有如下表达：

```
declaration:
    declaration_specifiers init_declarator_list?
    ;
declaration_specifiers:
```

```
     type_specifier declaration_specifiers?
    | function_specifier declaration_specifiers?
    ;

init_declarator_list:
    init_declarator (',' init_declarator_list)?
    ;

init_declarator:
    declarator
    | declarator '=' initializer
    ;
```

declarator

由于放弃了对`type-qualifier-list`的支持，因此`declarator`处许多和`type-qualifier-list`都可删除。

这里有一处注意的地方是`direct_declarator`在C语言中是左递归定义的，需要消除其直接左递归，使用`direct_declarator_r`来达成目的。

```
declarator:
    pointer? direct_declarator
    ;
direct_declarator:
    identifier direct_declarator_r?
    | '(' declarator ')' direct_declarator_r?
    ;
direct_declarator_r:
    |'[' assignment_expression ']' direct_declarator_r?
    |'(' parameter_type_list ')' direct_declarator_r?
    |'(' identifier_list? ')' direct_declarator_r?
    ;
pointer:
    '*'*
    ;
parameter_type_list:
    parameter_list
    | parameter_list ',' '...'
    ;
parameter_list:
    parameter_declaration
    | parameter_list ',' parameter_declaration
    ;
parameter_declaration:
    declaration_specifiers declarator
    | declaration_specifiers abstract_declarator?
    ;
identifier_list:
    identifier (',' identifier_list)?
    ;
```

type names

`specifier-qualifier-list`使用`type_name*`作为替代，`direct_abstract_declarator_r`处同样需要消除直接左递归：

```
type_name:
    type_specifier+ abstract_declarator?
    ;
abstract_declarator:
    pointer
    | pointer? direct_abstract_declarator
    ;
direct_abstract_declarator:
    '(' abstract_declarator ')' direct_abstract_declarator_r?
    ;
direct_abstract_declarator_r:
    '[' type_specifier* assignment_expression? ']'direct_abstract_declarator_r?
    |  '[' 'static' type_specifier* assignment_expression
']'direct_abstract_declarator_r?
    |  '[' type_specifier* 'static'  assignment_expression
']'direct_abstract_declarator_r?
    |  '[' '*' ']'direct_abstract_declarator_r?
    |  '(' parameter_type_list? ')'direct_abstract_declarator_r?
    ;
```

Initialization

要点同样是左递归的消除

```
initializer:
    assignment_expression
    |'{' initializer_list '}'
    |'{' initializer_list ',' '}'
    ;
initializer_list:
    designation? initializer initializer_list_r?
    ;
initializer_list_r:
    ',' designation? initializer initializer_list_r?
    ;
designation:
    designator_list '='
    ;
designator_list:
    designator designator_list?
    ;
designator:
    '[' constant_expression ']'
    |'.' identifier
    ;
```

语句 Statement

`Statement`定义了多种语句，涵盖了C语言几乎所有常见的流程。

由于`switch case`等不常用，因此去掉了带标签的`labeled_statement`。

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

compound_statement:
    '{' block_item_list? '}'
    ;
block_item_list:
    block_item block_item_list?
    ;
block_item:
    | declaration
    | statement
    ;
expression_statement:
    expression? ';'
    ;
```

选择语句

舍弃掉`switch`语句的支持

```
selection_statement:
    'if' '(' expression ')' statement
    |'if' '(' expression ')' statement 'else' statement
    ;
```

循环语句（迭代）

```
iteration_statement:
    'while' '(' expression ')' statement
    | 'do' statement 'while' '(' expression ')' ';'
    | 'for' '(' expression? ';' expression? ';' expression? ')' statement
    | 'for' '(' declaration expression? ';' expression? ')' statement
    ;
```

跳转语句

去掉`goto`语句的支持

```
jump_statement:
    'continue' ';'
    | 'break' ';'
    | 'return' expression? ';'
    ;
```

函数和外部定义

这里可以说是C语言的起始部分了，C语言的一切单条语句都能囊括在此：包括函数外的如全局变量等的声明以及函数声明（包括main函数等等）。

```
external_declaration:
    function_definition
    | declaration
    ;
function_definition:
    declaration_specifiers declarator declaration_list? compound_statement
    ;
declaration_list:
    declaration
    | declaration_list declaration
    ;
```

在最后，使用一条语句`external_declaration+`就能使其解析C语言源文件了。

```
start:
    external_declaration+
    ;
```

心得体会

在本次实验中，我有很深的感受。首先，文法的设计比我想象的要复杂许多。为了生成正确的ANTLR版本，我至少反复工作了4次。在本次实验中第一次见识到了真正的语法树，让我对编译器的前端有了更深刻的了解。

从C语言标准中选取可以正常运行的子集是十分困难的，需要很小心的取舍。我认为造成这种难度的原有之一是：语言的标准在指定时并非有着“高内聚，低耦合”的要求，各个语法成分之间结合的十分紧密，稍微弄错一个模块都可能有意想不到的问题出现。

ANTLR在使用过程中出现了一些不太愉快的情况，比如它运行偶尔会很慢，感觉自己需要进一步学习。