实验报告

一、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。

二、实验内容

1. 阅读附件提供的 C 语言和 Java 语言的规范草稿，了解语言规范化定义应包括的具体内容。

2. 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。

三、语言规范理解

语言规范定义了语言的字符集、词法规则、语法规则语义规则。它为不同视角的程序员提供了一致的参考标准。对于应用程序程序员和系统程序员，这个一致的标准使得它们在设计自己层级的程序时可以充分考虑不同层级之间的兼容，使得所有的程序设计人员之间能够有序分工合作，并充分交换理解对方的代码。因此，一种编程语言的标准规范制定是十分重要且必要的。

与自然语言相似，编程语言作为沟通程序员与机器之间的语言，也要遵守相应的编程语言规范。自然语言的沟通对象是人和人。人们在使用自然语言沟通时，即使内容不明确，甚至发生小的错误，绝大多数情况下听者仍然能够理解叙述者所想要表达的内容。编程语言沟通的对象是人和机器。由于计算机对于程序语言的解析严格遵循预定的逻辑，因此程序语言规范相较于自然语言规范会更加严格、精确，对于各个语法成分和语言元素（字符）的定义描述均要完备。如果语言规范设计不严谨，一是会导致对特定的程序段可能出现运行错误，一是对于一些程序段可能出现二义性的问题。

语言的规范直接决定了程序员所编写程序的复杂程度和最终编译产生程序的执行效率。如 C 语言，它的语法相对较为复杂，对于各个语言成分的约束都较为严格且更加贴近底层，因此对程序员要求更高，程序的执行效率也更高。而 Python 语言提供更加贴近自然语言、更加灵活的语法，因此代码可读性更好，程序鲁棒性更强，但执行效率也更低。

同时，对于一种编程语言，随着时间的增长和技术的发展，语言规范也会产生一些或大或小的变化。如 Python2 和 Python3，虽然它们均是 Python 语言，二者在语法上基本保持一致，但是 Python3 相对于 Python2 在语法上有重大改进，使得两个版本的代码相互不兼容。而以 C 语言为例，随着 C 语言的广泛使用和计算机的不断发展，C 语言也不断更新它的语法规范，引入了一些新的关键字和新的语法特性。但是这些新的语言规范对于旧的 C 语言程序仍然保持兼容。因此，在对语言规范进行制定时，也应同时考虑一定的扩展性需求。

四、文法设计（BNF）

使用 BNF 表示方法对文法进行描述。下面给出字母表及使用 2 型文法描述的 C 语言子集文法。

1、字母表

字母表说明了 BNF 描述中所涉及到的所有标识符。V_N 集合中包含了所有的非终结符，$V_{T\text{集合}}$中包含了所有的终结符，其中包含了 C 语言所用到的的关键字、数据、运算符等内容。这些终结符均由词法分析程序对相应的程序单词打标产生，因此词法分析所产生的标识符和终结符中所包含的内容应是完全重合的。

V_N = {primary-expression,identifier,constant,string-literal,expression,generic-selection,generic-assoc-list,postfix-expression,argument-expression-list,type-name,initializer-list,assignment-expression,unary-expression,unary-operator,cast-expression,multiplicative-expression,additive-expression,shift-expression,relational-expression,equality-expression,AND-expression,exclusive-OR-expression,inclusive-OR-expression,logical-AND-expression,logical-OR-expression,conditional-expression,assignment-operator,constant-expression,declaration,declaration-specifiers,init-declaration-list,static_assert-declaration,storage-class-specifier,type-specifier,type-qualifier,function-specifier,alignment-specifier,init-declarator,declarator,initializer,struct-or-union-specifier,enum-specifier,typedef-name,struct-or-union,struct-declaration-list,struct-declaration,specifier-qualifier-list,struct-declarator,enumerator-list,enumerator,enumeration-constant,atomic-type-specifier,pointer,direct-declarator,type-qualifier-list,parameter-type-list,identifier-list,parameter-list,parameter-declaration,abstract-declarator,direct-abstract-declarator,designation,designator_list,statement,labeled-statement,compound-statement,expression-statement,selection-statement,iteration-statement,jump-statement,block-item-list,block-item}

V_T = {identifier,constant,string-literal,(,),_Generic,,,:,default,[,],(,),.,->,++,--,{,},sizeof,&,,+,-,~,!,%,<<,>>,<,>,<=,>=,==,!=,^,|,&&,||,?,:,=,=,/=,%=,-=,<<=,>>=,&=,^=,|=,;,typedef,extern,static,_Thread_local,auto,register,void,char,short,int,long,float,double,signed,unsigned,_Bool,_Complex,struct,union,enum,_Atomic,const,restrict,volatile,inline,_Noreturn,_Alignas,static,...,_Static_assert,case,default,if,else,switch,while,do,fore,goto,continue,break,return}

S = <statement>

P 在下文中列出

2、产生式

按照 C 语言标准对于语法的描述，下面给出所有的产生式。根据这些产生式的属性对他们进行分类，最终确定了三大部分：表达式、声明、语句。

表达式是程序的基本构成，它由关键词按照语法规则进行拼接而形成。这部分说明了程序单词的运算逻辑、赋值语句等的说明。对于各类算数运算、逻辑运算操作，及它们之间的运算优先级和运算先后顺序均由表达式部分给出。此处的运算符优先级实现是通过不同非终结符产生式的嵌套来构成的。从结构上来看，所有的表达式均可归约至非终结符：<assignment-expression>，各个类型的表达式则具有层级的包含关系。这使得外层、高级的表达式所使用的运算符运算优先级是更高的。同时，在设计时保证包含关系为链式包含，不包含树形结构或环状结构，使得各运算符的优先级是确定、容易判定的。但同时这也使得非终结符的包含关系过于深，可能带来性能上的一些影响。

声明是对程序中所涉及到的变量声明、函数定义、结构体定义、枚举类型定义等设计数据结构和程序运行变量的语法说明。此部分的产生式定义严格依照 C 语言的语法要求来进行编写。对于定义，除了对数据结构的定义，还包括对初始化方法的定义；对变量声明，除了声明格式外也包括变量初始化部分的语法定义。

语句是由表达式和关键词构成的程序块。这部分产生式说明了表达式、声明之间的构成组织关系，同时对分支语句、循环语句、跳转语句的语法进行定义说明。分支语句包含 if、switch 语句两种类型；循环语句包含 for 循环、while 循环、do-while 循环三种；跳转语句包含 goto、continue、break、return 四种。

a、表达式

<primary-expression> -> <identifier>|<constant>|<string-literal>|<{expression}>|<generic-selection>

<generic-selection> -> _Generic(<assignment-expression>,<generic-assoc-list>>)

<generic-assoc-list> -> <generic-association>|<generation-assoc-list>,<generic-association>

<generic-association> -> <type-name>:<assignment-expression>|default:<assignment-expression>

<postfix-expression> -> <primary-expression>|<postfix-expression[expression]>|<postfix-expression(argument-expression-list_opt)>|<postfix-expression.identifier>|<postfix_expression->identifer>|<postfix-expression++>|<postfix-expression-->|<(type-name){initializer-list}>|<(type-name){initializer-list,}>

<argument-expression-list> -> <assignment-expression>|<argument-expression-list,assignment-expression>

<unary-expression> -> <postfix-expression>|<++unary-expression>|<--unary-expression>|<unary-operator><cast-expression>|sizeof<unary-expression>|sizeof(<type-name>)|_Alignof(<type-name>)

<unary-operator> -> &|*|+|-|~|!

<cast-expression> -> <unary-expression>|<(type-name)><cast-expression>

<multiplicative-expression> -> <cast-expression>|<multiplicative-expression>*<cast-expression>|<multiplicative-expression>/<cast-expression>|<multiplicative-expression>%<cast-expression>>

<additive-expression> -> <multiplicative-expression>|<additive-expression>+<multiplicative-expression>|<additive-expression>-<multiplicative-expression>

<shift-expression> -> <additive-expression>|<shift-expression><<<additive-expression>|<shift-expression>>><additive-expression>

<relational-expression> -> <shift-expression>|<relational-expression><<shift-expression>|<relational-expression>><shift-expression>|<relational-expression><=<shift-expression>|<relational-expression>>=<shift-expression>|

<equality-expression> -> <relational-expression>|<equality-expression>==<relational-expression>|<equality-expression>!=<relational-expression>

<AND-expression> -> <equality-expression>|<AND-expression>&<equality-expression>

<exclusive-OR-expression> -> <AND-expression>|<exclusive-OR-expression>^<AND-expression>

<inclusive-OR-expression> -> <exclusive-OR-expression>|<inclusive-OR-expression>\|<exclusive-OR-expression>

<logical-AND-expression> -<inclusive-OR-expression>|<logical-AND-expression>&&<inclusive-OR-expression>

<logical-OR-expression> -> <logical-AND-expression>|<logical-OR-expression>||<logical-AND-expression>

<conditional-expression> -> <logical-OR-expression>|<logical-OR-expression>?<expression><conditional-expression>

<assignment-expression> -> <conditional-expression>|<unary-expression><assignment-operator><assignment-expression>

<assignment-operator> -> =|*=|/=|+=|-=|<<=|>>=|&=|^=||=

<expression> -><assignment-expression>|<expression>,<assignment-expression>

<constant-expression> -> <conditional-expression>

b、声明

<declaration> -> <declaration-specifiers><init-declaration-list_opt>;|<static_assert-declaration>

<declaration-specifiers> -> <storage-class-specifier><declaration-specifiers_opt>|<type-specifier><declaration-specifiers_opt>|<type-qualifier><declaration-specifiers_opt>|<function-specifier><declaration-specifiers_opt>|<alignment-specifier><declaration-specifiers_opt>

<init-declarator-list> -> <init-declarator>|<init-declarator-list>,<init-declarator>

<init-declarator> -> <declarator>|<declarator>=<initializer>

<storage-class-specifier> -> typedef|extern|static|_Thread_local|auto|register

<type-specifier> -> void|char|short|int|long|float|double|signed|unsigned|_Bool|_Complex|<atomic-type-specifier>|<struct-or-union-specifier>|<enum-specifier>|<typedef-name>

<struct-or-union-specifier> -> <struct-or-union><identifier_opt>{<struct-declaration-list>}|<struct-or-union><identifier>

<struct-or-union> -> struct|union

<struct-declaration-list> -> <struct-declaration>|<struct-declaration-list><struct-declaration>

<struct-declaration> -> <specifier-qualifier-list><struct-declarator-list_opt>;|<static_assert-declaration>

<specifier-qualifier-list> -> <type-specifier><specifier-qualifier-list_opt>|<type-qualifier><specifier-qualifier-list_opt>

<struct-declarator> -> <declarator>|<declarator_opt>:<constant-expression>

<enum-specifier> -> enum<identifier_opt>{<enumerator-list>}|enum<identifier_opt>{<enumerator-list>,}|enum<identifier>

<enumerator-list> -> <enumerator>|<enumerator-list>,<enumerator>

<enumerator> -> <enumeration-constant>|<enumeration-constant>=<constant-expression>

<atomic-type-specifier> -> _Atomic(<type-name>)

<type-qualifier> -> const|restrict|volatile|_Atomic

<function-specifier> -> inline|_Noreturn

<alignment-specifier> -> _Alignas(<type-name>)|_Alignas(<constant-expression>)

<declarator> -> <pointer_opt><direct-declarator>

<direct-declarator> -> <identifier>|(<declarator>)|<direct-declarator>[<type-qualifier-list_opt><assignment-expression_opt>]|<direct-declarator>[static<type-qualifier-list_opt><assignment-expression>]|<direct-declarator>[<type-qualifier-list> static <assignment-expression>]|<direct-declarator>[tpye-qualifier-list_opt*]|<direct-declarator>(<parameter-type-list>)|<direct-declarator>(<identifier-list_opt>)

<pointer> -> *<type-qualifier-list_opt>|*<type-qualifier-list_opt><pointer>

<type-qualifier-list> -> <type-qualifier>|<type-qualifier-list><type-qualifier>

<parameter-type-list> -> <parameter-list>|<parameter-list>,...

<parameter-list> -> <parameter-declaration>|<parameter-list>,<parameter-declaration>

<parameter-declaration> -> <declaration-specifiers><declarator>|<declaration-specifiers><abstract-declarator_opt>

<identifier-list> -> <identifier>|<identifier-list>,<identifier>

<type-name> -> <specifier-qualifier-list><abstract-declarator_opt>

<abstract-declarator> -> <pointer>|<pointer_opt><direct-abstract-declarator>

<direct-abstract-declarator> -> (<abstract-declarator>)|<direct-abstract-declarator_opt>[<type-qualifier-list_opt><assignment_expression_opt>]|<direct-abstract-declarator_opt>[static<type-qualifier-list_opt><assignment-expression>]|<direct-abstract-declarator_opt>[<type-qualifier-list>static<assignment-expresion>]|<direct-abstract-declarator_opt>[*]|<direct-abstract-declarator_opt>(<parameter-type-list_opt>)

<typedef-name> -> <identifier>

<initializer> -> <assigment-expression>|{<initializer-list>}|{<initializer-list>,}

<initializer-list> -> <designation_opt><initializer>|<initializer-list>,<designation_opt><initializer>

<designation> -> <designator_list>=

<designation-list> -> <designator>|<designator-list><designator>

<designator> -> [<constant-expresion>]|.<identifier>

<static_assert-declaration> -> _Static_assert(<constant-expression>,<string-literal>);

c、语句

<statement> -> <labeled-statement>|<compound-statement>|<expression-statement>|<selection-statement>|<iteration-statement>|<jump-statement>

<labeled-statement> -> <identifier>:<statement>|case<constant-expression>:<statement>|default:<statement>

<compound-statement> -> {<block-item-list_opt>}

<block-item-list> -> <block-item>|<block-item-list><block-item>

<block-item> -> <declaration>|<statement>

<expression-statement> -> <expression_opt>;

<selection-statement> ->
if(<expression>)<statement>|if(<expression>)<statement> else
<statement>|switch(<expression>)<statement>

<iteration-statement> ->
while(<expression>)<statement>|do<statement>while(<expression>);|for(expression
_opt;expression_opt;expression_opt)<statement>|for(decleartion
expression_opt;expression)<statement>

<jump-statement> -> goto<identifier>;|continue;|break;|return expression_opt;

五、心得体会

本次实验中，我参考 C 语言规范，用 BNF 范式描述了 C 语言子集的语法。在设计中包含了对表达式、赋值语句、结构体定义语句、函数声明语句、变量声明语句、分支语句、循环语句的说明。通过阅读 C 语言规范，我对程序语言语法标准的描述设计思路有了一定的认识。以不同运算符的优先级为例，通过设计不同的非终结符调用等级可以实现对不同运算符的优先运算顺序。同时，可以通过加深包含层级、消除树形结构来消除二义性可能带来的歧义问题。语言规范的设计要考虑多方面的因素，不仅是对语言的符号、数据结构组织设计，而且还要考虑语言规范是否易用等使用方面的问题。在语言规范描述上使用 BNF 范式是较为规范且易于理解的。但在设计方面也有一些技巧，从而使描述更加精准、简洁。

通过这次实验，我对使用 BNF 范式描述语法规则有了更深的认识。对于 C 语言的语言规范也有了更加深入的了解。语言规范直接决定了程序语言的语法，也决定了一种程序语言的是否易用、是否易于理解和学习等特征。同时语法规则也影响着编译后续环节的进行，间接决定了程序语言的执行效率。进一步方面，可以使用 ANTLRWorks 等工具对设计的语法规则进行实现，并使用编程语言对语法分析器进行实现。并可以结合已有的词法分析器，进行联调测试，进一步完善自己的编译器程序。