编译原理 Lab4: C 语言语法文法设计与验证实验

郑子帆 1120200822

北京理工大学 计算机学院 07112002 班

日期：2023 年 4 月 6 日

摘　　要

本文为北京理工大学《编译原理与设计 2023》课程的 Lab4 实验报告。在本文中，我们通过阅读学习了 C11 的文法设计，自己用 BNF 表示法设计简单的 C 语言的语法文法设计，并用简单的 C 语言程序作为测试程序以验证我们的设计的正确性。

1　实验简介

1.1　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。

1.2　实验内容

本次实验需要依次完成以下三项内容:

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。
3. 根据自己定义的文法子集，推导出附录A的程序。

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。

2 实验准备

2.1 学习 C11 语言规范

本实验中我主要通过 C11 语言规范手册 [1] 设计了 C 语言中一些简单的语言规范的 BNF 表达式。

该手册的第 6 章对于 C 语言的语言规范进行了介绍，其每节内容大意如下：

- 6.1 标注了每个记号的含义
- 6.2 介绍了标识符 (identifiers)、对象 (object) 和类型 (type)
- 6.3 介绍了运算符对操作数的类型转换
- 6.4 介绍了各属性字的分类和词法规则描述
- 6.5 介绍了表达式的语法规则和语义规则
- 6.6-6.10 介绍了其他一些指令、语句的语法规则和解释声明

同时，手册的 Annex 附录对正文中的一些定义、描述进行了进一步更详细的补充说明与解释。

2.2 测试程序的选择

根据实验文档 [2]，我使用了文档末的实例代码作为测试代码以验证我们自己设计的文法的正确性，具体的代码可见附录A。

3 实验过程——文法设计

对于字符表，由于非终结符号过多，在这里我们不一一列举，我们将所有的非终结符号用"<" 和">" 包含，比如 <primary-expression> 等等。另外，标识符 (identifier)、常量 (constant) 和字符串 (string-literal) 也被视为非终结符号。

而终结符号主要有关键字 (Keywords)、标点符号 (Punctuators)、数字 (digit)、字母 (alpha)、类型名 (type) 等等。下面我们根据 C11 语言规范手册设计了文法的 BNF 表示。

3.1 表达式

```
<primary-expression>::=
    <identifier>
    | <constant>
    | <string-literal>
    | "(" <expression> ")"
```

```
<postfix-expression>::=
    <primary-expression>
    | <postfix-expression> "[" <expression> "]"
    | <postfix-expression> "(" <argument-expression-list> ")"
    | <postfix-expression>"(" ")"
    | <postfix-expression> "." <identifier>
    | <postfix-expression> "->" <identifier>
    | <postfix-expression> "++"
    | <postfix-expression> "--"
    | (<type-name>) "{" <initializer-list> "}"
    | (<type-name>) "{" <initializer-list> "," "}"

<unary-expression> ::=
    <postfix-expression>
    | "++" <unary-expression>
    | "--" <unary-expression>
    | ("&"|"*"|"+"|"-"|"~"|"!") <cast-expression>
    | "sizeof" <unary-expression>
    | "sizeof" "(" <type-name> ")"

<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>
```

3.2 声明

下面被"[""]" 所包含的内容为非必需。

```
<declaration> ::=
    <declaration-specifiers> [<init-declarator-list>]

<declaration-specifiers> ::=
    <storage-class-specifier> [<declaration-specifiers>]
    | <type-specifier> [<declaration-specifiers>]
    | <type-qualifier> [<declaration-specifiers>]
    | <function-specifier> [<declaration-specifiers>]

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

<type-specifier> ::=
    "void"|"char"|"short"|"int"|"long"|"float"
    | "double"|"signed"|"unsigned"

<type-qualifier> ::=
```

```
    "const" | "restrict" | "volatile" | "_Atomic"

<function-specifier> ::=
    "inline" | "_Noreturn"

<init-declarator-list> ::=
    <init-declarator>
    | <init-declarator-list> "," <init-declarator>

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

<declarator> ::=
    [<pointer>] <direct-declarator>

<initializer> ::=
    <assignment-expression>
    | "{" <initializer-list> "}"
    | "{" initializer-list> "," "}"

<pointer> ::=
    "*" [<type-qualifier-list>]
    | "*" [<type-qualifier-list>] <pointer>

<type-qualifier-list> ::=
    <type-qualifier>
    | <type-qualifier-list> <type-qualifier>
```

3.3 语句

语句包含五个部分，如下：

```
<statement> ::=
    <labeled-statement>
    | <compound-statement>
    | <expression-statement>
```

```
    | <selection-statement>
    | <iteration-statement>
    | <jump-statement>
```

各部分的具体定义如下:

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

<compound-statement> ::=
    "{" [<block-item-list>] "}"

<block-item-list>::=
    <block-item>
    | <block-item-list> <block-item>

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

<expression-statement> ::=
    [<expression>] ";"

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

<iteration-statement> ::=
    "while" "(" <expression> ")" <statement>
    | "do" <statement> "while" "(" <expression> ")" ";"
    | "for" "(" [<expression>] ";" [<expression>] "; [<expression>] ")" <
    | "for" "(" <declaration> [<expression>] ";" [<expression>] ")" <stat

<jump-statement> ::=
    "goto" <identifier> ";"
```

```
    | "continue" ";"
    | "break" ";"
    | "return" [<expression>] ";"
```

3.4 外部定义 (External definitions)

```
<translation-unit> ::=
    <external-declaration>
    | <translation-unit> <external-declaration>

<external-declaration> ::=
    <function-definition>
    | <declaration>

<function-definition> ::=
    <declaration-specifiers> <declarator> <declaration-list> <compound-st
    | <declaration-specifiers> <declarator> <compound-statement>

<declaration-list> ::=
    <declaration>
    | <declaration-list> <declaration>
```

4 文法推导

根据上面的文法规则，我们可以将附录A中的最简 C 语言程序通过最左推导方法进行推导，如下：

```
<translation-unit>
=> <external-declaration>
=> <function-definition>
=> <declaration-specifiers> <declarator> <declaration-list>
    <compound-statement>
=> <type-specifier> <declarator> <declaration-list> <compound-statement>
=> int <declarator> <compound-statement>
=> int <direct-declarator> <compound-statement>
=> int <direct-declarator> ( <parameter-type-list> ) <compound-statement>
=> int fun ( <parameter-type-list> ) <compound-statement>
```

```
=> int fun ( <parameter-list> ) <compound-statement>
=> int fun ( <parameter-declaration> ) <compound-statement>
=> int fun ( <declaration-specifiers> <declarator> ) <compound-statement>
=> int fun ( <type-specifier> <declarator> ) <compound-statement>
=> int fun ( int <declarator> ) <compound-statement>
=> int fun ( int a ) <compound-statement>
=> int fun ( int a ) { <block-item-list> }
=> int fun ( int a ) { <block-item-list> <block-item> }
=> int fun ( int a ) { <block-item-list> <block-item> <block-item> }
=> int fun ( int a ) { <block-item> <block-item> <block-item> }
=> int fun ( int a ) { <declaration> <block-item> <block-item> }
=> int fun ( int a ) { <declaration-specifiers> <init-declarator-list> ;
=> int fun ( int a ) { <type-specifier> <init-declarator-list> ;
    <block-item> <block-item> }
=> int fun ( int a ) { int <init-declarator-list> ;
    <block-item> <block-item> }
=> int fun ( int a ) { int <init-declarator> ; <block-item> <block-item>
=> int fun ( int a ) { int <declarator> ; <block-item> <block-item> }
=> int fun ( int a ) { int b ; <block-item> <block-item> }
=> int fun ( int a ) { int b ; <statement> <block-item> }
=> int fun ( int a ) { int b ; <expression-statement> <block-item> }
=> int fun ( int a ) { int b ; <expression> ; <block-item> }
=> int fun ( int a ) { int b ; <assignment-expression> ; <block-item> }
=> int fun ( int a ) { int b ; <unary-expression> <assignment-operator>
    <assignment-expression> ; <block-item> }
=> int fun ( int a ) { int b ; <postfix-expression> <assignment-operator>
    <assignment-expression> ; <block-item> }
=> int fun ( int a ) { int b ; <primary-expression> <assignment-operator>
    <assignment-expression> ; <block-item> }
=> int fun ( int a ) { int b ; <identifier> <assignment-operator>
    <assignment-expression> ; <block-item> }
=> int fun ( int a ) { int b ; b <assignment-operator>
    <assignment-expression> ; <block-item> }
=> int fun ( int a ) { int b ; b = <assignment-expression> ;
    <block-item> }
=> int fun ( int a ) { int b ; b = <conditional-expression> ;
```

```
    <block-item> }
=> int fun ( int a ) { int b ; b = <logical-OR-expression> ;
    <block-item> }
=> int fun ( int a ) { int b ; b = <logical-AND-expression> ;
    <block-item> }
=> int fun ( int a ) { int b ; b = <inclusive-OR-expression> ;
    <block-item> }
=> int fun ( int a ) { int b ; b = <exclusive-OR-expression> ;
    <block-item> }
=> int fun ( int a ) { int b ; b = <AND-expression> ; <block-item> }
=> int fun ( int a ) { int b ; b = <equality-expression> ; <block-item> }
=> int fun ( int a ) { int b ; b = <relational-expression> ; <block-item>
=> int fun ( int a ) { int b ; b = <shift-expression> ; <block-item> }
=> int fun ( int a ) { int b ; b = <additive-expression> ; <block-item> }
=> int fun ( int a ) { int b ; b = <additive-expression> +
    <multiplication-expression> ; <block-item> }
=> int fun ( int a ) { int b ; b = <multiplication-expression> +
    <multiplication-expression> ; <block-item>}
=> int fun ( int a ) { int b ; b = <cast-expression> +
    <multiplication-expression> ; <block-item> }
=> int fun ( int a ) { int b ; b = <unary-expression> +
    <multiplication-expression> ; <block-item> }
=> int fun ( int a ) { int b ; b = <primary-expression> +
    <multiplication-expression> ; <block-item> }
=> int fun ( int a ) { int b ; b = <identifier> +
    <multiplication-expression> ; <block-item> }
=> int fun ( int a ) { int b ; b = a + <multiplication-expression> ;
    <block-item> }
=> int fun ( int a ) { int b ; b = a + <cast-expression> ; <block-item> }
=> int fun ( int a ) { int b ; b = a + <unary-expression> ; <block-item>
=> int fun ( int a ) { int b ; b = a + <primary-expression> ;
    <block-item> }
=> int fun ( int a ) { int b ; b = a + <constant> ; <block-item> }
=> int fun ( int a ) { int b ; b = a + 10 ; <block-item> }
=> int fun ( int a ) { int b ; b = a + 10 ; <statement> }
=> int fun ( int a ) { int b ; b = a + 10 ; <jump-statement> }
```

```
=> int fun ( int a ) { int b ; b = a + 10 ; return <expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return
    <assignment-expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return
    <conditional-expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return
    <logical-OR-expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return
    <logical-AND-expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return
    <inclusive-OR-expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return
    <exclusive-OR-expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return <AND-expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return
    <equality-expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return
    <relational-expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return <shift-expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return
    <additive-expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return
    <multiplication-expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return <cast-expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return <unary-expression> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return <postfix-expression> ;
=> int fun ( int a ) { int b ; b = a + 10 ; return <primary-expression> ;
=> int fun ( int a ) { int b ; b = a + 10 ; return <identifier> ; }
=> int fun ( int a ) { int b ; b = a + 10 ; return b ; }
```

5 实验心得

本次实验通过阅读和学习 C11 的语言规范和自己设计文法，我对于文法设计有了更深的理解，并将课上学习到的理论知识和实际的对一种编程语言设计文法相融合到

了一起。这也为我后面的语法分析、语义分析打下了很好的基础。

参考文献

[1] *C11-N1570.pdf*. zh. 2023.

[2] *Lab4-C 语言语法文法设计与验证实验.pdf*. zh. 2023.

A 测试程序

```
int fun(int a){
    int b;
    b = a + 10;
    return b;
}
```