

Report 1 — April 12

Lecturer: Cui Huimin

Completed by: Yao Yongzhou, Wang Zeli, Zhang Jiawei

1.1 实验目标

1. 了解扩展巴科斯范式 (EBNF) 规范。
2. 正确运用扩展巴科斯范式描述计算机文法。
3. 熟悉词法、语法规则的区别,能作出正确区分。
4. 掌握一种实现编译器词法-语法部分的方式、工具。

1.2 实验内容

1. 熟悉 ANTLR 的安装和使用。
 - (a) 搭建 ANTLR 环境。
 - (b) 正确运行课程提供的 demo。
2. 完成词法和语法分析。
 - (a) 根据 CACT 文法规范编写 ANTLR 文法文件 (.g4),并通过 ANTLR 生成 CACT 的词法-语法分析。
 - (b) 覆写 ANTLR 默认的文法错误处理机制,能检查出源码中的词法语法错误。

1.3 实验过程

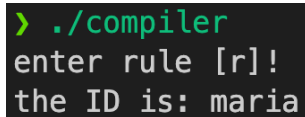
1.3.1 环境搭建

克隆好仓库之后,先将 `test.hello` 中内容补全为 `hello maria`,然后依照其中 `README.md` 中的说明,在终端执行如下指令:

```
1 # 在grammar目录下
2 java -jar ../deps/antlr-4.13.1-complete.jar -Dlanguage=Cpp Hello.g4 -visitor
   -no-listener
3
4 # 在项目根目录下
5 mkdir -p build
6 cd build
```

```
7  cmake ..
8  make
9
10 # 待编译器编译完成后, 执行
11 ./compiler
```

输出内容如下图所示:



```
> ./compiler
enter rule [r]!
the ID is: maria
```

图 1.1. hello maria 输出

1.3.2 文法文件编写

在 `grammar` 目录下, 创建 `CactLexer.g4` 和 `CactParser.g4` 两个文件, 分别用于词法分析和语法分析。

词法分析——`CactLexer.g4`

首先声明这是个词法分析器:

```
1  lexer grammar CactLexer;
```

随后按照讲义内容定义关键字, 再按照优先级顺序定义运算符及其他符号:

```
1  // Keywords
2  Const: 'const' ;
3  Int: 'int' ;
4  Float: 'float' ;
5  Char: 'char' ;
6  Void: 'void' ;
7  If : 'if';
8  Else : 'else';
9  While : 'while';
10 Break : 'break';
11 Continue : 'continue';
12 Return : 'return';
13
14 // Operators by precedence
15 // Priority 1
16 LeftBracket : '[' ;
17 RightBracket : ']' ;
```

```
18 LeftParenthesis : '(' ;
19 RightParenthesis : ')' ;
20
21 // Operators, ranked from highest to lowest priority
22 // Priority 2
23 ExclamationMark: '!';
24 // Priority 3
25 Asterisk: '*';
26 Slash: '/';
27 Percent: '%';
28 // Priority 4
29 // Binary Plus and Minus are also used as unary operators
30 Minus: '-';
31 Plus: '+';
32 // Priority 5
33 Less: '<';
34 LessEqual: '<=' ;
35 Greater: '>';
36 GreaterEqual: '>=' ;
37 // Priority 6
38 LogicalEqual: '==' ;
39 NotEqual: '!=' ;
40 // Priority 7
41 LogicalAnd: '&&' ;
42 // Priority 8
43 LogicalOr: '||' ;
44
45 // Assignment, comma does not act as operator
46 // The use of commas is limited to variable constant declarations, initial
    values, function declarations, and calls
47
48 // Other tokens
49 LeftBrace : '{' ;
50 RightBrace : '}' ;
51 Equal: '=' ;
52 Comma: ',' ;
53 Semicolon: ';' ;
```

接下来定义词法规则,包括标识符、常量、注释、空白字符,也都是讲义中的内容:

```
1 // Identifier and constants
2 // CACT identifiers can consist of uppercase and lowercase letters, numbers, and
    underscores, but must begin with a letter or underscore
3 Identifier: [a-zA-Z_][a-zA-Z0-9_]* ;
```

```
4 // IntConst → DecimalConst | OctalConst | HexadecConst
5 IntegerConstant: ('0' | [1-9][0-9]* | '0'[0-7]+ | '0'[xX][0-9a-fA-F]+);
6 // CharConst → ""character""
7 CharacterConstant: '\' ( EscapeSequence | ~['\\] ) '\';
8 fragment EscapeSequence: '\\' ['"?\\abfnrtv0];
9 // FloatConst
10 FloatConstant: ([-]? '.'[0-9]+ |
11                [-]?[0-9]+ '.'[0-9]* |
12                [-]? '.'[0-9]+ [eE][+-]?[0-9]+ |
13                [-]?[0-9]+('.'[0-9]*)?[eE][+-]?[0-9]+ ) [fF];
14
15 // Comments and white spaces
16 LineComment: '//' ~[\r\n]* -> skip;
17 BlockComment: '/*' .*? '*/' -> skip;
18 NewLine: ('\r' ('\n')? | '\n') -> skip;
19 WhiteSpaces : (' ' | '\t')+ -> skip;
```

下面对词法规则中的内容稍加解释:

1. 标识符规则

Identifier 规则定义了 CACT 语言中标识符的格式:必须以字母或下划线开头,后面可以跟任意数量的字母、数字或下划线。

2. 常量规则

代码定义了三种基本常量类型:

(a) **IntegerConstant** 支持三种表示方式:

- 十进制: **0** 或以非零数字开头的数字序列
- 八进制:以 **0** 开头的八进制数字序列
- 十六进制:以 **0x** 或 **0X** 开头的十六进制数字序列

(b) **CharacterConstant** 定义了字符常量,格式为单引号括起的单个字符,支持转义序列:

- 普通字符:任何非单引号和反斜杠的字符
- 转义字符:由 **EscapeSequence** 片段定义,包括常见转义序列如 **\n**、**\t** 等

(c) **FloatConstant** 定义了浮点常量,必须以 **f** 或 **F** 结尾,支持多种格式:

- 可选负号加小数点加数字(如 **- .123f**)
- 可选负号加数字加小数点加可选数字(如 **-42. f**)
- 带指数的浮点数表示,使用 **e** 或 **E** 表示指数部分

3. 注释和空白字符

(a) 单行注释:以 **//** 开头,直到行尾的所有内容

(b) 块注释: 以 /* 开头, 以 */ 结尾的所有内容

(c) 换行符: 支持 CR、LF 或 CRLF 格式

(d) 空白字符: 包括空格和制表符

所有这些注释和空白字符都使用 `-> skip` 指令, 表示词法分析器在识别它们后会跳过。

语法分析——CactParser.g4

同样先声明这是个语法分析器, 然后使用 `CactLexer` 中定义的词法规则:

```
1  parser grammar CactParser;
2
3  options {
4  tokenVocab=CactLexer;
5  }
```

然后是声明和定义部分:

```
1  // declaration & definition
2  // 编译单元: CompUnit → [ CompUnit ] ( Decl | FuncDef )
3  compilationUnit: (declaration | functionDefinition)*;
4  // 声明: Decl → ConstDecl | VarDecl
5  declaration: constantDeclaration | variableDeclaration;
6  // 常量声明: ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'
7  constantDeclaration: Const basicType constantDefinition (Comma
    constantDefinition)* Semicolon;
8  // 基本类型: BType → 'int' | 'float' | 'char'
9  basicType: Int | Float | Char;
10 // 常量定义: ConstDef → Ident { '[' IntConst ']' } '=' ConstInitVal
11 constantDefinition: Identifier (LeftBracket IntegerConstant RightBracket)* Equal
    constantInitializationValue;
12 // 初始值: ConstInitVal → ConstExp | '{' [ ConstInitVal { ',' ConstInitVal } ] '}'
13 constantInitializationValue: constantExpression | LeftBrace
    (constantInitializationValue (Comma constantInitializationValue)*)?
    RightBrace;
14 // 变量声明: VarDecl → BType VarDef { ',' VarDef } ';'
15 variableDeclaration: basicType variableDefinition (Comma variableDefinition)*
    Semicolon;
16 // 变量定义: VarDef → Ident { '[' IntConst ']' } [ '=' ConstInitVal ]
17 variableDefinition: Identifier (LeftBracket IntegerConstant RightBracket)* (Equal
    constantInitializationValue)?;
18 // 函数定义: FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
19 functionDefinition: functionType Identifier LeftParenthesis
    (functionFormalParameters)? RightParenthesis block;
```

```

20 // 函数类型FuncType → 'void' | 'int' | 'float' | 'char'
21 functionType: Void | Int | Float | Char;
22 // 形参列表FuncFParams → FuncFParam { ',' FuncFParam }
23 functionFormalParameters: functionFormalParameter (Comma
    functionFormalParameter)*;
24 // 函数形参FuncFParam → BType Ident [ '[' IntConst? ']' { '[' IntConst ']' } ]
25 functionFormalParameter: basicType Identifier (LeftBracket IntegerConstant?
    RightBracket (LeftBracket IntegerConstant RightBracket)*)?;

```

接着是语句和表达式部分:

```

1 // statement & expression
2 // 语句块: Block → '{' { BlockItem } '}'
3 block: LeftBrace (blockItem)* RightBrace;
4 // 语句块项: BlockItem → Decl | Stmt
5 blockItem: declaration | statement;
6 // 语句Stmt → LVal '=' Exp ';' | [ Exp ] ';' | Block | 'return' Exp? | 'if' '('
    Cond ')' Stmt [ 'else' Stmt ] | 'while' '(' Cond ')' Stmt | 'break' ';' |
    'continue' ';
7 statement: leftValue Equal expression Semicolon
8           | (expression)? Semicolon
9           | block
10          | Return expression? Semicolon
11          | If LeftParenthesis condition RightParenthesis statement (Else statement)?
12          | While LeftParenthesis condition RightParenthesis statement
13          | Break Semicolon
14          | Continue Semicolon;
15 // 表达式: Exp → AddExp
16 expression: addExpression;
17 // 常量算式: ConstExp → AddExp
18 constantExpression: addExpression;
19 // 条件算式Cond → LOrExp
20 condition: logicalOrExpression;
21 // 左值算式LVal → Ident { '[' Exp ']' }
22 leftValue: Identifier (LeftBracket expression RightBracket)*;
23 // 数值Number → IntConst | CharConst | FloatConst
24 number: IntegerConstant | CharacterConstant | FloatConstant;
25 // 函数实参表FuncRParams → Exp { ',' Exp }
26 functionRealParameters: expression (Comma expression)*;
27 // PrimaryExp → '(' Exp ')' | LVal | Number
28 primaryExpression: LeftParenthesis expression RightParenthesis
29                   | leftValue
30                   | number;

```

```

31 // UnaryExp → PrimaryExp | ('+' | '-' | '!') UnaryExp | Ident '(' [ FuncRParams ]
    ')' 注: '!'仅出现在条件表达式中
32 unaryExpression: primaryExpression
33                 | (Plus | Minus | ExclamationMark) unaryExpression
34                 | Identifier LeftParenthesis (functionRealParameters)?
                    RightParenthesis;
35 // MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
36 multiplicativeExpression: unaryExpression | multiplicativeExpression (Asterisk |
    Slash | Percent) unaryExpression;
37 // AddExp → MulExp | AddExp ('+' | '-') MulExp
38 addExpression: multiplicativeExpression | addExpression (Plus | Minus)
    multiplicativeExpression;
39 // RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
40 relationalExpression: addExpression | relationalExpression (Less | Greater |
    LessEqual | GreaterEqual) addExpression;
41 // EqExp → RelExp | EqExp ('==' | '!=') RelExp
42 equalityExpression: relationalExpression | equalityExpression (LogicalEqual |
    NotEqual) relationalExpression;
43 // LAndExp → EqExp | LAndExp '&&' EqExp
44 logicalAndExpression: equalityExpression | logicalAndExpression LogicalAnd
    equalityExpression;
45 // LOrExp → LAndExp | LOrExp '||' LAndExp
46 logicalOrExpression: logicalAndExpression | logicalOrExpression LogicalOr
    logicalAndExpression;

```

这两个部分也都是讲义中的内容,不再加以详细解释。

1.3.3 覆写 ANTLR 默认的文法错误处理机制

我们需要更改项目根目录下 `src` 文件夹中的内容。首先是 `main.cpp`,添加头文件如下:

```

1 #include <iostream> // 标准输入输出流
2 #include <filesystem> // 文件系统库 (C++17)
3 #include <fstream> // 文件流
4
5 #include "ANTLRInputStream.h" // ANTLR 输入流
6 #include "CactLexer.h" // 词法分析器
7 #include "CactParser.h" // 语法分析器
8 #include "include/syntax_error_listener.h" // 自定义语法错误监听器

```

这当中, `ANTLRInputStream.h`、`CactLexer.h` 和 `CactParser.h` 是我们在 `grammar` 目录下自动生成的文件, `syntax_error_listener.h` 是我们自定义的语法错误监听器。

接着继续修改 `main` 函数中的内容。

先检查是否提供了正确的命令行参数。编译器需要一个输入文件作为参数,如果参数数量不为 2(程序名称加一个文件路径),则显示使用说明并退出。

```
1 // 检查命令行参数
2 if (argc != 2) {
3     std::cerr << "Usage: " << argv[0] << " <input_file>" << std::endl << std::endl;
4     return 1;
5 }
```

然后尝试打开指定的源代码文件。如果文件无法打开,则输出错误信息并退出。同时,从完整路径中提取只包含文件名的部分,用于后续错误报告。

```
1 // 打开输入文件流
2 std::ifstream stream(argv[1]);
3 if (!stream) {
4     std::cerr << "Error: Could not open file: " << argv[1] << std::endl <<
        std::endl;
5     return 1;
6 }
7
8 // 获取文件名 (不包含路径)
9 std::string filename = std::filesystem::path(argv[1]).filename().string();
```

接下来初始化 ANTLR 组件。

```
1 // 从文件流读取字符
2 antlr4::ANTLRInputStream input(stream);
3
4 // 词法分析器,将字符流转换为标记流
5 CactLexer lexer(&input);
6
7 // 标记缓冲区
8 antlr4::CommonTokenStream tokens(&lexer);
9
10 // 语法分析器,分析标记流构建语法树
11 CactParser parser(&tokens);
```

移除默认的错误监听器,并添加自定义的错误监听器。

```
1 lexer.removeErrorListeners();
2 parser.removeErrorListeners();
3
4 cact_parser::CactErrorListener cact_error_listener;
5 lexer.addErrorListener(&cact_error_listener);
```



```
6 parser.addErrorListener(&cact_error_listener);
```

最后解析文件,进行三重检查,并捕获两种异常:

1. 三重检查

- (a) 确保语法树成功创建
- (b) 检查自定义错误监听器是否发现了语法错误
- (c) 确保所有输入都被消费(没有未匹配的输入)

2. 两种异常

- (a) ANTLR 特定的异常
- (b) 其他标准异常

```
1  try {
2      // 解析输入文件
3      antlr4::tree::ParseTree *tree = parser.compilationUnit();
4      if (!tree) {
5          std::cerr << "Error: Failed to parse input file." << std::endl;
6          return 1;
7      }
8      // 检查是否有语法错误
9      if (cact_error_listener.hasSyntaxError()) {
10         std::cerr << filename << ": Syntax error detected." << std::endl <<
11             std::endl;
12         std::cerr << "-----" << std::endl <<
13             std::endl; // 分隔线
14         return 1;
15     }
16     // 检查是否还有未消费的 token
17     if (tokens.LA(1) != antlr4::Token::EOF) {
18         std::cerr << filename << ": Error: Unmatched input detected after parsing."
19             << std::endl << std::endl;
20         std::cerr << "-----" << std::endl <<
21             std::endl; // 分隔线
22         return 1;
23     }
24     // 输出成功信息
25     std::cout << filename << ": Parsing succeeded." << std::endl << std::endl;
26     std::cerr << "-----" << std::endl <<
27         std::endl; // 分隔线
```

```
25 }catch (const antlr4::RecognitionException &e) {
26     // 处理 ANTLR 特定的异常
27     std::cerr << "ANTLR Recognition error: " << e.what() << std::endl;
28     std::cerr << filename << ": Parsing failed." << std::endl << std::endl;
29     return 1;
30 }
31 catch (const std::exception &e) {
32     // 捕获异常并输出错误信息
33     std::cerr << "Error while parsing file '" << argv[1] << "': " << e.what() <<
        std::endl << std::endl;
34     return 1;
35 }
```

接下来讲讲检查语法错误的部分。我们在 `syntax_error_listener.h` 中定义了一个自定义的语法错误监听器类 `CactErrorListener`，它继承自 ANTLR 的 `BaseErrorListener` 类。这个类重写了 `syntaxError` 方法，以便在发生语法错误时输出详细的错误信息。

当语法出现错误时，`syntaxError` 方法会被调用，我们在这个方法中输出错误信息。我们还设置了一个标志 `has_syntax_error`，用于指示是否发生了语法错误。

```
1  #ifndef CPLAB_SYNTAX_ERROR_LISTENER
2  #define CPLAB_SYNTAX_ERROR_LISTENER
3
4  #include "BaseErrorListener.h"
5
6  namespace cact_parser
7  {
8      class CactErrorListener final : public antlr4::BaseErrorListener
9      {
10     public:
11         // 语法错误监听器的构造函数
12         void syntaxError(antlr4::Recognizer *recognizer, antlr4::Token
            *offendingSymbol,
13             size_t line, size_t charPositionInLine, const std::string &msg,
14             std::exception_ptr e) override;
15
16         // 检查是否有语法错误
17         bool hasSyntaxError();
18
19     private:
20         bool has_syntax_error = false; // 标志是否有语法错误
21     };
22 } // namespace cact_parser
23 #endif // CPLAB_SYNTAX_ERROR_LISTENER
```

在 `syntax_error_listener.cpp` 中实现了这个类的成员函数, 输出错误信息, 包括错误消息、行号、字符位置和错误的符号。我们还提供了一个方法 `hasSyntaxError`, 用于返回是否发生了语法错误。

```
1 namespace cact_parser {
2     // 语法错误监听器的构造函数
3     void CactErrorListener::syntaxError(antlr4::Recognizer *recognizer,
4         antlr4::Token *offendingSymbol,
5         size_t line, size_t charPositionInLine, const std::string &msg,
6         std::exception_ptr e) {
7         has_syntax_error = true; // 设置语法错误标志为true
8         std::cerr << "Syntax Error Message: " << msg << std::endl; // 输出语法错误信息
9         std::cerr << "Line: " << line << ", Position: " << charPositionInLine <<
10            std::endl; // 输出错误位置
11         std::cerr << "Offending Symbol: " << (offendingSymbol ?
12            offendingSymbol->getText() : "null") << std::endl; // 输出错误的符号
13     }
14     bool CactErrorListener::hasSyntaxError() {
15         return has_syntax_error; // 返回语法错误标志
16     }
17 }
```

1.3.4 测试结果

编译得到编译器之后, 对于给出的 27 个测试用例, 我们编写了一个脚本来自动化测试。脚本会遍历 `test` 目录下的所有文件, 并将每个文件的内容传递给编译器进行解析, 并输出结果。

```
1 # 遍历 ./test 文件夹中的每个文件
2 for file in ./test/samples_lex_and_syntax/*; do
3     # 执行 ./build/compiler 命令, 传递文件名作为第二个参数
4     ./build/compiler "$file"
5 done
```

测试结果如下:

```
00_true_main.cact: Parsing succeeded.

-----

Syntax Error Message: extraneous input 'x' expecting {'', ' ',';'}
Line: 3, Position: 13
Offending Symbol: x
01_false_hex_num.cact: Syntax error detected.

-----

02_true_octo.cact: Parsing succeeded.

-----

Syntax Error Message: mismatched input '}' expecting ']'
Line: 2, Position: 11
Offending Symbol: }
03_false_bracket.cact: Syntax error detected.

-----

04_true_multi_dim_array.cact: Parsing succeeded.

-----

Syntax Error Message: token recognition error at: '~'
Line: 4, Position: 15
Offending Symbol: null
05_false_number.cact: Syntax error detected.
```

```
Syntax Error Message: extraneous input 'G' expecting {'', ' ',';'}
Line: 3, Position: 13
Offending Symbol: G
06_false_hex_num.cact: Syntax error detected.

-----

07_false_global_exp.cact: Error: Unmatched input detected after parsing.

-----

Syntax Error Message: token recognition error at: '"'
Line: 3, Position: 9
Offending Symbol: null
Syntax Error Message: token recognition error at: '"'
Line: 3, Position: 12
Offending Symbol: null
08_false_int_num_decl.cact: Syntax error detected.

-----

Syntax Error Message: extraneous input '7' expecting Identifier
Line: 3, Position: 5
Offending Symbol: 7
Syntax Error Message: extraneous input 'j' expecting ';'
Line: 4, Position: 9
Offending Symbol: j
09_false_val_name.cact: Syntax error detected.
```

```
Syntax Error Message: no viable alternative at input 'a[2,'
Line: 4, Position: 4
Offending Symbol: ,
Syntax Error Message: mismatched input ',' expecting ';'
Line: 4, Position: 4
Offending Symbol: ,
Syntax Error Message: mismatched input ']' expecting ';'
Line: 4, Position: 7
Offending Symbol: ]
Syntax Error Message: token recognition error at: '.0;'
Line: 4, Position: 12
Offending Symbol: null
Syntax Error Message: missing ';' at 'return'
Line: 5, Position: 1
Offending Symbol: return
10_false_array_visit.cact: Syntax error detected.
```

```
-----
Syntax Error Message: missing ';' at '}'
Line: 7, Position: 1
Offending Symbol: }
11_false_if_else.cact: Syntax error detected.
```

```
-----
12_true_comment.cact: Parsing succeeded.
```

```
Syntax Error Message: extraneous input '*' expecting {'const', 'int', 'float', 'char', 'if', 'while', 'break', 'continue', 'return', '(', '!', '-', '+', '{', '}', ';', Identifier, IntegerConstant, CharacterConstant, FloatConstant}
Line: 9, Position: 1
Offending Symbol: *
13_false_nested_comment.cact: Syntax error detected.
```

```
-----
14_true_sample.cact: Parsing succeeded.
```

```
-----
15_true_syntax_false_semantic.cact: Parsing succeeded.
```

```
-----
Syntax Error Message: extraneous input 'else' expecting {'const', 'int', 'float', 'char', 'if', 'while', 'break', 'continue', 'return', '(', '!', '-', '+', '{', '}', ';', Identifier, IntegerConstant, CharacterConstant, FloatConstant}
Line: 8, Position: 1
Offending Symbol: else
16_false_if_else.cact: Syntax error detected.
```

```
-----
17_true_multi_dim_fparam.cact: Parsing succeeded.
```

```
Syntax Error Message: mismatched input '=' expecting ';'
Line: 5, Position: 7
Offending Symbol: =
18_false_continuous_equation.cact: Syntax error detected.

-----

19_false_val_init.cact: Parsing succeeded.

-----

20_false_val_init_op.cact: Parsing succeeded.

-----

Syntax Error Message: no viable alternative at input 'retunra'
Line: 13, Position: 8
Offending Symbol: a
21_false_token.cact: Syntax error detected.

-----

22_true_func.cact: Parsing succeeded.

-----

23_false_val_init_func.cact: Parsing succeeded.

-----

Syntax Error Message: mismatched input 'foo' expecting IntegerConstant
Line: 11, Position: 7
Offending Symbol: foo
Syntax Error Message: extraneous input ']' expecting ';'
Line: 11, Position: 13
Offending Symbol: ]
24_false_array_size_func.cact: Syntax error detected.

-----

Syntax Error Message: mismatched input '(' expecting {'(', ',', ';' }
Line: 2, Position: 13
Offending Symbol: (
Syntax Error Message: mismatched input ')' expecting {'(', '!', '-', '+'
, Identifier, IntegerConstant, CharacterConstant, FloatConstant}
Line: 2, Position: 14
Offending Symbol: )
Syntax Error Message: missing ';' at '{'
Line: 2, Position: 15
Offending Symbol: {
25_false_nested_func_def.cact: Syntax error detected.

-----

26_true_multi_dim_const.cact: Parsing succeeded.
```

图 1.2. 测试结果

需要指出一点, 样例 19、20、23 均应输出通过, 这里样例的命名有误。由此可见, 我们的编译器在词法和语法分析方面表现良好, 能够正确处理各种样例。

1.4 思考题

1. 如何设计编译器的目录结构?

编译器的目录结构通常包括以下几个部分:

- (a) `src`: 源代码目录, 存放编译器的源代码文件。
- (b) `grammar`: 文法文件目录, 存放 ANTLR 生成的文法文件。
- (c) `test`: 测试用例目录, 存放测试用例和测试脚本。
- (d) `deps`: 依赖库目录, 存放编译器所需的外部库和工具。
- (e) `build`: 构建目录, 存放编译生成的可执行文件和中间文件。

2. 如何把表达式优先级体现在文法设计中?

在文法设计中, 可以通过调整规则的定义顺序来体现表达式的优先级。举例如下:

```
1      // 乘法优先级高于加法
2      S -> E + F | E
3      E -> E * F | F
4      F -> id
```

在生成的解析树中, 乘法操作会在加法操作之前进行解析, 从而实现优先级。

3. 如何设计数值常量的词法规则?

数值常量分成整数常量、浮点常量、字符常量三种类型。在我们的代码中, 这三种变量定义如下:

```
1      IntegerConstant: ('0' | [1-9][0-9]* | '0'[0-7]+ | '0'[xX][0-9a-fA-F]+);
2
3      CharacterConstant: '\\' ( EscapeSequence | ~['\\'] ) '\\';
4      fragment EscapeSequence: '\\' ['"?\\abfnrtv0];
5
6      FloatConstant: ([-]? '.'[0-9]+ |
7                      [-]?[0-9]+'.'[0-9]* |
8                      [-]? '.'[0-9]+ [eE][+-]?[0-9]+ |
9                      [-]?[0-9]+('.'[0-9]*)?[eE][+-]?[0-9]+ ) [fF];
```

其中整数常量支持十进制、八进制和十六进制表示, 字符常量用单引号括起来, 浮点常量支持多种格式。

4. 如何替换 ANTLR 的默认异常处理方法?

我们可以通过覆写 `BaseErrorListener` 类中的 `syntaxError` 方法来实现自定义的异常处理。在这个方法中, 我们可以输出更详细的错误信息, 并设置一个标志来指示是否发生了语法错误。

1.5 实验总结

本次实验主要是对编译器的词法分析和语法分析部分进行实现。通过使用 ANTLR 工具,我们能够快速地生成词法分析器和语法分析器,并且能够自定义错误处理机制。这是我们第一次接触编译器的实现,虽然过程有些繁琐,但通过实验,我们对编译器的工作原理有了更深入的理解。这使得我们在后续的实验中能够更好地进行语义分析和代码生成等工作。