实验报告

一、实验目的

1. 熟悉 C 语言语法规则，了解编译器语法分析器的主要功能；
2. 熟练掌握典型语法分析器构造的相关技术和方法，设计并实现具有一定复杂度和分析能力的 C 语言语法分析器；
3. 了解 ANTLR 的工作原理和基本思想，学习使用工具自动生成语法分析器；
4. 掌握编译器从前端到后端各个模块的工作原理，语法分析模块与其他模块之间的交互过程。

二、实验内容

本次实验实现了 C 语言的一个子集。基于 BIT-MINICC 构建了 C 语言子集的语法分析器，包括基本的函数声明、变量声明、数组声明使用、基本表达式、分支语句、循环语句进行了语法分析的实现。此语法分析器使用递归下降分析的方式，主要以 LL(1)文法进行设计与实现。语法分析器以词法分析生成的属性字节流作为输入，进行语法分析和错误处理。若输入符合语法规范，则最终生成 json 格式的语法树；输入不正确则报告相应的语法错误。

语法分析器支持的语法项目包括：函数声明、变量定义与初始化、一维数组变量定义初始化、循环语句、分支语句、跳转语句、赋值表达式、逻辑运算与算术运算表达式、前缀表达式、后缀表达式等。

三、实验过程与步骤

1、语法规则设计

参照 C 语言标准规范，对实验 4 所设计的文法进行简化与修改，得到本次使用的文法。同时考虑到实现的难易程度，此文法主要为 LL(1)文法，但对于个别处理较为复杂之处，则考虑实际实现进行优化。

由于在实际实现过程中遇到了一系列的问题，因此此处设计的语法与代码实现的文法略有出入。但是从功能上看，实现代码能够完成对语法规则所描述的文法的分析，并生成正确的语法树。

```
TYPE -> 'int' | 'float' | 'char' | 'void'
PLUS_SUB -> '+' | '-'
MUT_DIV -> '*' | '/' | '%'

ASSIGN_OP -> '=' | '*=' | '/=' | '%=' | '+=' | '-=' | '<<=' | '>>=' | '&=' |
'^=' | '|='
UNARY_OP -> '++' | '--'
```

```
POST_OP -> '++' | '--'
SHIFT_OP -> '<<' | '>>'
RELATIONAL_OP -> '<' | '>' | '<=' | '>='
EQUALITY_OP -> '==' | '!='
AND_OP -> '&'
EXC_OR_OP -> '^'
INC_OR_OP -> '|'
LOG_AND_OP -> '&&'
LOG_OR_OP -> '||'

IDENTIFIER -> 'identifier'

compliation_unit -> erternal_declaration
external_declaration -> function external_declaration | e
function -> TYPE function_declarator code_block

function_declarator -> variable_declarator '(' arguments ')'
variable_declarator -> IDENTIFIER
declarator -> IDENTIFIER '[' expression ']' | IDENTIFIER

code_block -> '{' compound_statement '}'
_codeblock -> code_block | statement

compound_statement -> line
line -> declaration | statement

declaration -> TYPE init_lists ';'
init_lists -> init_list ',' init_lists | e
init_list -> declarator | declarator '=' expression | declarator '=' '{'
expression_statement '}'

arguments -> arg_list | e
arglist -> argument ',' arglist | argument | e
argument -> TYPE declarator
declaration -> TYPE init_lists ';'

statement -> jump_statement ';' | selection_statement | iteration_statement |
expression_statement ';'
iteration_statement -> for '(' declaration ; expression  ; expression  ')'
_code_block| for '(' expression ; expression ; expression ')' _code_block
selection _statement -> 'if' '(' expression ')' _code_block | 'if' '('
expression ')' _code_block 'else' _code_block
```

```
jump_statement -> 'return' expression | 'goto' expression | 'break' |
'continue'
expression_statement -> expression ',' expression_statement | expression

expression -> assignment_expression
assignment_expression -> logical_OR_expression | logical_OR_expression
ASSIGN_OP assignment_expression
logical_OR_expression -> logical_AND_expression | logical_AND_expression
LOG_OR_OP logical_OR_expression
logical_AND_expression -> inclusive_OR_expression | inclusive_OR_expression
LOG_AND_OP logical_AND_expression
inclusive_OR_expression -> exclusive_OR_expression | exclusive_OR_expression
INC_OR_OP inclusive_OR_expression
exclusive_OR_expression -> AND_expression | AND_expression EXC_OR_OP
exclusive_OR_expression
AND_expression -> equality_expression | equality_expression AND_OP
AND_expression
equality_expression -> relational_expression | relational_expression
EQUALITY_OP equality_expression
relational_expression -> shift_expression | shift_expression RELATIONAL_OP
relational_expression
shift_expression -> additive_expression | additive_expression SHIFT_OP
shift_expression
additive_expression -> multiplicative_expression | multiplicative_expression
PLUS_SUB additive_expression
multiplicative_expression -> unary_expression | unary_expression MUT_DIV
multiplicative_expression
unary_expression -> UNARY_OP unary_expression | postfix_expression
postfix_expression -> primary_expression | pimary_expression POST_OP |
primary_expression '[' expression_statement ']'
primary_expression -> IntegerConstant | FloatingConstant | CharacterConstant |
StringLiteral | '(' expression ')' | IDENTIFIER '(' expression_statement ')' |
IDENTIFIER
```

2、代码实现

a、实现思路与工具函数设计

代码实现部分主要参考了 BIT-MiniCC 编译器中 ExampleParser 的代码风格和实现方式，借助 ast 文件包中语法树节点类进行语法树的构建。在程序执行时，先通过 load_Tokens()函数将词法分析所提供的 Token 流解析为 ScannerToken 的属性字节流结构体列表。解析程序运行时，tokenIndex 控制对属性字节流访问的控制，nextToken 存储当前读取到的属性字节流。

这里设计了两种函数用来获取下一个 Token，next()、match()。next 方法不校验当前获取的属性字节流，直接获取下一个；而 match()方法需要匹配输入的 type 类型与当前读取到的属性字节流 nextToken 的类型。若类型不符，则会提示报错，并继续进行识别。

针对语法分析程序的报错功能，在每个函数的状态转中均会对获取希望语法单词以外的单词进行报错提示。其中，一部分是由相应的函数体进行进行输出，另一部分则是由 match()函数所产生。

b、递归下降分析

在进行代码实现时，考虑到代码的逻辑可理解性，并非所有的状态-即函数均在开始时便使用 FIRST(S)集合对状态进行转移，部分状态下会先转去识别一个串，再返回使用 FIRST(S)集合进行状态转。这两种情况的典型是对陈述（如 statement）和表达式（如 assignment_expression）的实现。下面的代码给出两种的对比：

```
//陈述
private ASTStatement statement() {
    //跳转语句
    if (JUMP_WORD.contains(nextToken.type)) {
        ASTStatement res = jump_statement();
        res.children.add(new ASTToken(nextToken.lexme, tokenIndex));
        match("';'");
        return res;
    }
    //选择语句陈述
    else if (SELECT_WORD.contains(nextToken.type)) {
        return selection_statement();
    }
    //循环语句
    else if(ITERATION_WORD.contains(nextToken.type)){
        return iteration_statement();
    }
    //其它 一般语句
    else {    //表达式
        ASTStatement res = expression_statement();
        res.children.add(new ASTToken(nextToken.lexme, tokenIndex));
        match("';'");
        return res;
    }
}
```

```
//赋值表达式
private ASTExpression assignment_expression() {
    //尝试匹配前缀表达式
    int curtoken = tokenIndex;
    ASTExpression expr = logical_OR_expression();
    //匹配 赋值表达式 成功
    if(ASSIGN_OP.contains(nextToken.type)) {
        ASTBinaryExpression res = new ASTBinaryExpression();
        //前表达式
        res.expr1 = expr;
        //匹配赋值符
        res.op = new ASTToken(nextToken.lexme,tokenIndex);
        next();
        //后表达式
        res.expr2 = assignment_expression();
        //添加孩子
    res.children.add(res.expr1);
            res.children.add(res.op);
        res.children.add(res.expr2);
    return res;
    }
    //匹配 赋值表达式 失败    回溯   匹配二元表达式
    else{
        return expr;
    }
}
```

对像 assignment_expression 这类函数进行进一步拆分，可以得到严格符合 LL(1)文法的分析器。但这样会使得程序中出现过多的状态转移，使得函数的调用结构和状态转移更加复杂，同时也会使得分析栈较深。这样一方面会使得需要编写的程序更加繁琐，同时也可能使分析的效率较为低下。因此最终采用此种较为简单，但更加符合人对程序认知的方式进行编码的实现。

同时针对 assignment_expression 赋值语句的实现，在最初的实现时考虑到 C 语言中赋值运算符左侧的表达式必须为“可修改的左值”（词短语来源于 Visual Studio 对赋值语句的报错），即要求左侧必须包含变量并满足一定的约束条件。因此使用带有约束条件的 pistfix_expression 前缀表达式对左侧表达式进行匹配。而这样对于一般的前缀表达式可能无法匹配，因此需要根据匹配后剩余的字符进行回溯与重新匹配。经过慎重考虑，我认为“可修改的左值”这一条件应是在语义分析阶段产生的报错，因此左侧表达式直接匹配下一级表达式 logical_OR_expression 即可。

四、运行效果

使用 BIT-MiniCC 框架对语法分析器进行运行测试。测试代码为框架中所提供的四条测试代码，并与框架自带的语法分析器所生成的语法分析结果进行对比以验证语法分析程序的正确性。下面以框架中两条典型的测试用例进行简要分析，介绍我所实现的语法分析功能。对于语法分析器支持的数组的声明使用、前缀表达式的解析均为体现。

1、0_example_test

```
int sum(int a, int b) {
    return a + b;
}
```

图 1 0_example_test 代码图

上图展示了测试代码。这个测试代码仅包含了简单的函数定义与返回。函数定义方面支持 a、b 两个整型参数作为输入；返回方面返回了一个表达式。

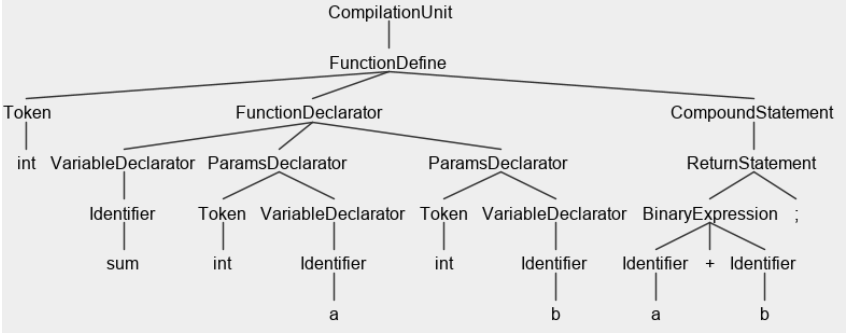

图 2 0_example_test 语法树图

上图是语法分析后生成的语法树图片。从图中可以看到，语法分析程序对这段程序的解析式正确的。

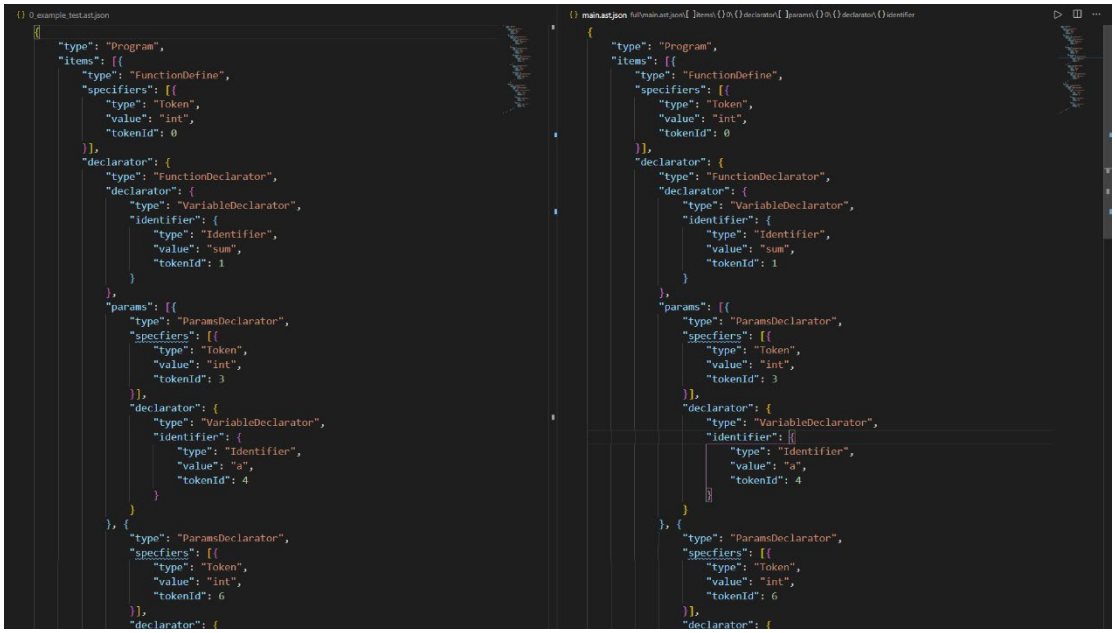

图 3 0_example_test 生成 json 文件对比图

上图左侧是由 BIT-MiniCC 生成的 json 文件；右侧是语法分析程序生成的文件。显然，两个文件的内容完全一致，说明语法分析程序能够正确解析目标程序，并生成符合 BIT-MiniCC 框架格式要求的 json 格式语法树文件。

2、2_parser_test1

```
int fibonacci ( int n ) {
    if ( n < 2 )
        return n;
    int f = 0 , g = 1 ;
    int result = 0 ;
    for ( int i = 0 ; i < n ; i ++ ) {
        result = f + g ;
        f = g ;
        g = result ;
    }
    return result ;
}
int main ( ) {
  int a = 10;
  int res = fibonacci ( a ) ;
  _OUTPUT ( res ) ;
  return 0 ;
}
```

图 4 2_parser_test1 代码图

上图展示了测试代码。此测试代码功能为使用函数调用的方式计算斐波那契数列的第 n 项。代码中包含了函数定义、函数调用、循环语句、条件判断语句等多种语句类型。

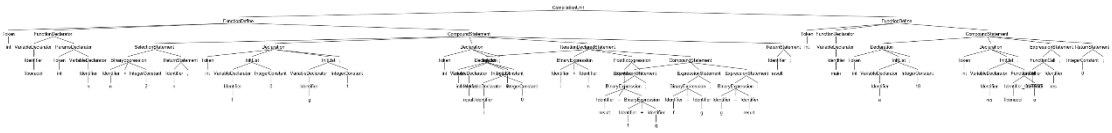

图 5 2_parser_test1 语法树图

上图是语法分析后生成的语法树图片。与实验报告中提供的语法树结构相对比，这棵树的结构与其相似，但在叶子节点上，其深度更小。主要是由于在实现中，对表达式递归解析时以最深层匹配到的表达式类型作为当前节点类型。从正确性上分析，这颗语法树是正确的。

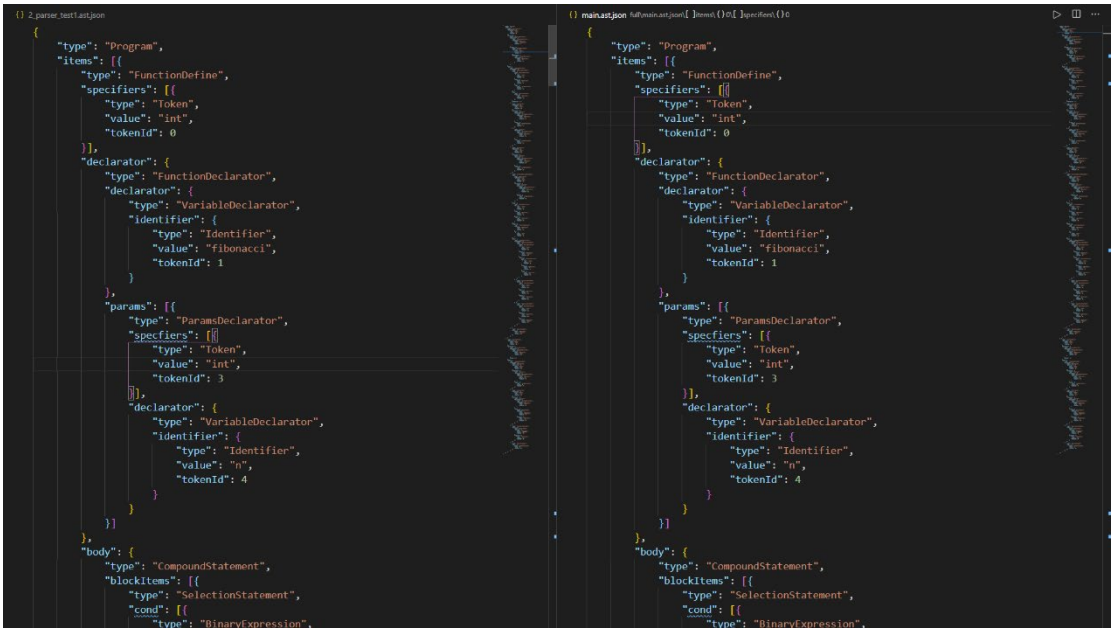

图 6 2_parser_test1 生成 json 文件对比图

上图左侧是由 BIT-MiniCC 生成的 json 文件；右侧是语法分析程序生成的文件。经过对比，两个文件完全相同。这说明语法分析程序能够正确解析函数调用、循环语句、分支语句等多种语法结构。

3、3_parser_test2

```
int main() {
  int a[5], i, j, t;
  for(i = 0; i < 5; i++) {
    a[i] = MARS_GETI();
  }
  for(i = 0; i < 4; i++) {
    for(j = 0; j < 5 - i; j++) {
      if(a[j] > a[j+1]) {
        t = a[j];
        a[j] = a[j+1];
        a[j+1] = t;
      }
    }
  }
  for(i = 0; i < 5; i++) {
    t = a[i];
    MARS_PUTI(t);
    MARS_PUTS("\n");
  }
  return;
}
```

图 7 3_parser_test2 代码图

上图展示了测试代码。此代码主要测试了对循环语句的解析、分支语句、函数调用语句和数组声明使用的解析，返回值为空表达式。同时这个程序对循环语句和分支语句进行了嵌套，结构相较更为复杂。

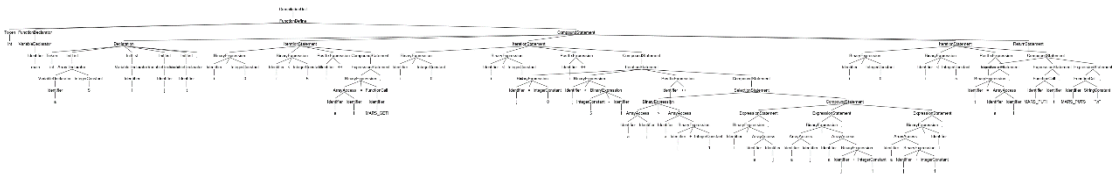

图 8 3_parser_test2 语法树图

上图是语法分析后生成的语法树图片。

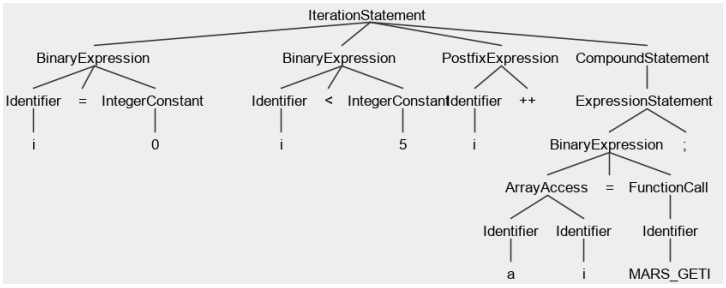

图 9 3_parser_test2 数组部分语法树图

上图是含有数组的表达式的语法树图片。可以观察到数组变量被正确的解析和使用。

图 10 3_parser_test2 生成 json 文件对比图

上图左侧是由 BIT-MiniCC 生成的 json 文件；右侧是语法分析程序生成的文件。通过对语法树的分析，程序能够正确的解析数组、循环语句、分支语句及它们的嵌套结构。

4、4_parser_test3

```
int main() {
  int a;

  a = MARS_GETI();
  a *= 10;
  MARS_PUTI(a);
  a ++;
  MARS_PUTI(a);
  return 0;
}
```

图 11 4_parser_test3 代码图

上图展示了测试代码。此代码主要测试了函数调用、赋值语句和后缀表达式的解析。

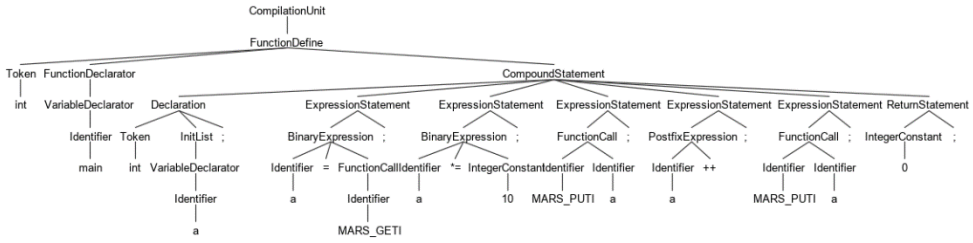

图 12 4_parser_test3 语法树图

上图是语法分析后生成的语法树图片。

图 13 4_parser_test3 生成 json 文件对比图

上图左侧是由 BIT-MiniCC 生成的 json 文件；右侧是语法分析程序生成的文件。

五、心得体会

本次实验使用递归下降分析的思路，设计了一个对 C 语言子集的语法分析程序。程序能够正确的解析目标程序，并生成相应的 json 格式语法树，以供后续编译阶段的程序使用。在设计实现的学习过程中，一方面我对递归程序的理解程度得到了加深，另一方面我对语法分析的算法思路有了更加清晰的认识。对于 LL(1)文法中 FISRT(S)集合、FOLLOW(S)集合在词法分析过程中所起到的作用有了更加理性的认识。

从效果上来看，这个语法分析器能够在 BIT-MiniCC 框架下正确运行，并能够解析目标程序构建正确的语法树。但是由于在实现过程中出现了一些问题因此对函数调用结构进行了一些调整，从而使得部分程序的调用结构不便于理解。同时，多维数组的定义初始化与使用、指针变量的定义使用等语法规则此语法分析器也没有进行实现。进一步改进方面，可以通过优化语法规则的设计，为程序增加更多的功能，也使程序的结构更加合理便于理解。