

语法分析

ANTLR 介绍

ANTLR 不仅是一个功能强大的语法分析器自动生成工具，也可以用于结构化文本或者二进制文件的读入、处理、执行和翻译转换。目前在工业界广泛用于各种语言的实现和文本处理。Twitter Search 使用 ANTLR 解析每天 20 亿条查询关键字，Hive 和 Pig 语言的实现，Jython 和 GraalVM 虚拟机的实现中都使用 ANTLR 生成的语法分析器，除此之外，还可以用来实现 JSON 和 XML 的解析等。对于给定的形式化语言的文法描述，ANTLR 自动生成的语法分析器可以用于该语言的语法分析树的自动生成，除此之外，还生成了用于语法树遍历的 visitor，极大方便了语言前端分析功能的实现**错误!未找到引用源。**

ANTLR 使用了一种称为自适应 LL(*) 或者 ALL(*) 的语法分析算法，ALL(*) 是在 v3 LL(*) 的基础之上的一个扩展，LL(*) 在生成的语法分析器运行之前完成文法解析，而 ALL(*) 在编译时动态进行文法解析。ALL(*) 与 LL(*) 相比的优势在于能够根据输入序列选择使用文法，而采用静态分析的 LL(*) 则需要考虑所有可能的输入。

ANTLR 根据输入的文法生成输出一个自顶向下的递归下降分析器，所生成的递归下降分析器的每个方法对应文法中每个规则，并从语法树树根开始语法树的构建。文法开始符号对应的规则完成语法树根节点的构造。

ANTLR 主要由两部分构成，一个是 ANTLR 工具自身，另一个是 ANTLR 运行时（分析时）API。前者对应 `class org.antlr.v4.Tool`，用于处理文法，生成包括 lexer 和 parser 所需的代码，而 ANTLR 运行时包括了生成代码运行所需的辅助类和方法。使用时首先使用 ANTLR 工具针对某个文法生成代码，然后将这些代码与运行时 jar 文件一起编译生成应用所需的代码。应用运行时以来 ANTLR 运行时库。

以数组或者结构体初始化的大括号嵌套语句为例，例如 {1, 2, 3} 或者 {1, {2, 3}, 4}，对应的 ANTLR 输入文件 `ArrayInit.g4` 文件内容如下所示：

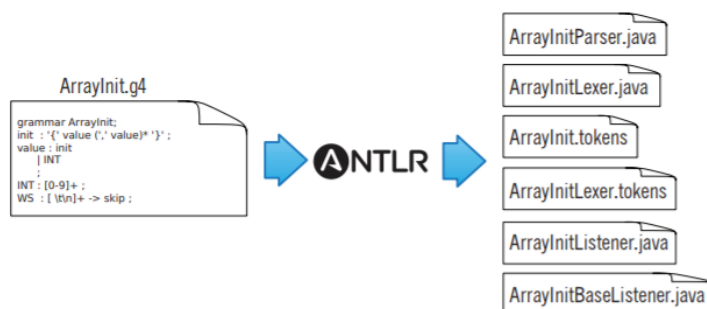
```
grammar ArrayInit;  
init : '{' value (',' value)* '}'
```

```

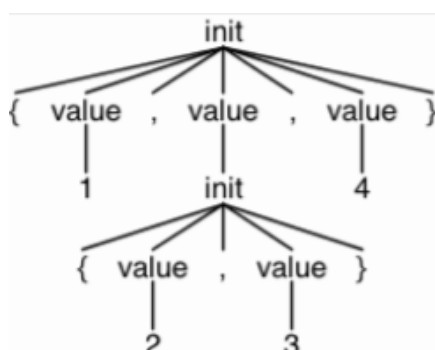
value: init
      | INT
      ;
INT:   [0-9]+;
WS:    [ \t\r\n]+ -> skip;

```

上述文法经过 ANTLR 处理生成如下所示的 6 个 java 文件，



这些代码与 ANTLR 运行时库一起编译，然后处理输入 {1, {2, 3}, 4} 后形成如下的语法树。



语法树定义

为保证自定义实现的语法分析器能够与 BIT-MiniCC 框架后续处理阶段正常衔接，要求语法分析器输出规定格式的抽象语法树中间表示，输出采用 JSON 格式表示。

每个 JSON 结点使用名为 “type” 的属性区分 AST 节点类别，不同种类的 AST 节点内部结构不同，需保证每种类型的结点属性与规范完全一致（属性名，值的类别，属性顺序）。下面依次对 BIT-MiniCC 定义的 AST 结点进行说明。

（1）表达式结点

1. CastExpression: 转换类型表达式，譬如 (int)a
2. ArrayAccess: 数组访问类型表达式，譬如 a[1]





3. ConditionExpression: 条件表达式, 譬如 `flag?x++:x--`
4. FunctionCall: 函数调用, 譬如 `fun(1,2)`
5. PostfixExpression: 后缀表达式, 譬如 `x++`
6. UnaryExpression: 一元(前缀)表达式, 譬如 `&x` , `++x`, `sizeof(x)`
7. UnaryTypename: 特殊的一元表达式, 譬如 `sizeof(int)`
8. BinaryExpression: 二元表达式, 譬如 `x*2` , `x<<2` 等

(2) 声明结点

1. 普通变量类型的声明
2. 数组类型变量的声明
3. 函数声明

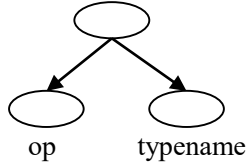
(3) 语句结点

1. LabeledStatement: 只包括 Identifier ':' statement 类型。
2. CompoundStatement
3. ExpressionStatement
4. SelectionStatement: 包括 if, if else, if else if, 不包括 switch。
5. IterationStatement: 包括 for 语句, 不包括 while 和 do while。
6. JumpStatement: 包括 goto , break, continue, return。

结点类型	结点示例
IntegerConstant IntegerConstant 	<pre>// 1 { "type": "IntegerConstant", "value": 1, "tokenId": 7 // 词法分析器输出 token 的 id }</pre>
CharConstant CharConstant 	<pre>// 'k' { "type": "CharConstant", "value": "k", "tokenId": 5 }</pre>
FloatConstant FloatConstant 	<pre>{ "type": "FloatConstant", "value": 1.234, "tokenId": 5 }</pre>
StringLiteral StringLiteral 	<pre>// "Hello, world!" { "type": "StringLiteral", "value": "\"Hello, World!\"", "tokenId": 5 }</pre>

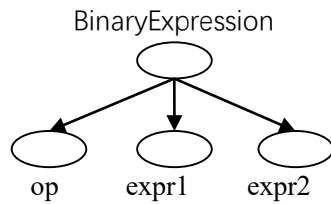
<p>Identifier</p> <p>Identifier</p> 	<pre>// x { "type": "Identifier", "value": "a", "tokenId": 6 }</pre>
<p>UnaryExpression</p> <p>UnaryExpression</p> 	<pre>// ++a { "type": "UnaryExpression", "op": { "value": "++", "tokenId": 5 }, "expr": { "type": "Identifier", "value": "a", "tokenId": 6 } }</pre>
<p>UnaryTypename</p>	<pre>// sizeof(int) { "type": "UnaryTypename", "op": { "value": "sizeof", "tokenId": 5 }, </pre>

UnaryType



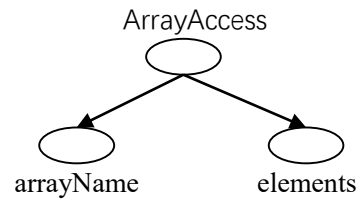
```
"typename": {  
  "type": "Type",  
  "specifiers": [  
    {  
      "value": "int",  
      "tokenId": 7  
    }  
  ],  
  "declarator": null  
}
```

BinaryExpression

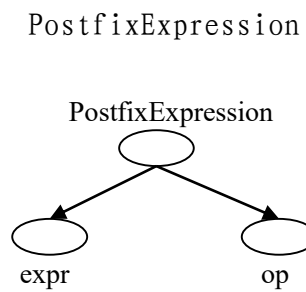


```
// 1*2  
{  
  "type": "BinaryExpression",  
  "op": {  
    "value": "*",  
    "tokenId": 9  
  },  
  "expr1": {  
    "type": "IntegerConstant",  
    "value": 1,  
    "tokenId": 8  
  },  
  "expr2": {  
    "type": "IntegerConstant",  
    "value": 2,  
  }  
}
```

	<pre> "tokenId": 10 } } </pre>
<p>FunctionCall</p>  <pre> graph TD FC([FunctionCall]) --> FNC([funcname]) FC --> AL([argList]) </pre>	<pre> // fun(a,1) { "type": "FunctionCall", "funcname": { "type": "Identifier", "value": "fun", "tokenId": 12 }, "argList": [{ "type": "Identifier", "value": "a", "tokenId": 14 }, { "type": "IntegerConstant", "value": 1, "tokenId": 16 }] } </pre>
ArrayAccess	<pre> // a[1] { </pre>

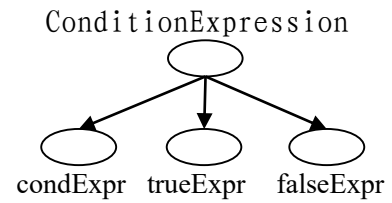


```
"type": "ArrayAccess",  
"arrayName": {  
  "type": "Identifier",  
  "value": "a",  
  "tokenId": 5  
},  
"elements": [  
  {  
    "type": "IntegerConstant",  
    "value": 1,  
    "tokenId": 7  
  }  
]  
}
```



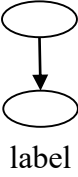


```
// a++  
{  
  "type": "PostfixExpression",  
  "expr": {  
    "type": "Identifier",  
    "value": "a",  
    "tokenId": 25  
  },  
  "op": {  
    "value": "++",  
    "tokenId": 26  
  }  
}
```


	<pre> }</pre>
<p>CastExpression</p>  <pre> graph TD CastExpression --> typename CastExpression --> expr</pre>	<pre> // (int)a { "type": "CastExpression", "typename": { "specifiers": [{ "value": "int", "tokenId": 29 }], "declarator": null }, "expr": { "type": "Identifier", "value": "a", "tokenId": 31 } } }</pre>
<p>ConditionExpression</p>	<pre> // flag ? x-- : x++; { "type": "ConditionExpression", "condExpr": { "type": "Identifier", "value": "flag", "tokenId": 33 } }</pre>



```
},  
"trueExpr": [  
  {  
    "type": "PostfixExpression",  
    "expr": {  
      "type": "Identifier",  
      "value": "x",  
      "tokenId": 35  
    },  
    "op": {  
      "value": "--",  
      "tokenId": 36  
    }  
  }  
],  
"falseExpr": {  
  "type": "PostfixExpression",  
  "expr": {  
    "type": "Identifier",  
    "value": "x",  
    "tokenId": 38  
  },  
  "op": {  
    "value": "++",  
    "tokenId": 39  
  }  
}
```

	<pre> } } </pre>
<p>BreakStatement</p> <p>BreakStatement</p> 	<pre> // break; { "type": "BreakStatement", "tokenId": 10 } </pre>
<p>ContinueStatement</p> <p>ContinueStatement</p> 	<pre> // continue; { "type": "ContinueStatement", "tokenId": 12 } </pre>
<p>GotoStatement</p> <p>GotoStatement</p>  <p>label</p>	<pre> // goto k; { "type": "GotoStatement", "flag": { "type": "Identifier", "value": "k", "tokenId": 15 }, "tokenId": 14 } </pre>
<p>ReturnStatement</p>	<pre> // return 1; { "type": "ReturnStatement", "expr": [</pre>

<p>ReturnStatement</p>  <p>expr</p>	<pre> { "type": "IntegerConstant", "value": 1, "tokenId": 18 }], "tokenId": 17 } </pre>
<p>LabeledStatement</p>  <p>label stat</p>	<pre> // k : break; { "type": "LabeledStatement", "label": { "type": "Identifier", "value": "k", "tokenId": 20 }, "stat": { "type": "BreakStatement", "tokenId": 22 } } } </pre>
<p>CompoundStatement</p>	<pre> // { goto k; break; } { "type": "CompoundStatement", "blockItems": [{ </pre>

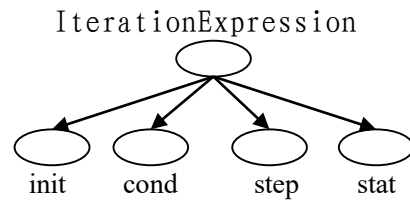
<p>CompoundStatement</p>  <p>blockItems</p>	<pre>"type": "GotoStatement", "flag": { "type": "Identifier", "value": "k", "tokenId": 26 }, "tokenId": 25 }, { "type": "BreakStatement", "tokenId": 28 }] }</pre>
<p>ExpressionStatement</p> <p>ExpressionStatement</p>  <p>exprs</p>	<pre>// x++; { "type": "ExpressionStatement", "exprs": [{ "type": "PostfixExpression", "expr": { "type": "Identifier", "value": "x", "tokenId": 31 }, "op": {</pre>

	<pre> "value": "++", "tokenId": 32 } }] }</pre>
<p>SelectionStatement</p> <p>SelectionExpression</p>  <pre>graph TD SE((SelectionExpression)) --> C((cond)) SE --> T((then)) SE --> O((otherwise))</pre>	<pre>// if (flag) break; else x++; { "type": "SelectionStatement", "cond": [{ "type": "Identifier", "value": "flag", "tokenId": 36 }], "then": { "type": "BreakStatement", "tokenId": 38 }, "otherwise": { "type": "ExpressionStatement", "exprs": [{ "type": "PostfixExpression", "expr": {</pre>

	<pre> "type": "Identifier", "value": "x", "tokenId": 41 }, "op": { "value": "++", "tokenId": 42 } }] }</pre>
<p>IterationStatement</p> <p>IterationExpression</p>  <pre>graph TD IE([IterationExpression]) --> init([init]) IE --> cond([cond]) IE --> step([step]) IE --> stat([stat])</pre>	<pre>// for(i=0; i<3; i++) { } { "type": "IterationStatement", "init": [{ "type": "BinaryExpression", "op": { "value": "=", "tokenId": 8 }, }, "expr1": { "type": "Identifier", "value": "i", "tokenId": 7 }] }</pre>

```
    },  
    "expr2": {  
      "type": "IntegerConstant",  
      "value": 0,  
      "tokenId": 9  
    }  
  },  
],  
"cond": [  
  {  
    "type": "BinaryExpression",  
    "op": {  
      "value": "<",  
      "tokenId": 12  
    },  
    "expr1": {  
      "type": "Identifier",  
      "value": "i",  
      "tokenId": 11  
    },  
    "expr2": {  
      "type": "IntegerConstant",  
      "value": 3,  
      "tokenId": 13  
    }  
  }  
]
```

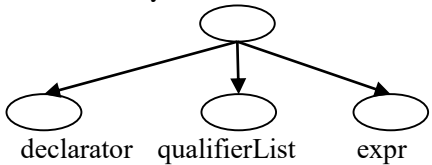
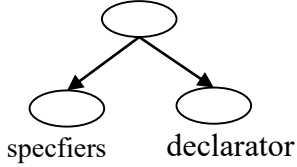

	<pre>], "step": [{ "type": "PostfixExpression", "expr": { "type": "Identifier", "value": "i", "tokenId": 15 }, "op": { "value": "++", "tokenId": 16 } }], "stat": { "type": "CompoundStatement", "blockItems": [] } } </pre>
IterationDeclaredStatement	<pre> // for(int i = 0; i<3; i++) {} { "type": "IterationDeclaredStatement", "init": { "type": "Declaration", "specifiers": [</pre>

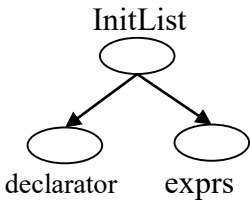


```
{
  "value": "int",
  "tokenId": 22
},
"initLists": [
  {
    "declarator": {
      "type": "VariableDeclarator",
      "identifier": {
        "type": "Identifier",
        "value": "i",
        "tokenId": 23
      }
    },
    "exprs": [
      {
        "type": "IntegerConstant",
        "value": 0,
        "tokenId": 25
      }
    ]
  }
],
"cond": [
```

	<pre>{ "type": "BinaryExpression", "op": { "value": "<", "tokenId": 28 }, "expr1": { "type": "Identifier", "value": "i", "tokenId": 27 }, "expr2": { "type": "IntegerConstant", "value": 3, "tokenId": 29 } },], "step": [{ "type": "PostfixExpression", "expr": { "type": "Identifier", "value": "i", "tokenId": 31 } }, </pre>
--	---

	<pre> "op": { "value": "++", "tokenId": 32 } }, "stat": { "type": "CompoundStatement", "blockItems": [] } } </pre>
<p>VariableDeclarator</p> <p>VariableDeclarator</p>  <pre> graph TD A([VariableDeclarator]) --> B([identifier]) </pre> <p>identifier</p>	<pre> { "type": "VariableDeclarator", "identifier": { } } </pre>
<p>ArrayDeclarator</p>	<pre> { "type": "ArrayDeclarator", "declarator": { }, "qualifierList": [], "expr": { } } </pre>

<p>ArrayDeclarator</p>  <pre>graph TD; A([ArrayDeclarator]) --> B([declarator]); A --> C([qualifierList]); A --> D([expr]);</pre>	<pre>} }</pre>
<p>FunctionDeclarator</p>  <pre>graph TD; A([FunctionDeclarator]) --> B([declarator]); A --> C([params]);</pre>	<pre>{ "type": "FunctionDeclarator", "declarator": { }, "params": [] }</pre>
<p>ParamsDeclarator</p>  <pre>graph TD; A([ParamsDeclarator]) --> B([specifiers]); A --> C([declarator]);</pre>	<pre>{ "specifiers": [], "declarator": }</pre>

<div>InitList</div> <div><pre>graph TD; InitList([InitList]) --> declarator([declarator]); InitList --> exprs([exprs]);</pre></div>	<div><pre>{ "declarator": { }, "exprs": [] }</pre></div>
<div>Declaration</div>	<div><pre>{ "type": "Declaration", "specifiers": [], "initLists": [] }</pre></div> <div>三种情况对应的样例如下: // int a, b=1; { "type": "Declaration", "specifiers": [{ "type": "Token", "value": "int",</div>

	<pre> "tokenId": 14 }], "initLists": [{ "type": "InitList", "declarator": { "type": "VariableDeclarator", "identifier": { "type": "Identifier", "value": "a", "tokenId": 15 } }, "exprs": [] }, { "type": "InitList", "declarator": { "type": "VariableDeclarator", "identifier": { "type": "Identifier", "value": "b", "tokenId": 17 } } }],</pre>
--	---

	<pre> "exprs": [{ "type": "IntegerConstant", "value": 1, "tokenId": 19 }] }] } // int a[2] = {1,2}; "type": "Declaration", "specifiers": [{ "type": "Token", "value": "int", "tokenId": 21 }], "initLists": [{ "type": "InitList", "declarator": { "type": "ArrayDeclarator", "declarator": { "type": "VariableDeclarator",</pre>
--	--

	<pre> "identifier": { "type": "Identifier", "value": "a", "tokenId": 22 }, "expr": { "type": "IntegerConstant", "value": 2, "tokenId": 24 }, }, "exprs": [{ "type": "IntegerConstant", "value": 1, "tokenId": 28 }, { "type": "IntegerConstant", "value": 2, "tokenId": 30 }] }</pre>
--	--

```
}

// void fun(double , int a); // 例子只是为了说明，第一个是抽象参数，第二个是
// 普通形参
{
    "type": "Declaration",
    "specifiers": [
        {
            "type": "Token",
            "value": "void",
            "tokenId": 0
        }
    ],
    "initLists": [
        {
            "type": "InitList",
            "declarator": {
                "type": "FunctionDeclarator",
                "declarator": {
                    "type": "VariableDeclarator",
                    "identifier": {
                        "type": "Identifier",
                        "value": "fun",
                        "tokenId": 1
                    }
                }
            }
        }
    ],
}
```

	<pre>"params": [{ "type": "ParamsDeclarator", "specifiers": [{ "type": "Token", "value": "double", "tokenId": 3 }], "declarator": null }, { "type": "ParamsDeclarator", "specifiers": [{ "type": "Token", "value": "int", "tokenId": 5 }], "declarator": { "type": "VariableDeclarator", "identifier": { "type": "Identifier", "value": "a",</pre>
--	---

	<pre> "tokenId": 6 } } },], "identifiers": null, "subtype": "paramtype" }, "exprs": [] }] } } </pre>
FunctionDefine	<pre> // int main() { } { "type": "FunctionDefine", "specifiers": [{ "type": "Token", "value": "int", "tokenId": 0 }], "declarator": { "type": "FunctionDeclarator", "declarator": { "type": "VariableDeclarator", </pre>

	<pre> "identifier": { "type": "Identifier", "value": "main", "tokenId": 1 }, "params": null, "identifiers": [], "subtype": "identifiertype" }, "declarations": [], "body": { "type": "CompoundStatement", "blockItems": [] } }</pre>
--	---

在 `bit.minisys.minicc.paser.ast` 包中提供了上述所有 AST 节点对应的类, 推荐将自己实现输出的语法分析树转换至上述对应的 AST 类构成的抽象语法树, 再使用 Jackson 库(或其他)自动生成 JSON 文件。其中 `ASTCompilationUnit` 类对应 Json 树的根节点, 如下为样例代码:

```
ObjectMapper mapper = new ObjectMapper();  
mapper.writeValue(new File(filename), root);
```

`filename` 为输出文件名, `root` 为 `ASTCompilationUnit` 类的对象, 即生成的 AST 的根结点。

实验目的

- (1) 熟悉 C 语言的语法规则, 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。

实验内容

该实验选择 C 语言的一个子集, 基于 BIT-MiniCC 构建 C 语法子集的语法分析器, 该语法分析器能够读入词法分析器输出的存储在文件中的属性字符流, 进行语法分析并进行错误处理, 如果输入正确时输出 JSON 格式的语法树, 输入不正确时报告语法错误。

实验过程与方法

在 BIT-MiniCC 框架下, 可以按照如下步骤完成语法分析实验:

- (1) 参照后续给出的文法, 扩充定义自己希望实现的 C 语言语法子集。参考文法只给出了函数定义以及简单的表达式相关的文法。局部变量声明、分支语句以及循环语句等需要自己进行扩充。采用自顶向下的分析方法时, 不能有左递

归,避免文法产生式的多个候选式存在公共因子。如果出现左递归或者公共因子,则可以通过文法等价变换进行消除。也可以使用实验 3 中自己完成的文法。

(2) 从递归下降分析方法、LL(1)分析方法、LR 分析方法中选择一种算法,基于 BIT-MiniCC 设计并实现语法分析器。可以使用 ANTLR,也可以手动编码实现。

(3) 手动构建语法分析器时,BIT-MiniCC 中已经定义了一个示例实现 ExampleParser.java,使用了递归下降分析法。语法分析的输入为词法分析的输出,因此语法分析器首先要读入 xxx.tokens 文件;在分析的过程中构建语法树。

(4) 将语法树输出为 JSON 文件;

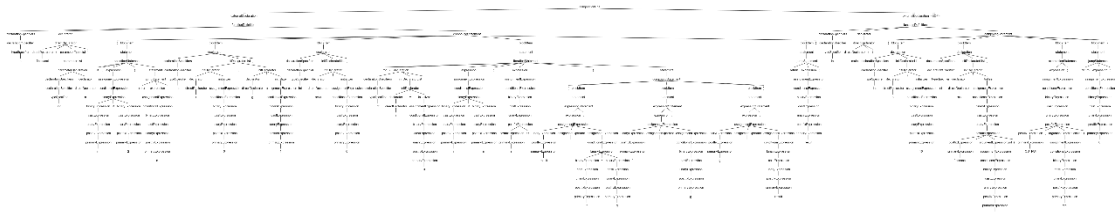
目前已有的分析方法包括递归下降、LL(1)和 LR 等多种分析方法,可以选择其中的一种实现,递归下降更为直观。

例如,基于框架自带的文法及其对应的实现,当输入为如下的程序时:

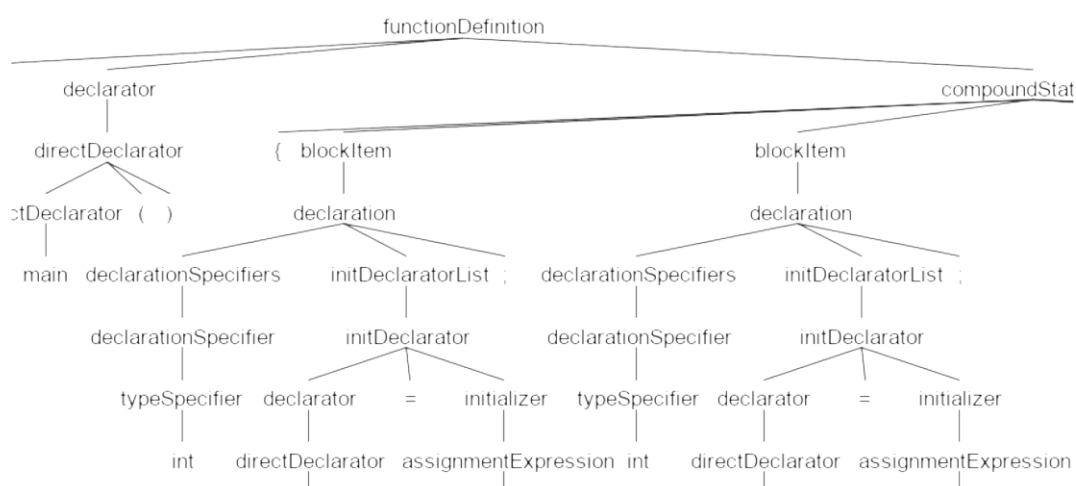
```
int fibonacci ( int n ) {
    if ( n < 2 )
        return n;
    int f = 0 , g = 1 ;
    int result = 0 ;
    for ( int i = 0 ; i < n ; i ++ ) {
        result = f + g ;
        f = g ;
        g = result ;
    }
    return result ;
}

int main ( ) {
    int a = 10;
    int res = fibonacci ( a ) ;
    _OUTPUT ( res ) ;
    return 0 ;
}
```

得到的语法树如下所示:



部分细节如下所示：



对应的输出的 JSON 的部分内容为：

```

1 {
2   "type": "Program",
3   "items": [
4     {
5       "type": "FunctionDefine",
6       "specifiers": [
7         {
8           "type": "Token",
9           "value": "int",
10          "tokenId": 0
11        }
12      ],
13      "declarator": {
14        "type": "FunctionDeclarator",
15        "declarator": {
16          "type": "VariableDeclarator",
17          "identifier": {
18            "type": "Identifier",
19            "value": "fibonacci",
20            "tokenId": 1
21          }
22        },
23        "params": [
24          {
25            "type": "ParamsDeclarator",
26            "specifiers": [
27              {
28                "type": "Token",
29                "value": "int",
30                "tokenId": 3
31              }
32            ]
33          }
34        ]
35      }
36    ]
37  }
38 }

```

需要说明的是，能够分析的输入程序依赖于选用的文法子集，而输出的语法树的结构又与文法的定义密切相关。

如下为 C 语言文法的一个子集：


```

CMPL_UNIT → FUNC_LIST
FUNC_LIST → FUNC_DEFFUNC_LIST | ε
FUNC_DEF → TYPE_SPEC ID (ARG_LIST ) CODE_BLOCK
TYPE_SPEC → int | void
PARA_LIST → ARGUMENT | ARGUMENT , PARA_LIST | ε
ARGUMENT → TYPE_SPEC ID
CODE_BLOCK → { STMT_LIST }
STMT_LIST → STMT STMT_LIST | ε
STMT → RTN_STMT | ASSIGN_STMT
RTN_STMT → return EXPR
ASSIGN_STMT → ID = EXPR
EXPR → TERM EXPR2
EXPR2 → + TERM EXPR2 | - TERM EXPR2 | ε
TERM → FACTOR TERM2
TERM2 → * FACTOR TERM2 | / FACTOR TERM2 | ε
FACTOR → ID | CONST | ( EXPR )

```

读者可以在此基础之上进行文法扩充，包括全局变量声明，循环语句、分支语句、函数调用语句以及 switch 语句等。要求至少包括局部变量声明语句、赋值语句、返回语句、一种分支语句（if, if-else, switch 等）和一种循环语句（for, while, do-while 等）。

注意事项:

- 示例程序 ExampleParser.java 与内部自带的实现是两个不同的程序，建议在 ExampleParser.java 的基础上扩展增加代码完成自己的语法分析器，也可以重新创建一个类；
- 运行示例程序时要修改 config.xml 文件中的 parser, 将 path 设置为 “bit.minisys.minicc.parser.xxx”, 框架才会调用对应的代码，其中 xxx 对应语法分析类名。当 path 为空时会自动调用内部实现。
- Test 文件夹下面的 0-example-test.c 是用于 ExampleScanner 和 ExampleParser 的测试程序，由于 ExampleScanner 和 ExampleParser 功能比较弱，使用其他测

试用例会出错;

- Test 文件夹下面的 2_paser-test1.c、3_paser-test2.c、4_paser-test3.c 可以用于内部自带的语法分析器测试运行, 将 config.xml 文件中 parser 的 path 设置为 “”, 并将输入程序改为上述 3 个之一就会看到可视化的输出, 以及对应的 JSON 输出。

实验提交内容

本实验要求提交语法分析器实现源码 (项目 src 下所有文件), 以及对应的 config.xml 和 classpath 两个文件; C/C++ 需提供对应的可执行程序 (不需要编译的中间文件), Java 提供编译后的可运行的 jar 包, 每个人提交一份实验报告。

实验报告放置在 doc 目录下, 应包括如下内容:

- 实验目的和内容
- 实现的具体过程和步骤
- 运行效果截图
- 实验心得体会