**编译原理与设计**

**实验报告**

实验名称： Lab 5：语法分析实验

姓名/学号： 李昊阳/1120203053

1. **实验目的和内容**

**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。

具体步骤如实验指导文档所示。

1. **实验环境**

设备：RedmiBook 14 锐龙版

操作系统：Windows 10 Pro, 64-bit (Build 19045.2604) 10.0.19045

Java：openjdk version "1.8.0\_312"

ANTLRWorks：antlrworks-1.5.1(including ANTLR 3.5rc2)

1. **实现的具体过程和步骤**

按照实验文档的指示：

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。

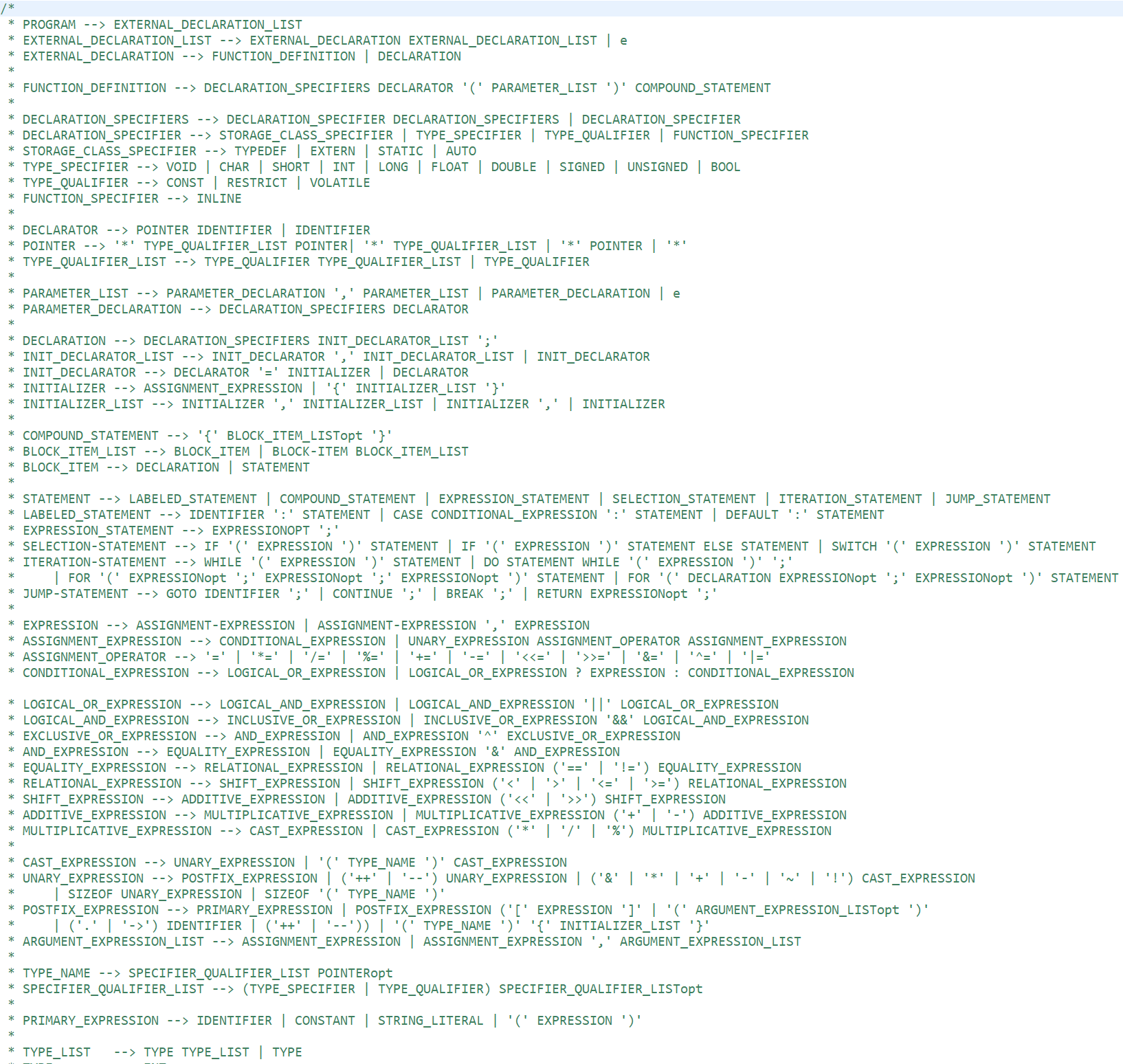
（4）将语法树输出为 JSON 文件；目前已有的分析方法包括递归下降、LL(1)和 LR 等多种分析方法，可以选择其中的一种实现，递归下降更为直观。

在综合考虑多种方法后，选择在 BIT-MiniCC 框架下，通过扩写ExampleParser 类，手动编程来完成语法分析实验。

首先，词法与文法定义都参考了 C11-n1570 标准文档，并作了少量修改，删去了不常用的语法部分，保留了所有类型的语句与表达式。

在设计语法时，修改了标准语法中不符合自顶向下分析方法的部分，将其从左递归改为右递归或由循环表示，同时避免文法产生式的多个候选式存在公共因子。

以下为主要语法：

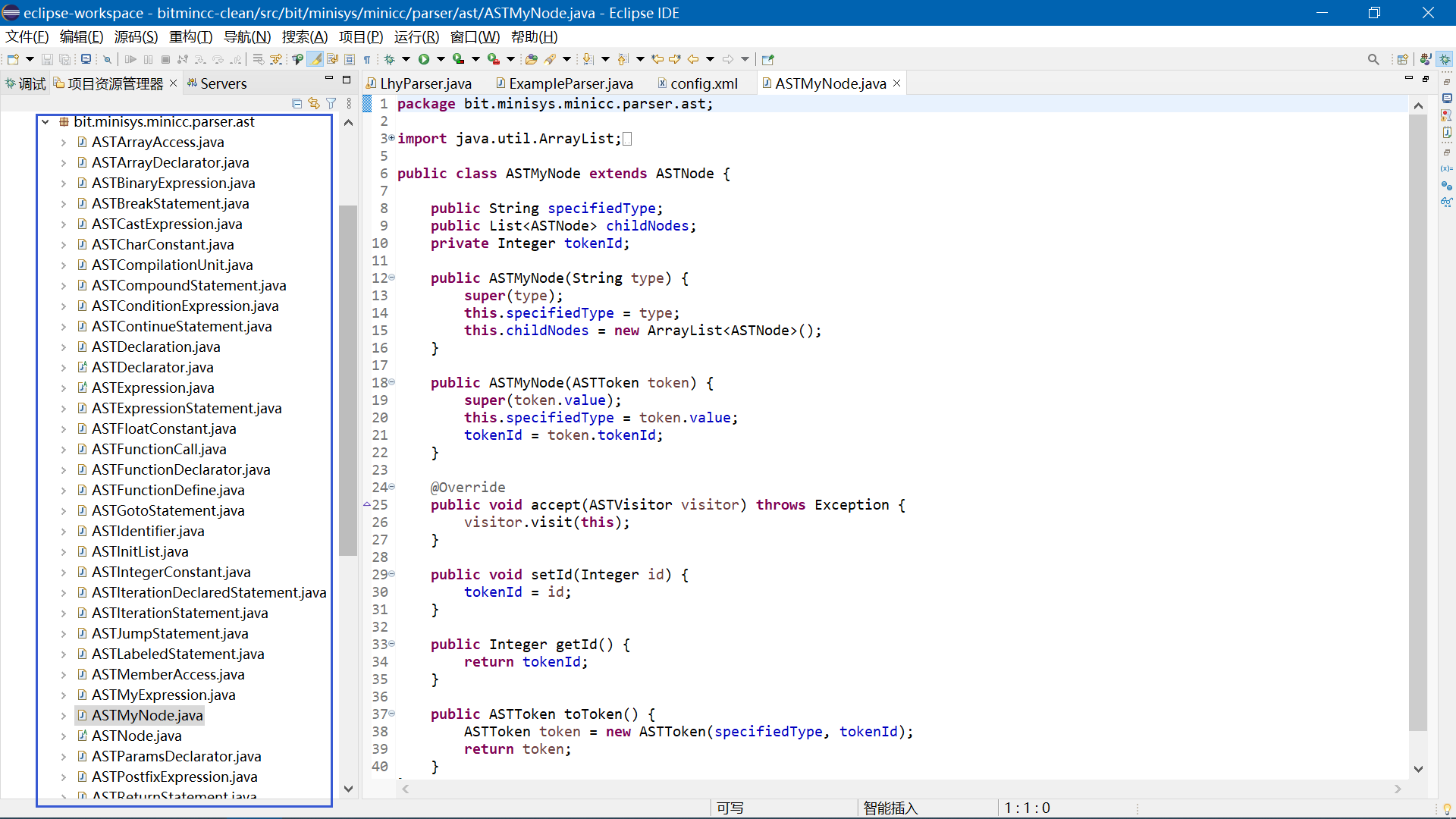
从语法可见，语法分析程序从PROGRAM进入，并在EXTERNAL\_DECLARATION处区分FUNCTION\_DEFINITION与DECLARATION。进而定义了DECLARATOR、DECLARATION以及众多STATEMENT、EXPRESSION。

DECLARATION\_SPECIFIER声明说明符进一步定义了STORAGE\_CLASS\_SPECIFIER储存类别说明符、TYPE\_SPECIFIER类型说明符、TYPE\_QUALIFIER类型限定符以及函数说明符FUNCTION\_SPECIFIER。DECLARATOR声明符也增含了指针POINTER与IDENTIFIER。

在STATEMENT中，定义了标记语句LABELED\_STATEMENT、表达式语句EXPRESSION\_STATEMENT、选择语句SELECTION\_STATEMENT、循环语句ITERATION\_STATEMENT、跳转语句JUMP\_STATEMENT。

在EXPRESSION中，定义了更多的表达式，通过分析递归的先后顺序来区分运算符的优先级。主要包括赋值表达式ASSIGNMENT\_EXPRESSION、条件表达式CONDITIONAL\_EXPRESSION、强制转换表达式CAST\_EXPRESSION、一元表达式UNARY\_EXPRESSION、后缀表达式POSTFIX\_EXPRESSION、基本表达式PRIMARY\_EXPRESSION等等。

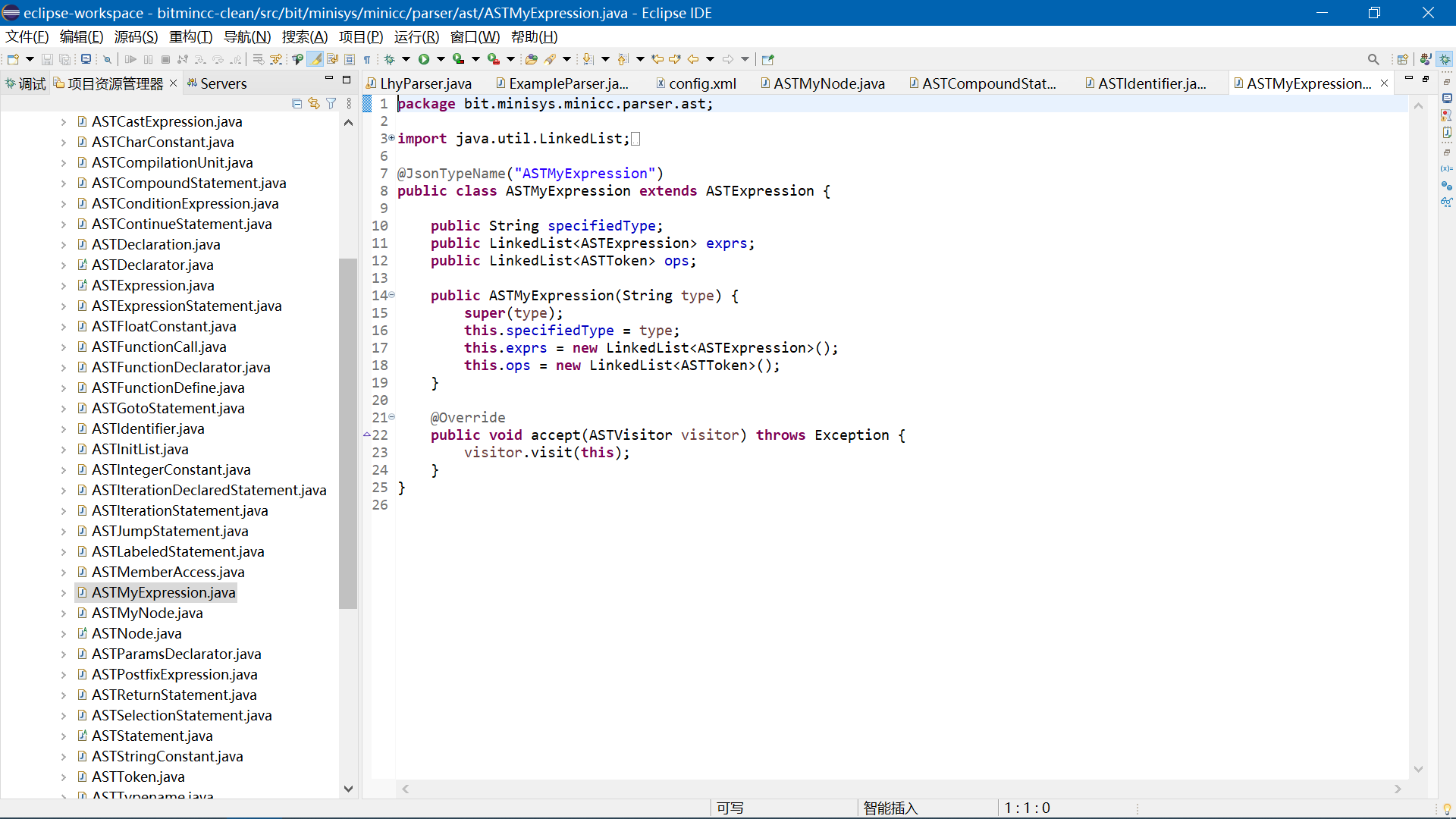
根据实验文档的指示，在使用BIT-MiniCC框架时，在 bit.minisys.minicc.paser.ast 包中提供了实验文档内所述的所有 AST 节点对应的类，可以将自己实现输出的语法分析树转换至上述对应的 AST 类构成的抽象语法树，再使用 Jackson 库(或其他)自动生成 JSON 文件。

bit.minisys.minicc.parser.ast 包中包括如下节点（见下图左侧）：

可见，框架内已经给出了大量我们语法分析构建语法树所需要的结点，如ASTFunctionDefine、ASTCompoundStatement、ASTIdentifier等等。但是，在应用在我所定义的语法中，结点类型还有所不足，且ASTToken结点无法在语法树中展示token的值，所生成的语法树不够直观、完整。因此扩充定义了ASTMyNode（见上图右侧）与ASTMyExpression（见下图）。

ASTMyNode继承自抽象类ASTNode，既可以作为双亲结点，完善语法层次；也可以作为叶结点，直观地表现token内容。specifiedType字段代表当前实例特指的结点类型，childNodes储存子结点，便于在JSON中展示，tokenId储存tokenIndex，该值只在作为叶结点的时候被赋值。

ASTToken与ASTMyNode结点可通过公有的tokentoMyNode()、myNodetoToken()方法或new ASTMyNode(token)、myNode.toToken()来相互转换。

ASTMyExpression继承自抽象类ASTExpression。由于ASTExpression子类过少，无法延伸出完整的表达式分析语法树，因此拓展定义该子类。specifiedType字段同样表示该实例的指定类型，exprs与ops则储存分析后的子表达式及运算符。

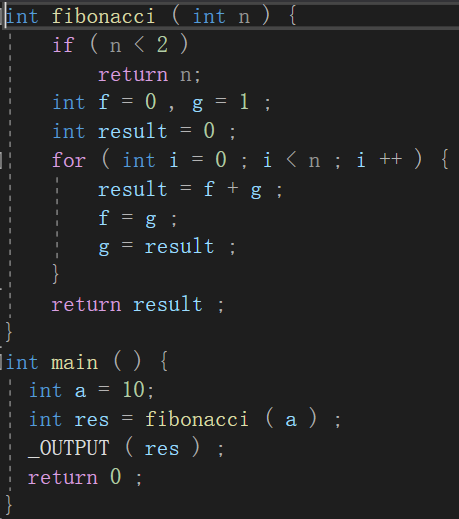
以下为示例部分代码：

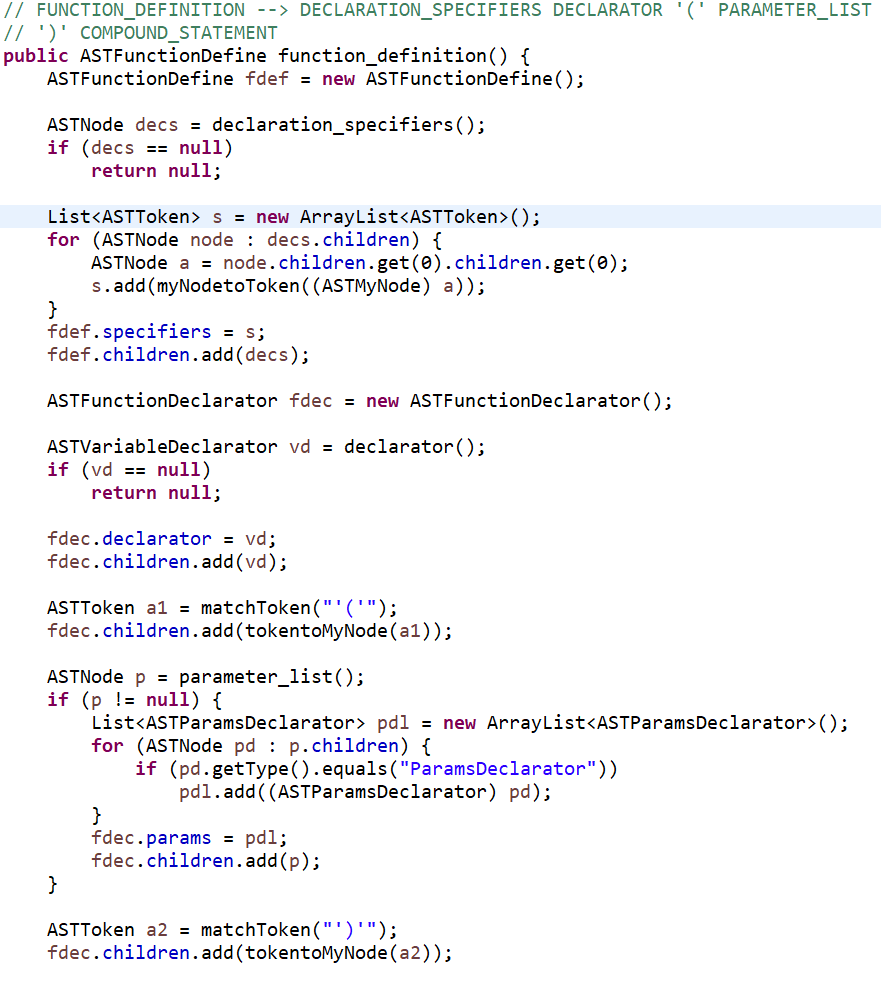
1. *// PROGRAM --> EXTERNAL\_DECLARATION\_LIST*
2. public ASTNode program() {
3. ASTCompilationUnit p = new ASTCompilationUnit();
4. ASTNode e = external\_declaration\_list();
5. if (e != null) {
6. p.items.add(e);
7. p.children.add(e);
8. }
9. return p;
10. }
11. *// EXTERNAL\_DECLARATION\_LIST --> EXTERNAL\_DECLARATION EXTERNAL\_DECLARATION\_LIST*
12. *// | e*
13. public ASTMyNode external\_declaration\_list() {
14. ASTMyNode myNode = new ASTMyNode("externalDeclarationList");
15. nextToken = tknList.get(tokenIndex);
16. while (true) {
17. if (nextToken.type.equals("EOF"))
18. break;
19. ASTNode e = external\_declaration();
20. if (e != null) {
21. myNode.children.add(e);
22. myNode.childNodes.add(e);
23. } else if (!myNode.children.isEmpty())
24. return myNode;
25. else
26. break;
27. }
28. return null;
29. }

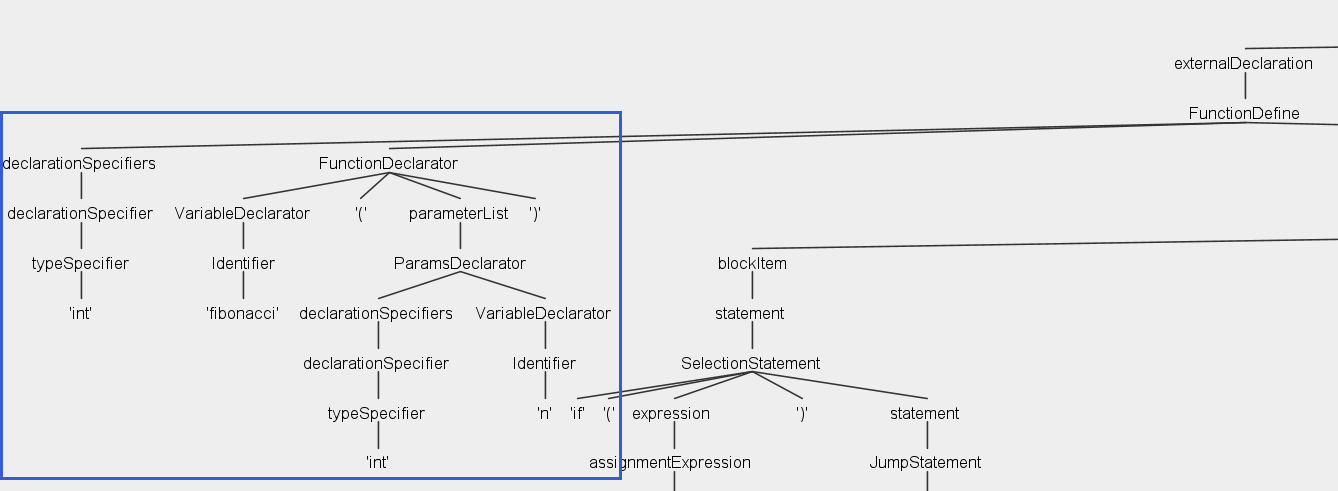
由此可见，在代码编写过程中，我为每一类结点都定义了一个公有方法，以此来生成该结点，并且返回该结点实例。在方法中，会根据语法定义调用对应子结点函数，以生成其子结点，并将其添加至子结点链表children以及对应字段，如ASTCompilationUnit包含字段List<ASTNode> items。

1. **运行效果截图**

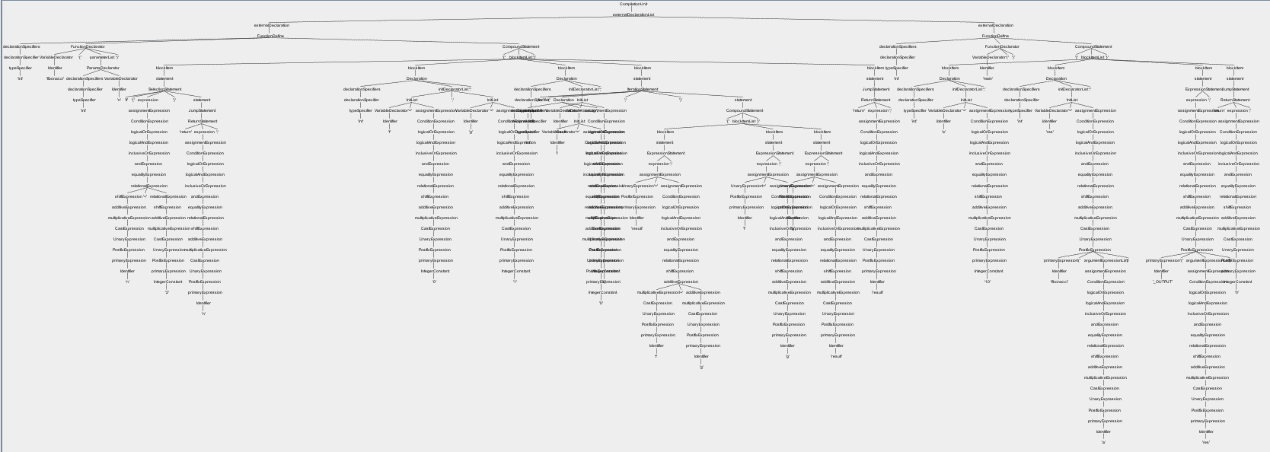
以下为函数定义结点为例，展示语法树与生成的json文件，目标代码使用示例2\_parser\_test1.c。

目标代码：

parser代码（仅包括函数声明与参数部分）：

 该部分生成的语法树如下：

 对应的输出的json内容为：

 分析所得的完整语法树如下所示：

1. **实验心得体会**

本次实验是编译原理与设计的第五次实验，本次实验我选择使用了手动编码实现，而非使用第四次实验所用的Antlr。本次实验让我进一步认识到了语法分析过程，更加深入地阅读了C语言的规范文件。从代码编写的角度上，深入了解了语法分析器的工作原理。

本次实验花费时间极长，首先必须阅读ExampleParser类是如何编写的，了解如何在BIT-MiniCC框架下编写语法分析器，并熟练掌握所有AST工具类。还需在不熟悉的eclipse IDE下进行Java编程，难度极大，用时极长。因此本次实验也是本人倾注精力的一次实验，从环境到调试，从词法到文法，从使用到创造，从tokens最终生成json，收获颇丰，实力有了长足长进。

\*需要注意的是，declarator部分在语法设计时，疏漏了数组类型数据（如int a[5]，‘[’会无法识别），因此本程序无法分析3\_parser\_test2.c示例，而其他三个示例均可运行。