天津大学

编译技术实验报告

题目：语法分析树的生成

学　　院　计算机科学与技术

年　　级　本科 2010 级

班　　级　计算机 4 班

指导老师　胡 静

组　　员　张旭（组长）

赵凯 路遥 王维 邹伟伟

2013 年 6 月 8 日

目录

1 实验目的

编译技术是理论与实践并重的课程，实验需要综合运用二三年级所学的多门课程内容，用来完成一个小的编译程序，从而巩固和加强对词法分析、语法分析、语义分析等理论的认识和理解。

实验的第一部分要求设计、编制并调试一个词法分析和语法分析程序，加深对编译前端原理的理解。

2 实验成员及贡献

张旭：统筹组内各项工作，精简了 C 语言的语法文件，研究出生成语法分析树的方法，对实验报告进行了整体的修改校订。

路遥：搜集 C.g 的原型模板；撰写环境搭建、c 语言语法规则说明、语法树生成原理；修改实验目的，调整部分排版。

赵凯：收集了大量关于词法分析和语法分析的资料，协调组内成员完成了实验报告，在报告中对我组的代码进行了详细的解释。

王维：在实验报告中，对实验结果进行了展示，协助组长完成了 g 文件的整理工作。

邹伟伟：收集 antlr4 的相关资料，协助组长完成了语法树的生成。

3 环境配置

3.1 运行环境配置

运行本程序前，需要简单地配置一下 antlr4 的环境变量，本章第 3 节中有详细配置步骤。

1、打开 cmd。方法：在“开始”菜单里输入 cmd，点击回车键，即可启动 cmd。

2、使用 cd 命令打开可执行程序所在的文件夹，比如:

cd /d D:\学习\旭之课件\大三下\编译技术\release

3、运行主类，比如：

java Main

输入源文件名，比如：

t1.c

4、运行时，自动弹出语法分析树，四元式结果保存在“Quaternary”文件中，请自行用编辑器（devcpp，vim，codeblocks，vs 等编辑器都行）查看，建议不要用记事本。

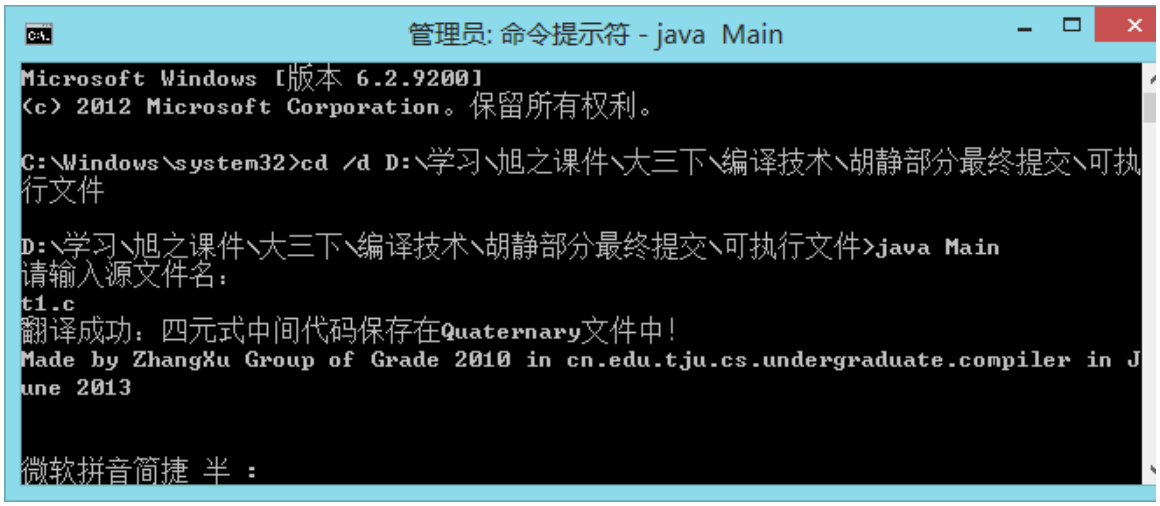

图 3-1 可执行文件的运行方法示例图

3.2 开发环境配置

编译本项目之前，需要简单地配置一下 antlr4 的环境变量，下文有详细配置步骤。

1、打开 cmd。

2、使用 cd 命令打开 java 源文件所在的文件夹。

3、编译 g 文件(antlr4 C.g)的命令行语句示例如下：

java -jar D:\workspace\java\lib\antlr-4.0-complete.jar C.g

注意：我们提交的源代码中已经编译过 g 文件了，您可以直接忽略此步。您如需重新编译 g 文件，请先对 CBaseListener.java 文件做好备份，因为我们的代码大都写在这个文件里。重新编译 g 文件会覆盖此文件，造成代码丢失。

4、编译 java 源文件，命令行示例如下：

javac *.java

5、接下来就是运行程序了，请参照本章的第 1 节。

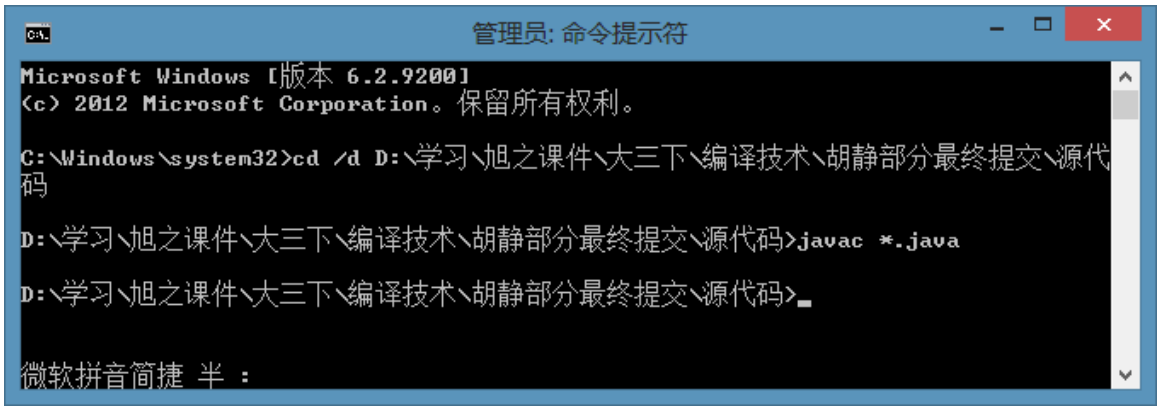

图 3-2 开发环境配置示例图

3.3 antlr 环境配置

默认已经配置好了 java 环境变量。Antlr4 的配置方法，就是在 CLASSPATH 环境变量里添加 antlr-4.0-complete.jar 的路径。比如：

“;.;D:\workspace\java\lib\antlr-4.0-complete.jar”

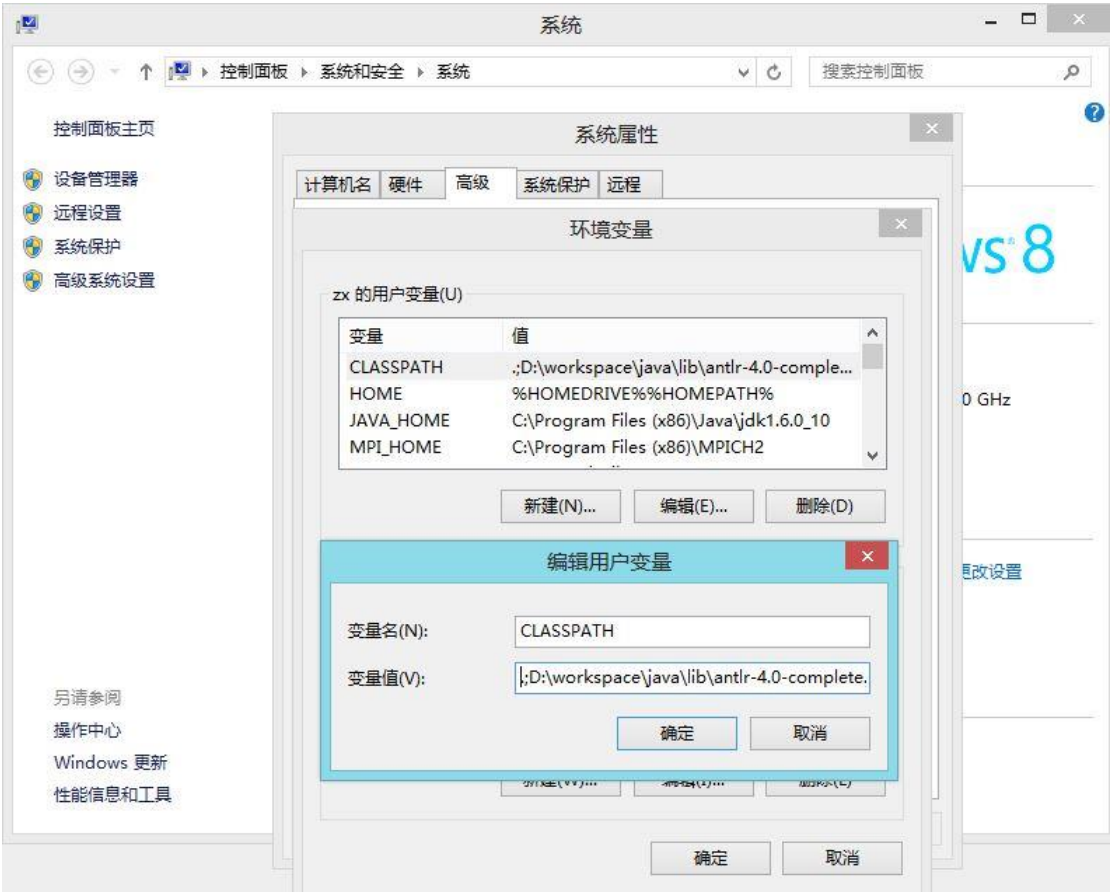

图 3-3 antlr4 的环境变量的配置

4 实验要求

4.1 词法分析

单词被分为以下几大类：

（1）关键字（所有的关键字都是小写），比如：

```
if   while  int float break continue
```

（2）运算符和界符，比如：

```
:  =  +  -  *  /  <  <=  <>  >  >=  =  ;  (  )  #
```

（3）标识符（IDENTIFIER）、整型常数、浮点数、字符串等常量。例如，标示符的定义如下：

```
    IDENTIFIER
:   LETTER (LETTER|'0'..'9')*
;
```

（4）空格有空白、制表符和换行符组成。空格一般用来分隔以上几种类型的词汇，空格在词法分析阶段通常被忽略。

(5)注释包括多行注释和单行注释，词法分析器会直接忽略注视。

这个语言的部分的单词符号，以及它们的种别编码如下表：

单词符号	种别编码	助记符
IF	1	$IF

标识符	2	$ID
常数（整）	3	$INT
=	4	$ASSI
+	5	$PLUS
*	6	$STAR
**	7	$POW
,	8	$COM
（	9	$LPAR
）	10	$RPAR

表 4-1 单词符号的种别编码

对于这个语言，有几点重要的限制：

首先，所有的关键字（如 IF、WHILE 等）都是“保留字”。所谓的保留字的意思是，用户不得使用它们作为自己定义的标示符。例如，下面的写法是绝对禁止的：

IF = 5

其次，由于把关键字作为保留字，故可以把关键字作为一类特殊标示符来处理。也就是说，对于关键字不专设对应的转换图。但把它们（及其种别编码）预先安排在一张表格中（此表叫作保留字表）。当转换图识别出一个标识符时，就去查对这张表，确定它是否为一个关键字。

因为对于后者，我们的分析器将无条件地将 IFI 看成一个标识符。

这个语言的单词符号的状态转换图，如下图：

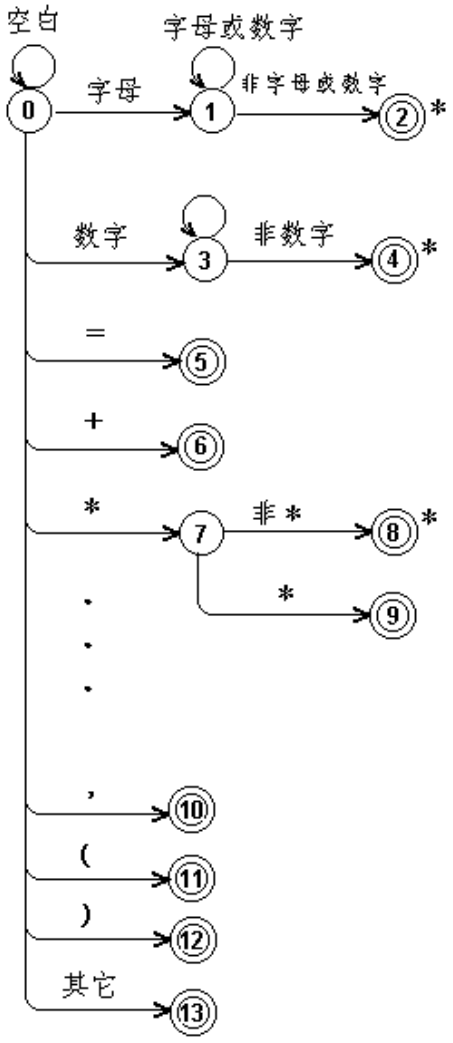

图 4-1 单词符号的状态转换图

4.2 语法分析

能识别 C 语言的各种语法，比如算术表达式的文法定义如下：

```
expression
    :   primary_expression
    |   IDENTIFIER op='(' expressionList? ')'
    |   op=('+'|'-'|'++'|'--') expression
    |   op=('~'|'!') expression
    |   expression op=('*'|'/'|'%') expression
    |   expression op=('+'|'-') expression
    |   expression op=('<=' | '>=' | '>' | '<') expression
    |   expression op=('==' | '!=') expression
    |   expression op='&&' expression
    |   expression op='||' expression
    |   IDENTIFIER op='='<assoc=right> expression
```

;

使用的语法分析算法可以是：预测分析法；递归下降分析法；算符优先分析法；LR 分析法等。我们采用的自顶向下的 LL 分析法。

5 概要设计及重点原理介绍

5.1 定义 C 语言子集的文法

我们是在 C.g 文件里定义 C 语言的文法的。部分 C 语言语法规则定义如下（详见 C. g 文件）：

```
grammar C;

// starting point for parsing a c file
file:   (functionDecl | varDecl)+ ;

varDecl
   // :   type_specifier IDENTIFIER ('=' expression)? ';'
    :   type_specifier  expression';'
    ;

functionDecl
    :   type_specifier IDENTIFIER '(' formalParameters? ')' block // "void f(int x) {...}"
    ;
formalParameters
    :   formalParameter (',' formalParameter)*
    ;
formalParameter
    :   type_specifier IDENTIFIER
    ;

block:  '{' statement* '}' ;   // possibly empty statement block


type_specifier
    : 'void'
    | 'char'
    | 'int'
    | 'float'
    | 'double'
    ;
```

图 5-1 语法文件的根节点部分

从上图中可以看出，我们的语法分析的根节点是 file，从 file 的产生式可以看出，file 是由一系列全局变量和函数组成。type_specifier 定义了程序中的各种数据类型。

```
constant
    :   HEX_LITERAL
    |   OCTAL_LITERAL
    |   DECIMAL_LITERAL
    |   CHARACTER_LITERAL
    |   STRING_LITERAL
    |   FLOATING_POINT_LITERAL
    ;

//Statements

statement
    : expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    | varDecl
    | block
    ;

expression_statement
    : ';'
    | expression ';'
    ;

selection_statement
    : if_statement statement else_statement?
    ;

if_statement
    : 'if' '(' expression ')'
    ;
```

图 5-2 statement 的语法定义

5.2 词法分析与语法分析的实现

编译过程采用一趟扫描的方式，以语法分析程序为核心，词法分析程序和代码生成程序都作为一个独立的过程，当语法分析需要读单词就调用词法分析程序，而当语法分析正确需生成相应的目标代码时，则调用代码生成程序。此外，用表格管理程序建立变量、常量和过程标识符的说明与引用之间的信息联系；用出错处理程序对词法和语法分析遇到的错误给出在源程序中出错的位置和错误性质；然后我们用 java 画出了一棵语法分析树，如果源程序有语法错误的话，我们会在树中用红色标记错误位置。

5.2.1 词法分析

词法分析要识别的符号包括以下几种：

1） 运算符： 如：+、-、*、/、=等

2） 标识符： 用户定义的变量名、常数名

3） 常数： 如：整数、浮点数、字符串等

4） 界符： 如：‘;’ 、‘(’ 、‘)’等

5） 函数调用：如 sin ， cos ， tan 等。

词法分析器的流程如下图：

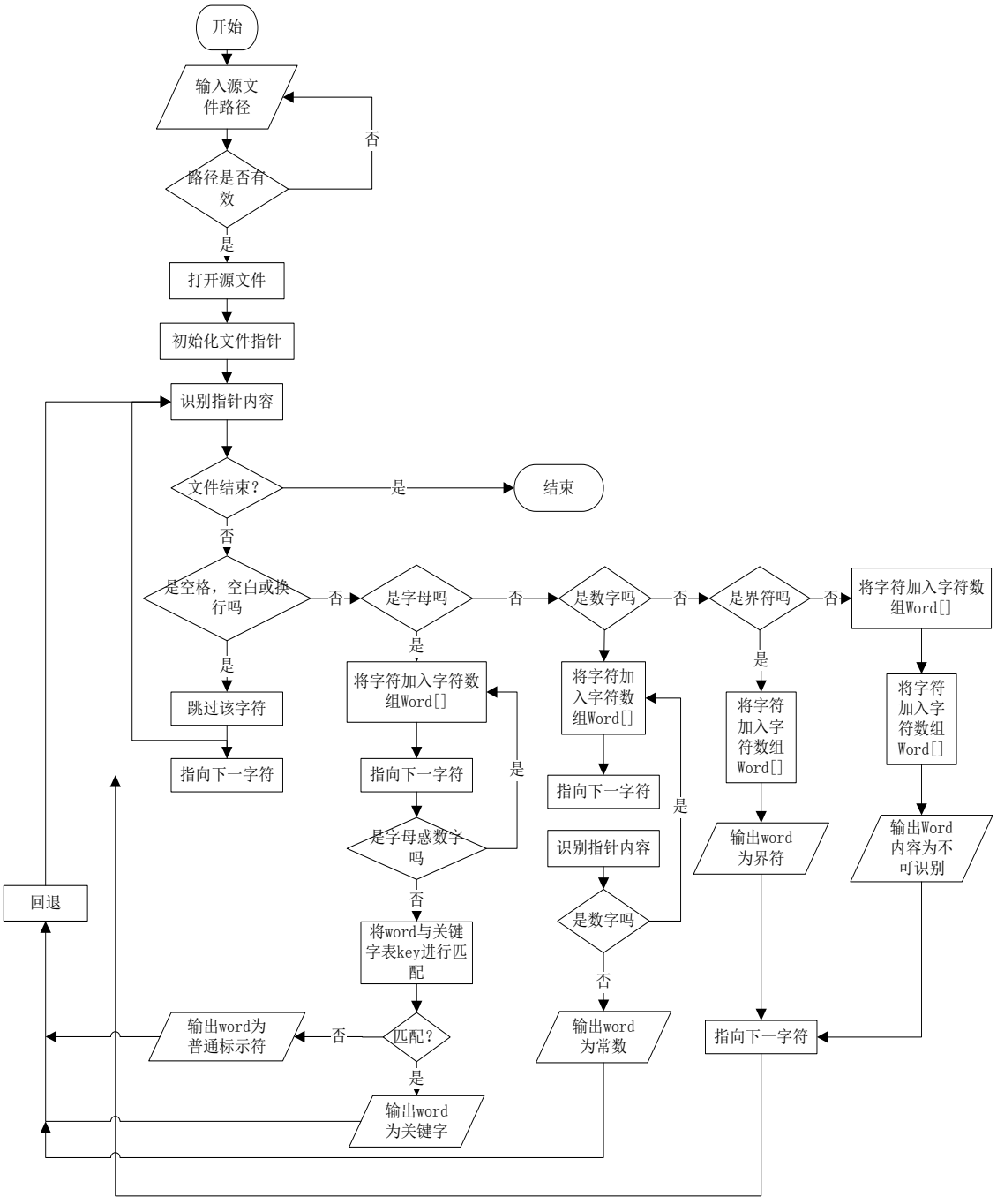

图 5-3 词法分析器的流程图

5.2.3 语法分析器主程序

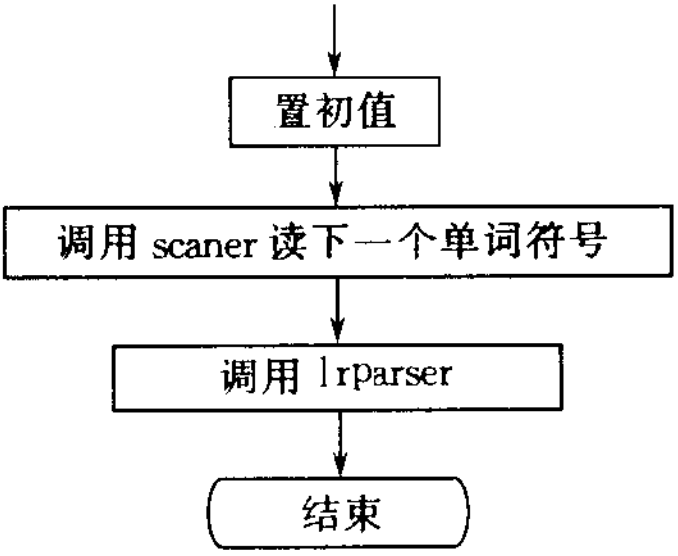

图 5-4 语法分析器主程序图

5.2.3 编译程序语法分析的设计与实现

递归子程序法：对应每个非终结符语法单元，编一个独立的处理过程（或子程序）。语法分析从读入第一个单词开始，由非终结符<程序>（即开始符）出发，沿语法描述图箭头所指出的方向进行分析。当遇到非终结符时，则调用相应的处理过程，从语法描述图看，也就进入了一个语法单元，再沿当前所进入的语法单元所指箭头方向继续进行分析。当遇到描述图中是终结符时，则判断当前读入的单词是否与图中的终结符相匹配，若匹配，再读取下一个单词继续分析。遇到分支点时，将当前的单词与分支点上多个终结符逐个相比较，若都不匹配时可能是进入下一个非终结符语法单位或是出错。 下图举例说明了语法分析器对语法的识别过程：

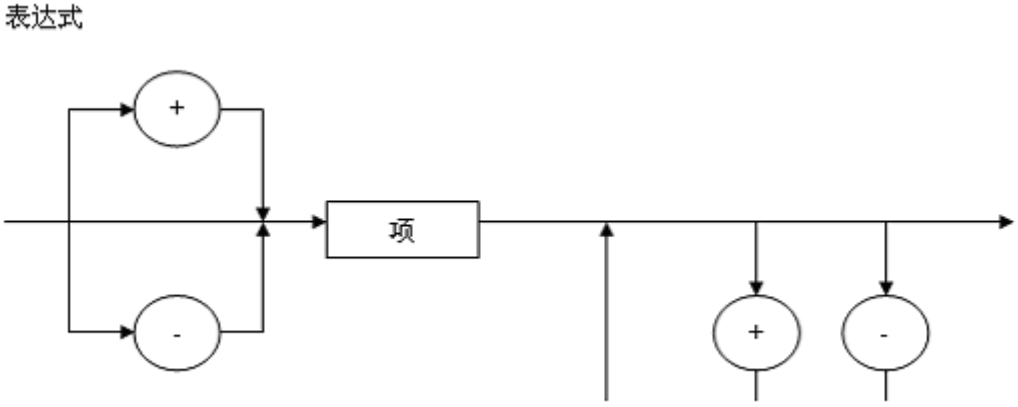

图 5-5 语法分析器对表达式的识别过程

5.3 语法分析树的生成

5.3.1 构造语法分析树

我们的 main 函数在 Main.java 文件内，生成语法分析树的代码在 TestRig.java 里，我们在 Main 里调用 TestRig 类的接口，来生成语法分析树，代码如下图：

```
ANTLRInputStream input = new ANTLRInputStream(is);
CLexer lexer = new CLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
CParser parser = new CParser(tokens);
ParseTree tree = parser.file(); // parse; start at prog <label id="code.tour.main.6"/>
String[] testArgs = new String[4];
testArgs[0] = "C"; testArgs[1] = "file"; testArgs[2] = "-gui"; testArgs[3] =  inputFile;
TestRig testRig = new TestRig(testArgs);
testRig.process();
```

图 5-6 主函数中构造语法分析树的代码

5.3.2 语法分析树的构造原理

在 TestRig 类中，我们使用如下代码生成语法分析树（详见 TestRig. java 文件）：

```
    public void process() throws Exception {
//      System.out.println("exec "+grammarName+"."+startRuleName);
        String lexerName = grammarName+"Lexer";
        ClassLoader cl = Thread.currentThread().getContextClassLoader();
        Class<? extends Lexer> lexerClass = null;
        try {
            lexerClass = cl.loadClass(lexerName).asSubclass(Lexer.class);
        }
        catch (java.lang.ClassNotFoundException cnfe) {
            // might be pure lexer grammar; no Lexer suffix then
            lexerName = grammarName;
            try {
                lexerClass = cl.loadClass(lexerName).asSubclass(Lexer.class);
            }
            catch (ClassNotFoundException cnfe2) {
                System.err.println("Can't load "+lexerName+" as lexer or parser");
                return;
            }
        }
```

```
        Constructor<? extends Lexer> lexerCtor = lexerClass.getConstructor(CharStream.class);
        Lexer lexer = lexerCtor.newInstance((CharStream)null);

        Class<? extends Parser> parserClass = null;
        Parser parser = null;
        if ( !startRuleName.equals(LEXER_START_RULE_NAME) ) {
            String parserName = grammarName+"Parser";
            parserClass = cl.loadClass(parserName).asSubclass(Parser.class);
            if ( parserClass==null ) {
                System.err.println("Can't load "+parserName);
            }
            Constructor<? extends Parser> parserCtor = parserClass.getConstructor(TokenStream.class);
            parser = parserCtor.newInstance((TokenStream)null);
        }

        if ( inputFiles.size()==0 ) {
            InputStream is = System.in;
            Reader r;
            if ( encoding!=null ) {
                r = new InputStreamReader(is, encoding);
            }
            else {
                r = new InputStreamReader(is);
            }

            process(lexer, parserClass, parser, is, r);
            return;
        }

        for (String inputFile : inputFiles) {
            InputStream is = System.in;
            if ( inputFile!=null ) {
                is = new FileInputStream(inputFile);
            }
            Reader r;
            if ( encoding!=null ) {
                r = new InputStreamReader(is, encoding);
            }
            else {
                r = new InputStreamReader(is);
            }

            if ( inputFiles.size()>1 ) {
                System.err.println(inputFile);
            }
            process(lexer, parserClass, parser, is, r);
        }
    }
```

图 5-7 TestRig 中构造语法分析树的部分代码

6 程序测试与结果

6.1 覆盖范围广的测试样例

我们编写了一个 c 语言的程序，作为测试程序，存放在 t1.c 文件中。这个测试数据囊括了大多数的 c 语言的语句，各种表达式、if-else 语句、while 语句、函数调用、嵌套语句等。t1.c 的部分内容如下图所示：

```
int main(int argc){
    a = 1;
    int b = a;
    float b = 1.0;
    c = b = (a + b) * -c - a == ++a;
    int i;
    if(i == 0){
        c = a +b;
        int j;
        if(j == -1)
            c = 2;
    }else if(i == 2){
        a = a*2;
    }else
    c = c * 3;
```

图 6-1 测试文件的部分内容

6.2 语法分析树

运行方法见本文档的第 3 章，通过分析 t1.c 得到的语法分析树如下图所示：

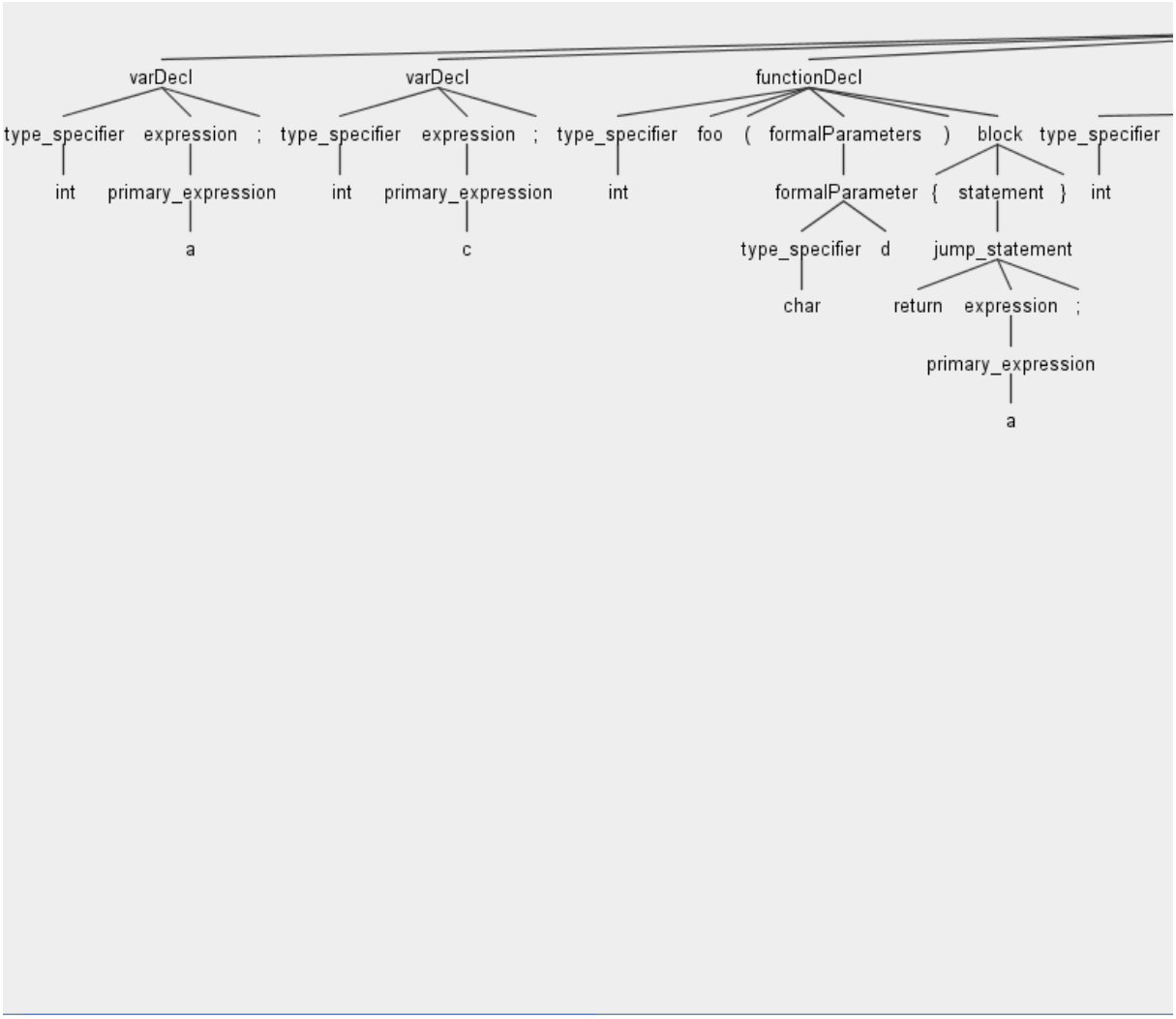

图 6-2 语法分析树的第 1 部分

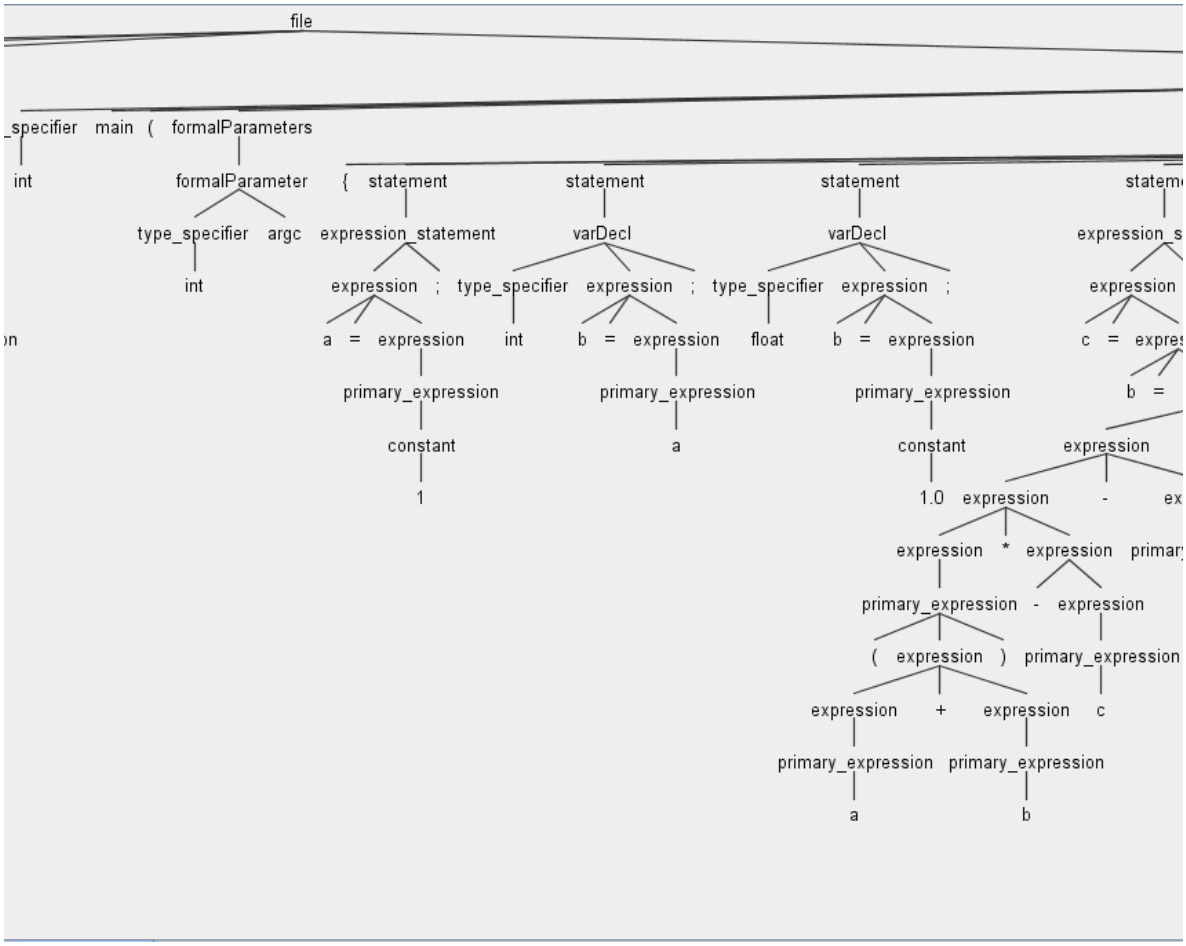

图 6-3 语法分析树的第 2 部分

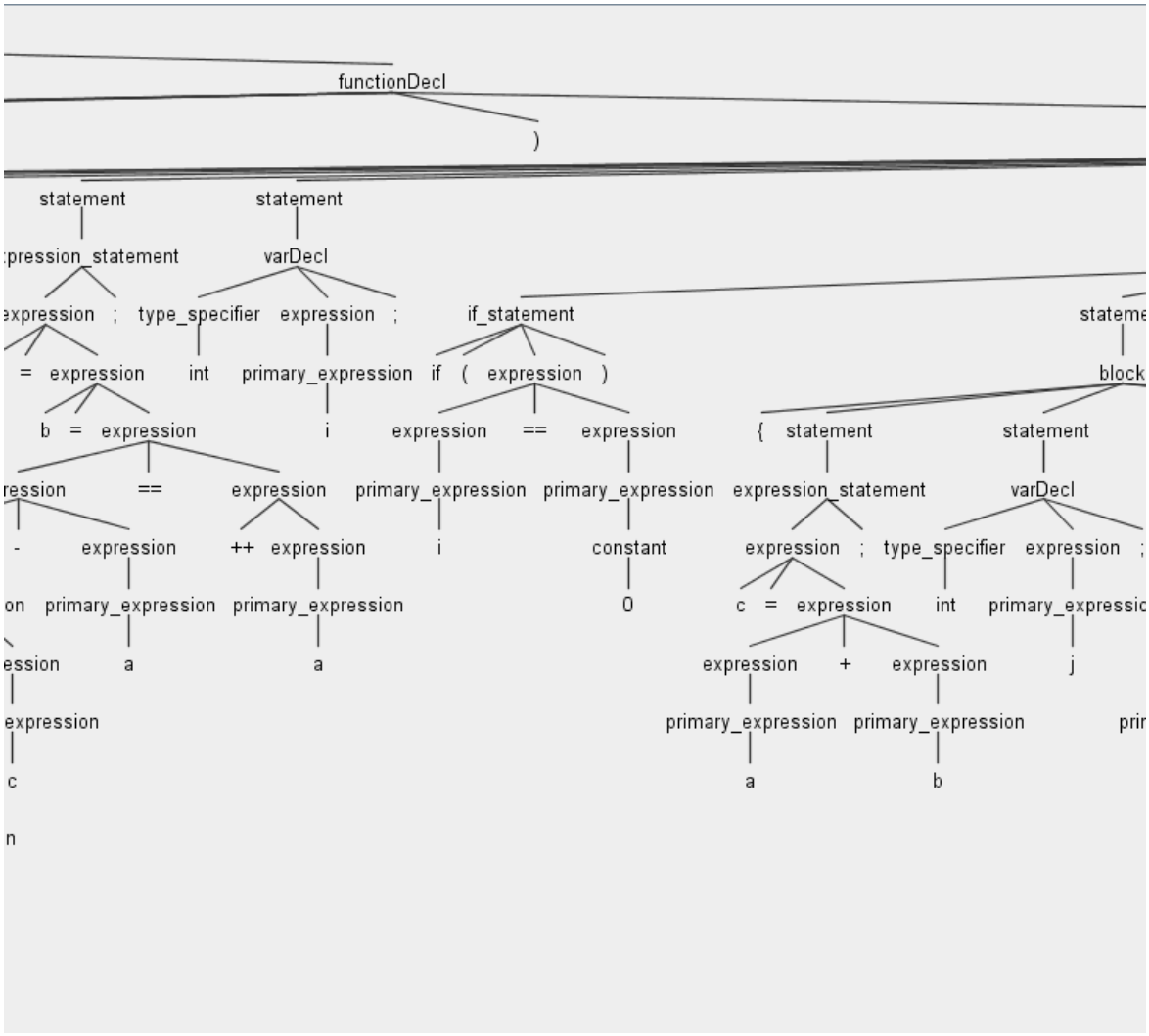

图 6-4 语法分析树的第 3 部分

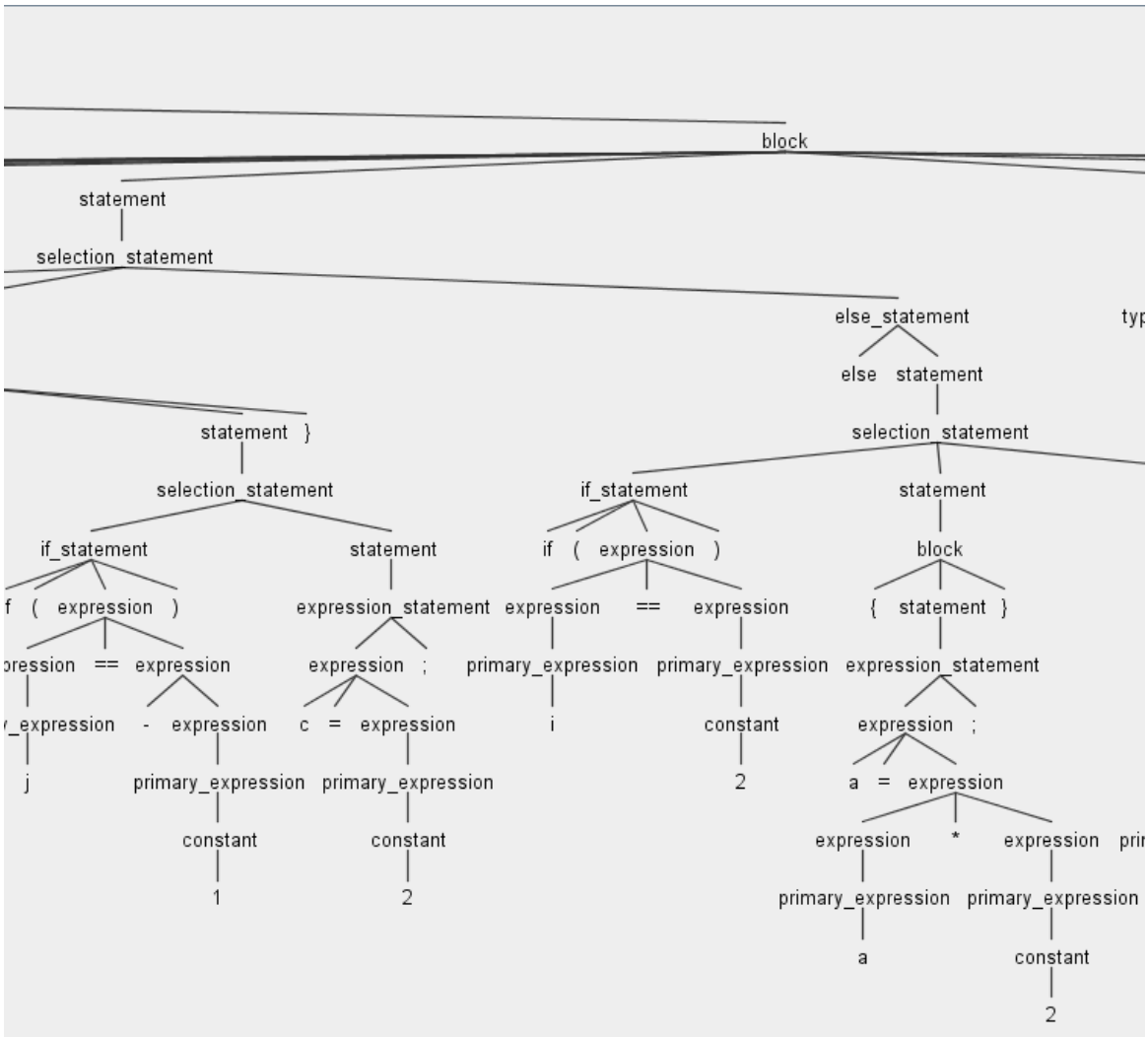

图 6-5 语法分析树的第 4 部分

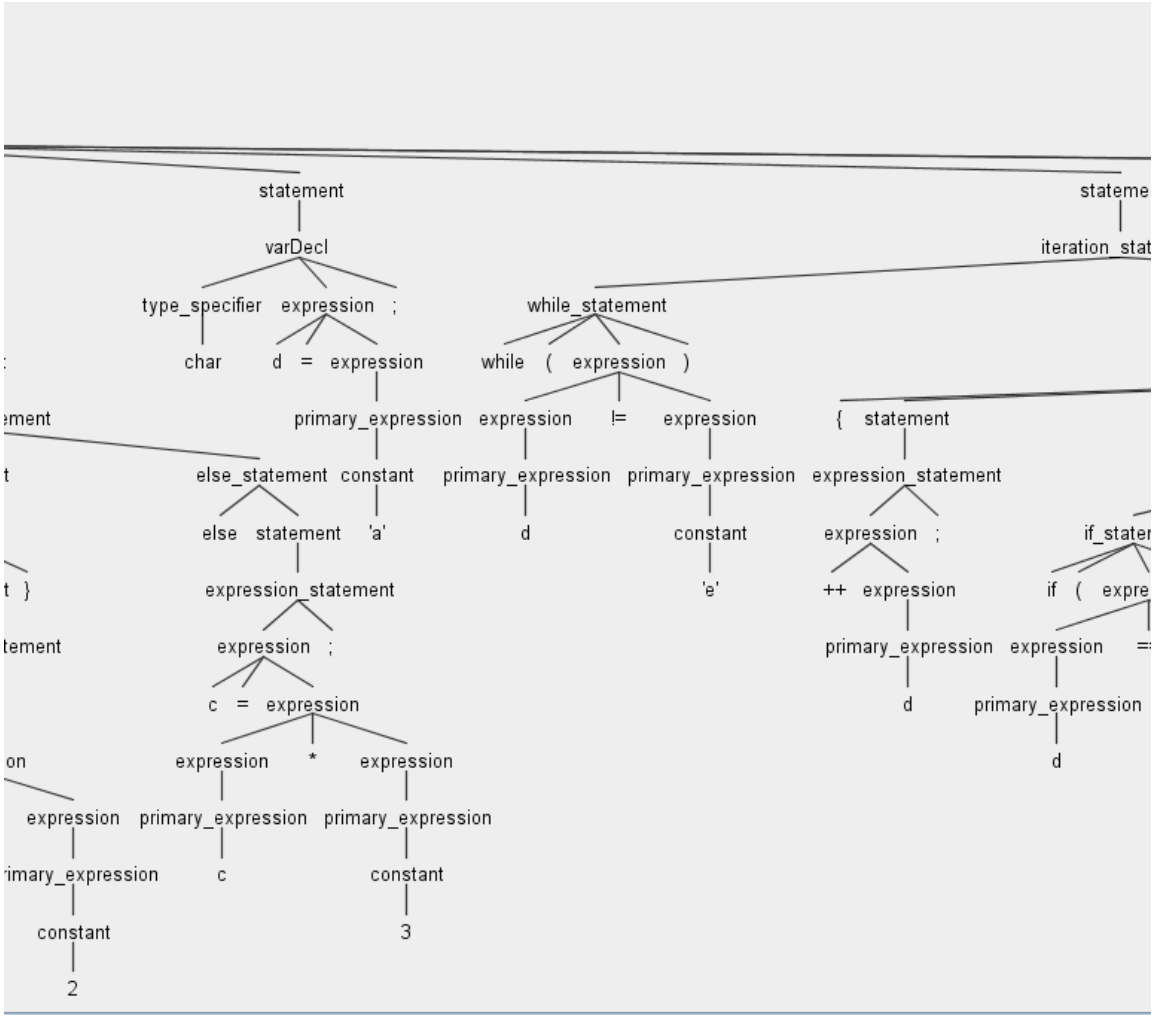

图 6-6 语法分析树的第 5 部分

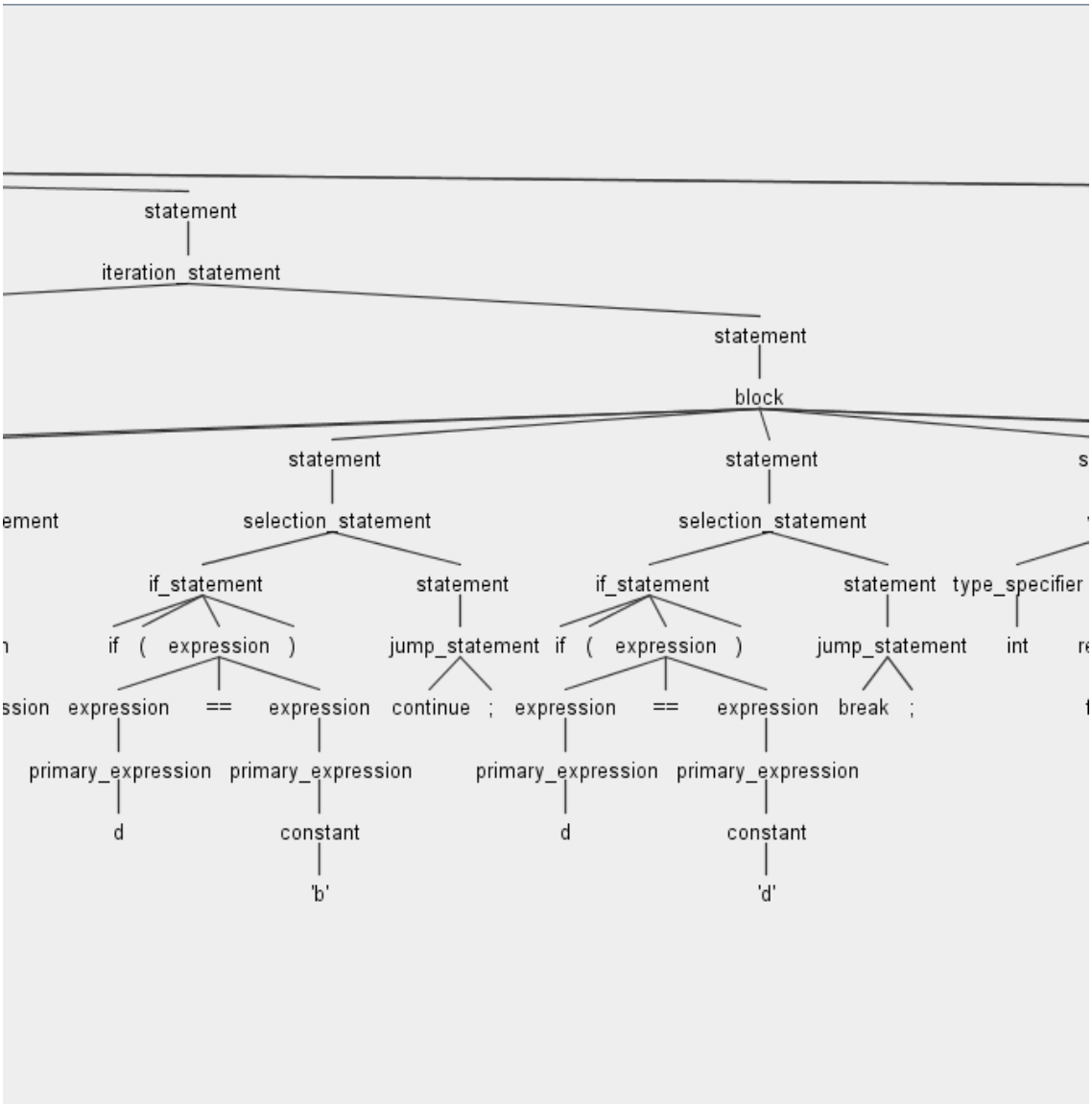

图 6-7 语法分析树的第 6 部分

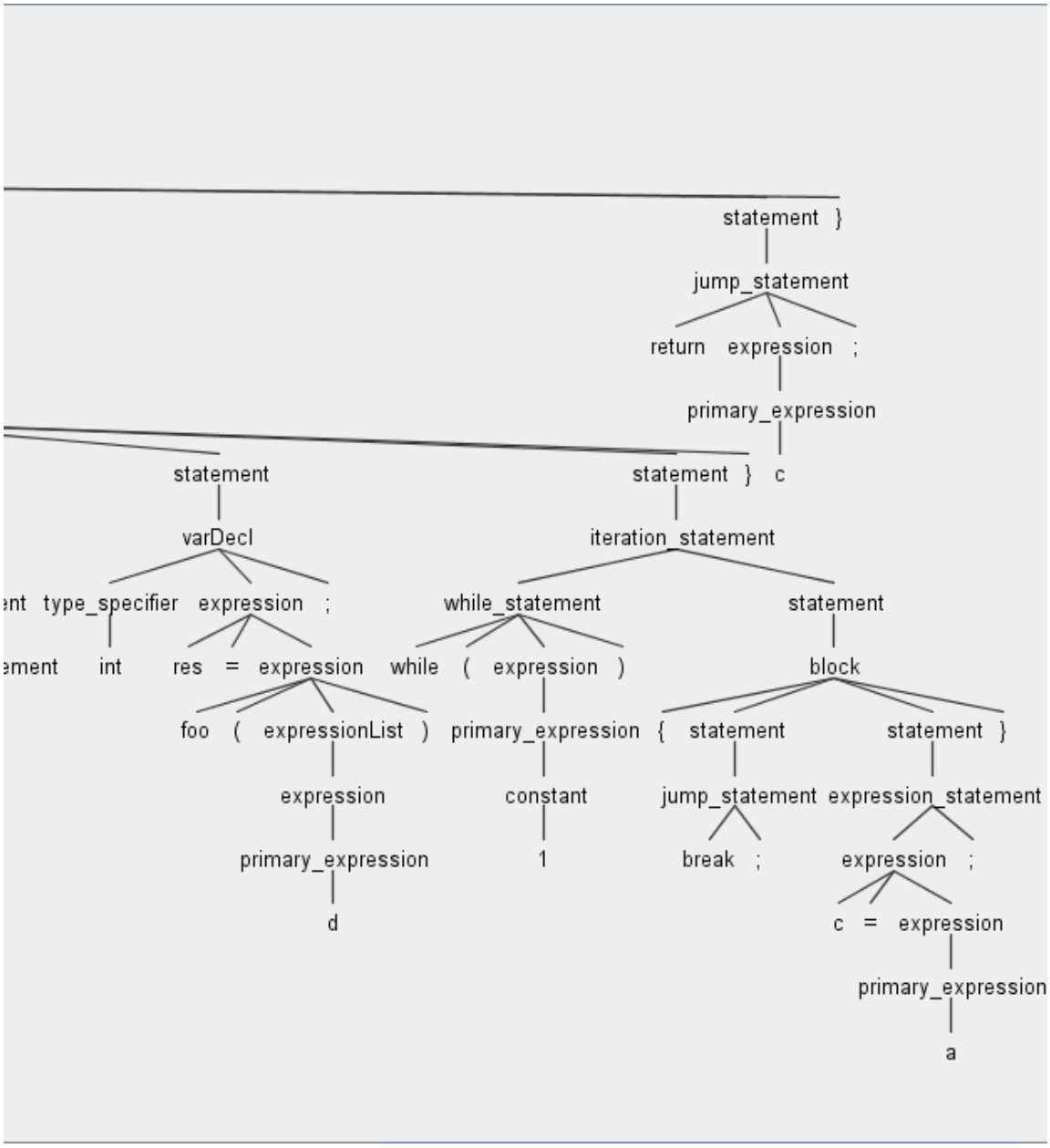

图 6-8 语法分析树的第 7 部分

6.3 语法错误检查与报错

把 t1.c 中的 a=1 后面分号去掉后就有了语法错误，然后保存，如图所示：

t1.c

```
4  int c;
5
6  int foo(char d){
7      return a;
8  }
9  //made by 张旭（Kim Johnson）
10 /*cn.edu.tju.cs.compiler*/
11
12 int main(int argc){
13      a = 1
14      int b = a;
15
```

图 6-9 对 t1.c 进行修改，使其有语法错误

然后在控制台程序中运行 java Main，输入源文件名 t1.c，会出现报错情况：

```
C:\Users\zk\Desktop\release> java Main
请输入源文件名:
t1.c
line 14:5 mismatched input 'int' expecting {'*', '-', '<', '!=', '<=', '%', '+',
 ';', '&&', '||', '>', '==', '/', '>='}
line 14:5 mismatched input 'int' expecting {'*', '-', '<', '!=', '<=', '%', '+',
 ';', '&&', '||', '>', '==', '/', '>='}
```

图 6-10 在控制台界面中会提示有相应的错误

在生成的语法树中也会显示相应的出错情况：

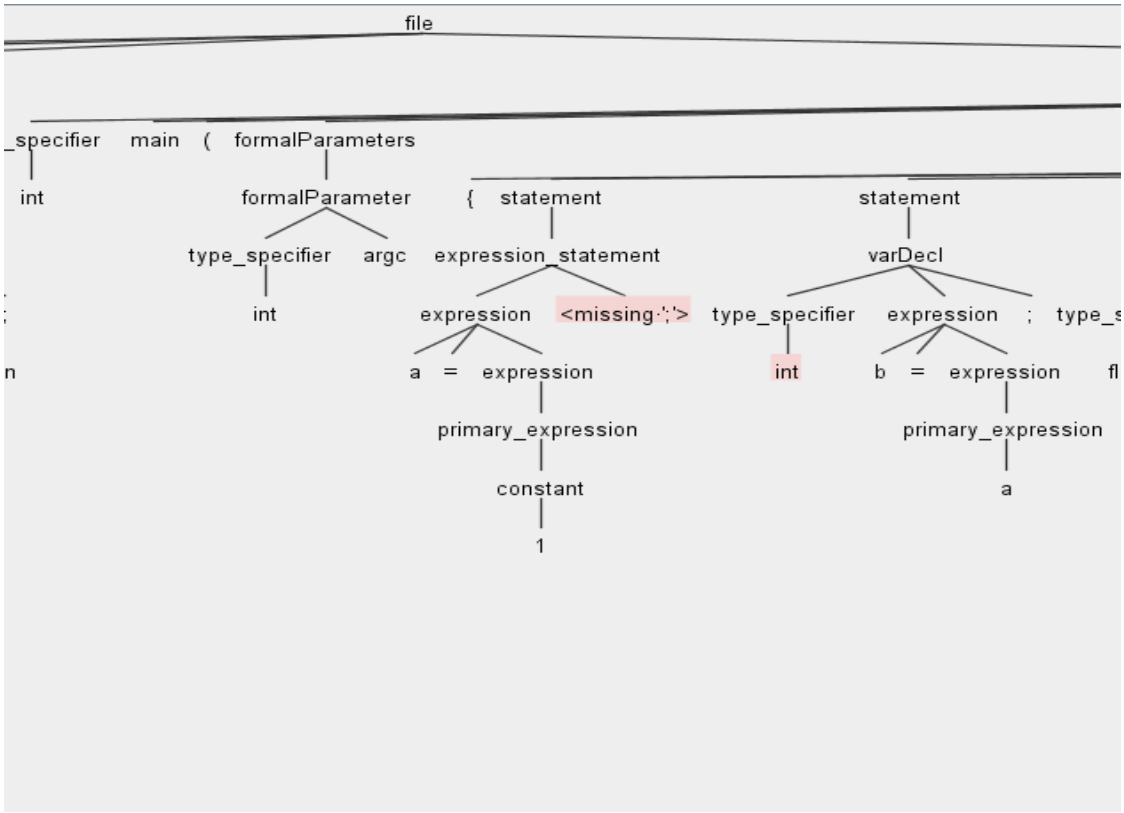

图 6-11 语法分析树也会提示相应的错误

7 实验总结

经过将近一个月的不断修改和完善,我们总算完成了编译技术课程的实验设计。此次实验可以说是我们自从进入大学以来所接手的工作量最大,设计最复杂,涉及数据结构和算法知识最多的一次实验。可以说整个实验不仅是对我们个人计算机能力的考验,更是小组团队成员是否具有配合意识和团队精神的考验。通过这次的实验，使我明白了其实设计想象中这样一个优秀的编译器不是这么简单的，它需要很多优秀程序员的智慧和汗水。

我们在这次设计后深深的感受到“一分耕耘，一分收获”。设计的过程是辛苦的，但当设计完成时会有一种满足感，而且自己也收获不少。通过这次设计，我们学到了很多东西，把学到的东西都派上了用场。在设计中,我们感受到了这几年来大学学习和生活的价值。作为一名计算机专业的学生，我的目标不仅仅是程序员,而这一次设计就是我向自己的目标迈出的第一步。

刚开始我们从编译器组成的各个模块入手，我们发现每个模块之间的联系都异常的紧密，从词法分析，语法分析等再到最终的解释程序等等。整个编译器就是一个工程而模块设计的好坏直接关系到了这个工程的质量。我们小组在这次实验中分工明确，虽然每个人的能力不一样，但我们努力做到互相沟通互相帮助尽量不让任何一个成员觉得自己没干什么活，学不到什么实践知识的想法。只有这样我们才能在将来踏上社会后能有足够的团队意识和学习动力。从刚开始的输入源代码，到后面的反复测试，我们都花费了大量的精力。到最后我们渐渐意识到其实整个编译器的C++代码并不是我们要重点学习的，而我们重点要掌握的则是编译原理的分析思想和将这个思想转化为编译器的软件工程上的思想。这些无穷长的代码背后始终贯彻着这些思想,其实任何一个软件又何尝不是如此呢？所有软件的代码都展示出软件工程师和架构师们的思想,而程序员只是实现他们思想的人。所以作为一名计算机专业的学生必须要牢记:任何一门语言都会过时,而永不过时的是数据结构和优秀的算法。

此时我想感谢老师给我们布置的实验，起初我们很不能理解我们从来没写过大程序的学生却要求写出编译器，但现在我明白了，如果你只是写一个词法分析器那你永远无法真正从实践上理解编译的整个过程的实现更不可能有什么软件

工程的思想给你去体会了。

其实我们这次的实验还是有很多不足之处的，程序健壮性还很不够,没有比较好的界面。与真正的软件比起来实在相差太远，但毕竟这是实验，实验里犯的错误多了不要紧，把该犯的犯完以后走上工作岗位后就不会再犯了。作为一名大三的学生我们要走的软件设计之路还相当的远,但我相信用计算机底层思想武装起来的IT工作者永远要比只懂表层技术的工作者走在计算机行业的最先列。