|  |  |
| --- | --- |
|  |  |
| **语法分析器** |  |
|  |  |
|  | **2020.05.22**  **编译原理** |
|  |  |

目录

[1 实验目的以及要求 3](#_Toc41086908)

[1.1 理解编译器的工作机制，并编译器的工作原理 3](#_Toc41086909)

[1.2 掌握语法分析器生成工具bison的用法 3](#_Toc41086910)

[1.3 测试输入文件的要求 3](#_Toc41086911)

[1.4 输出文件的要求 3](#_Toc41086912)

[2 实验框架 4](#_Toc41086913)

[2.1 核心程序层次结构 4](#_Toc41086914)

[2.2 实验步骤 4](#_Toc41086915)

[3 设计思路 5](#_Toc41086916)

[3.1 语法分析器 5](#_Toc41086917)

[3.1.1 语法规则 5](#_Toc41086918)

[3.2 词法分析器 7](#_Toc41086919)

[3.3 构建语法树 7](#_Toc41086920)

[3.3.1 生成词法和语法分析的综合文档 7](#_Toc41086921)

[3.3.2 使用栈进行规约 7](#_Toc41086922)

[3.3.3 用dot工具绘图 8](#_Toc41086923)

[为什么不一边规约一边直接生成树 8](#_Toc41086924)

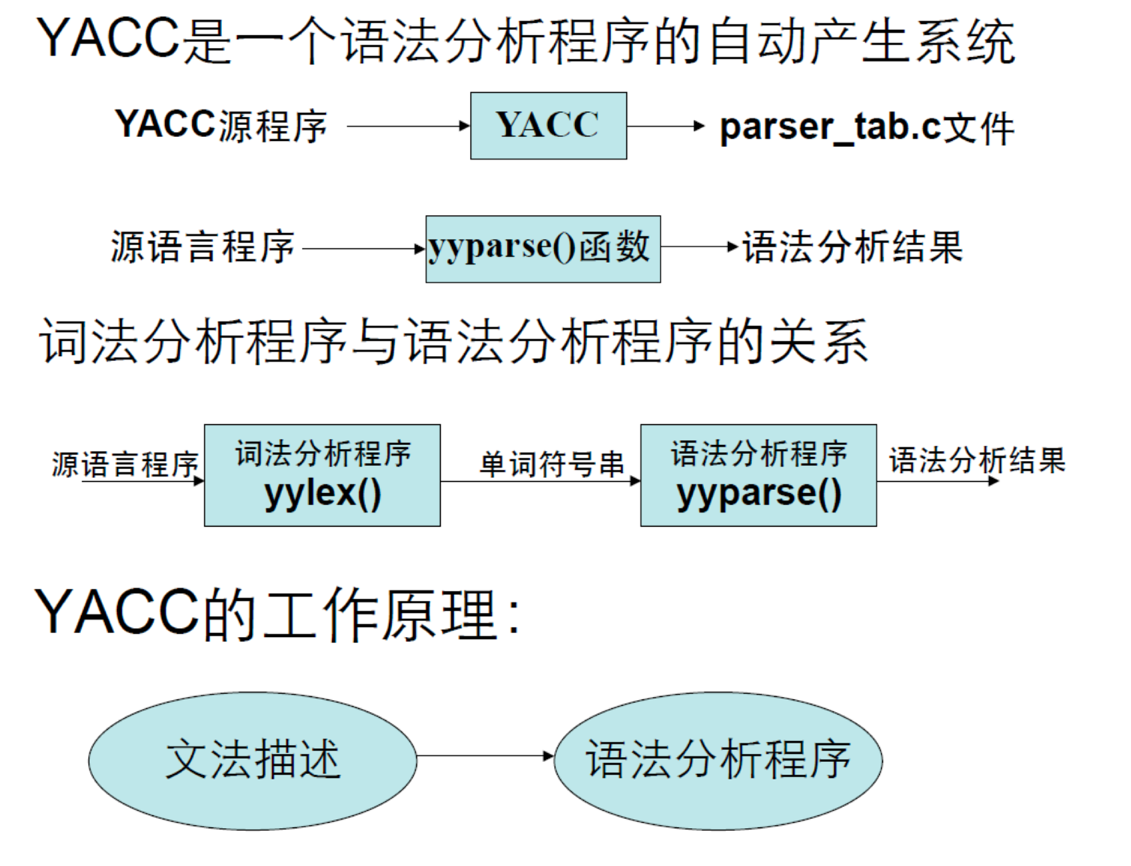
[4 遇到的问题和解决方法 9](#_Toc41086925)

# 实验目的以及要求

## 理解编译器的工作机制，并编译器的工作原理

## 掌握语法分析器生成工具bison的用法

YACC使用LALR(1)文法表示上下文无关文法



## 测试输入文件的要求

* 支持的所有语法成分（含扩展成分）
* 嵌套过程（三层）
* 并列过程

## 输出文件的要求

* 按归约顺序用到的语法规则序列。
* 语法树（或能表示语法单位的层次结构关系的其他形式）。

# 实验框架

## 核心程序层次结构

**可执行文件res对测试程序进行语法分析，之后draw.cpp再对分析结果进行处理，得到处理结果后，使用graphviz中的dot工具画出语法树。**

## 实验步骤

具体命令参考readme.md文件

* 用flex工具生成lex.yy.c文件
* 用bison工具生成SyntaxPl0.tab.c和SyntaxPl0.tab.h文件
* 使用g++将 SyntaxPl0.tab.c和lex.yy.c文件联合编译生成可执行文件res
* 使用可执行文件res对测试程序test\_sum.pl0语法分析
* 使用draw.cpp文件处理分析结果，构建语法树
* 使用dot工具画出语法树

# 设计思路

## 语法分析器

### 语法规则

语法规则对应于EBNF范式

例如：

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **左为EBNF范式**  **右为对应的语法规则**  <程序> ::= <分程序>.  <分程序> ::= [<常量说明部分>][<变量说明部分>] [<过程说明部分>]<语句>  语法规则就是对EBNF范式进行了翻译，并且在规约时，将所做的规约动作的规则输出到两个文件中。   * 第一个文件存放按照规约顺序用到的语法规则序列 * 第二个文件将用于构造语法树，在下文中具体解释 |  |  |

下面为扩展文法的EBNF范式和语法规则的对应

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | 数组  <变量说明部分> ::= (VAR<标识符>[(ARRAY \[<无符号整数>..<无符号整数>|<表达式>{,<无符号整数>..<无符号整数>|<表达式>}\]OF<类别1>) | (:<类别2>)]{,<标识符>[(ARRAY \[<无符号整数>..<无符号整数>|<表达式>{,<无符号整数>..<无符号整数>|<表达式>}\]OF<类别1>) | (:<类别2>)};) |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | For和Repeat  <FOR循环语句>::= FOR<变量>:=<表达  式>TO<表达式>DO<语句>  <REPEAT循环语句>::= REPEAT<语句>UNTIL<条件> |  |  |

更多的实现细节，可以在原代码和EBNF的word文档中看到。

## 词法分析器

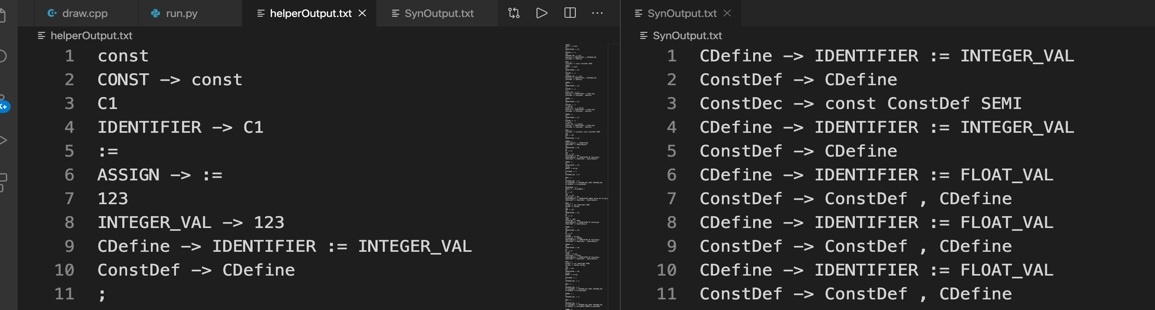
与上次实验的词法分析器相比，做出了以下改进

* 使用strcasecmp()函数进行不区分大小写的匹配
* 每当扫描到一个词将它规约为token的时候返回对应的词法匹配结果
* 每当扫描到单词以及决定匹配方式分别将扫描的单词和匹配方式输出到一个文件中，和语法分析器中第二个输出的文件相同，下面建立语法树会用到

## 构建语法树

### 生成词法和语法分析的综合文档

词法分析器中扫描到词时立刻输出当前的yytext，接着输出所选择的词法规则，如图a中所示，扫描测试程序test\_sum.pl0时，扫描到const,将const输出，词法选择将const匹配为CONST，再输出“CONST->const”。



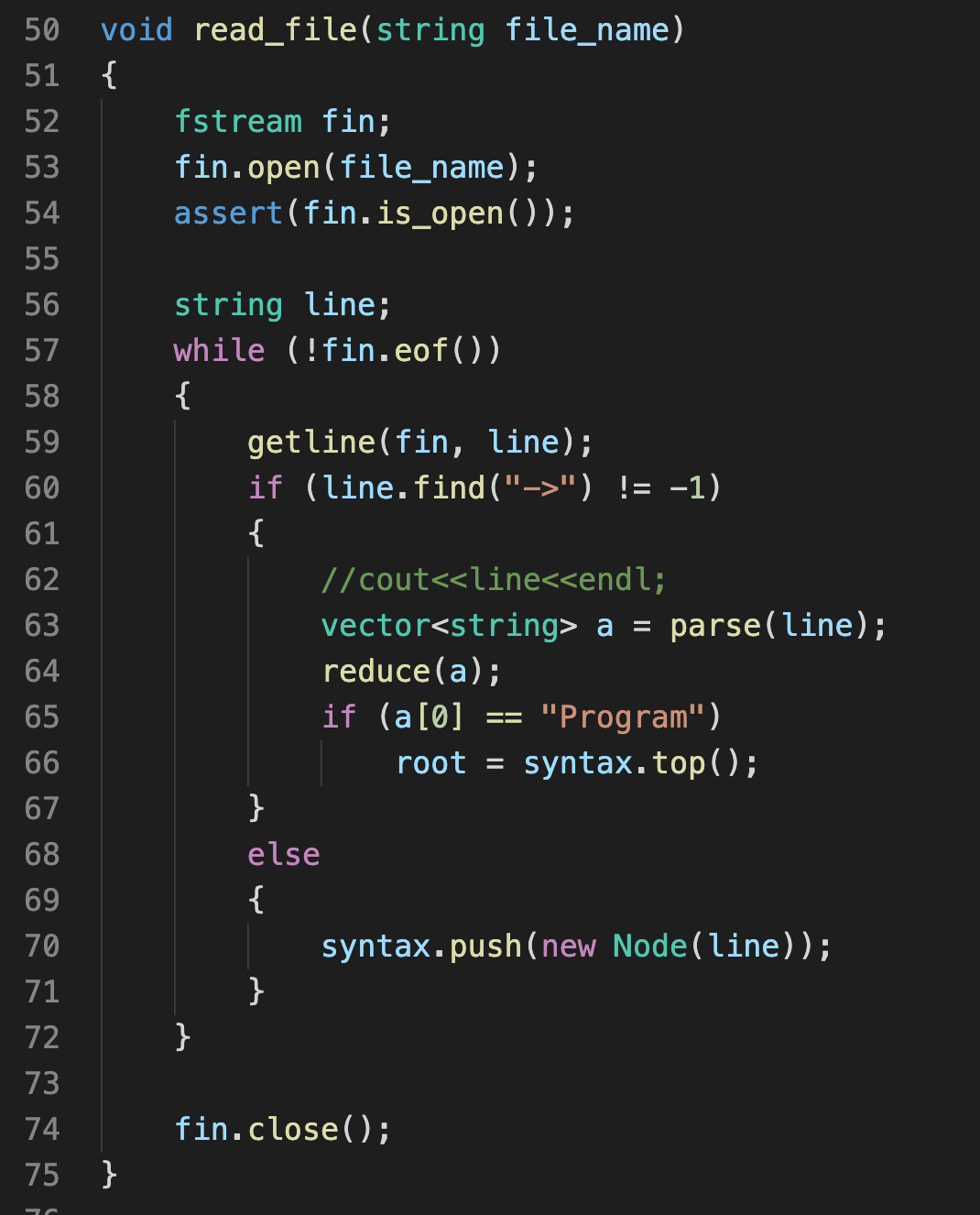
图表 ：左边为帮助文档，右边为语法规则的规约序列

语法分析器中，决定使用某个语法规则进行规约以后，将该规则输出到2个文件中，一个是图a中左边的帮助文档，一个是图a中右边的语法规则的规约序列文档。例如首先使用“CDefine -> INDETIFIER := INTEGER\_VAL”进行规约，对应于左边第9行，右边第1行。

### 使用栈进行规约

遇到yytext的内容就压入栈中，遇到词法规则或者语法规则就进行规约，将栈顶的对应规则的“->”后数量的内容pop出栈，将“->”左侧的内容入栈。该内容在draw.cpp中实现，对应的核心函数如图b所示。

其中reduce函数将pop掉的部分作为子结点插入到规约后生成的结果当中。对“Program”项目特殊处理，如果是这个项目说明是根节点root。



图表

### 用dot工具绘图

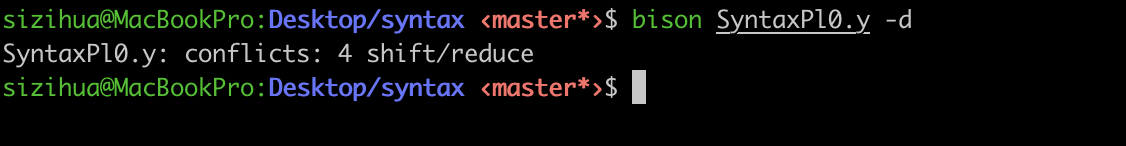
* 使用DFS（深度优先搜索）将语法树的信息输出到ASTvis.gv文件中
* 在命令行使用dot绘制语法树

|  |  |  |
| --- | --- | --- |
|  |  |  |
| 为什么不一边规约一边直接生成树 关键问题在于保证语法树中同一层结点的顺序。  如果在确定规约的时候就生成结点，那么非终结符不好判断应该将什么作为其子结点。  如果在词法分析器识别token的时候生成结点，考虑到flex会进行超前搜索（flex的—Batch），那么某些终结符和非终结符的顺序会出现混乱。 | |  |

# 遇到的问题和解决方法

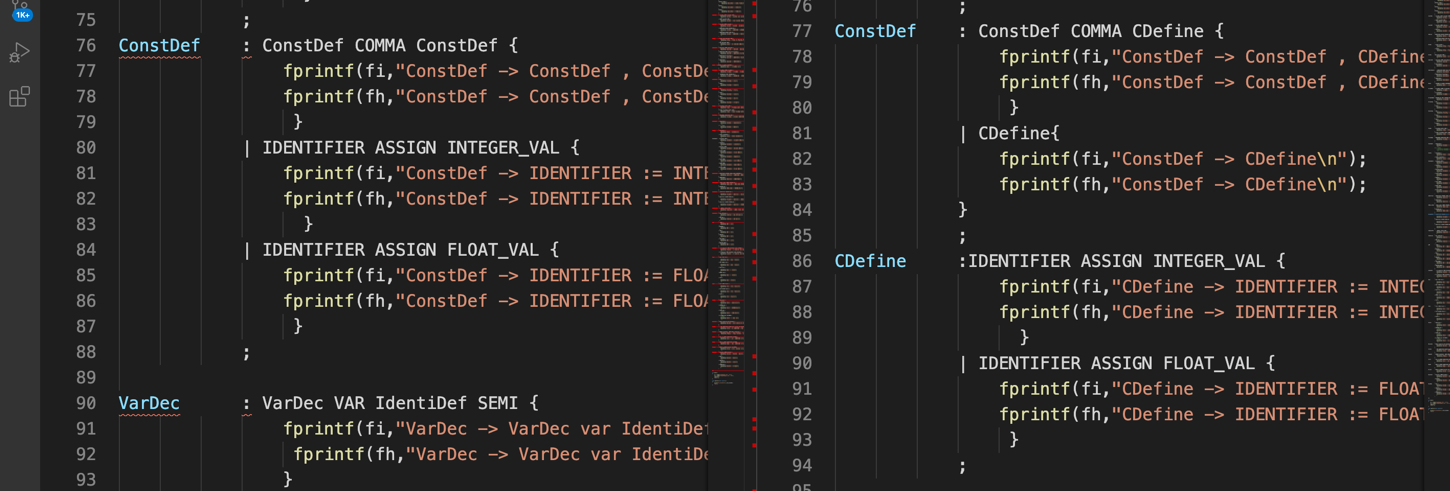
**问题1**

* 使用bison时出现shift/reduce错误，如图c所示



图表

* 图d中，左侧为其中一处shif/reduce错误，右侧为解决方法
  + 出现该错误的原因是语法规则中同时出现左递归和右递归，我这样写原以为可以简化语法规则，没想到会出现错误
  + 使用右边的方法改为左递归以后，错误就消失了

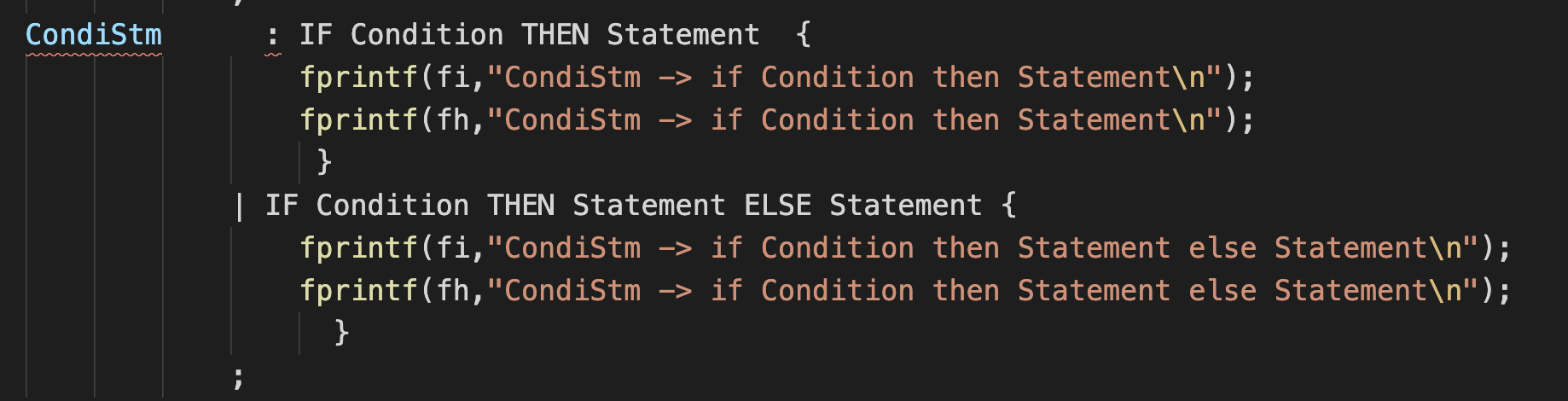


图表

**问题2**

**二义文法**

* IF Than Else 和IF Than 出现移进规约冲突
  + 产生冲突的代码如图e所示



图表

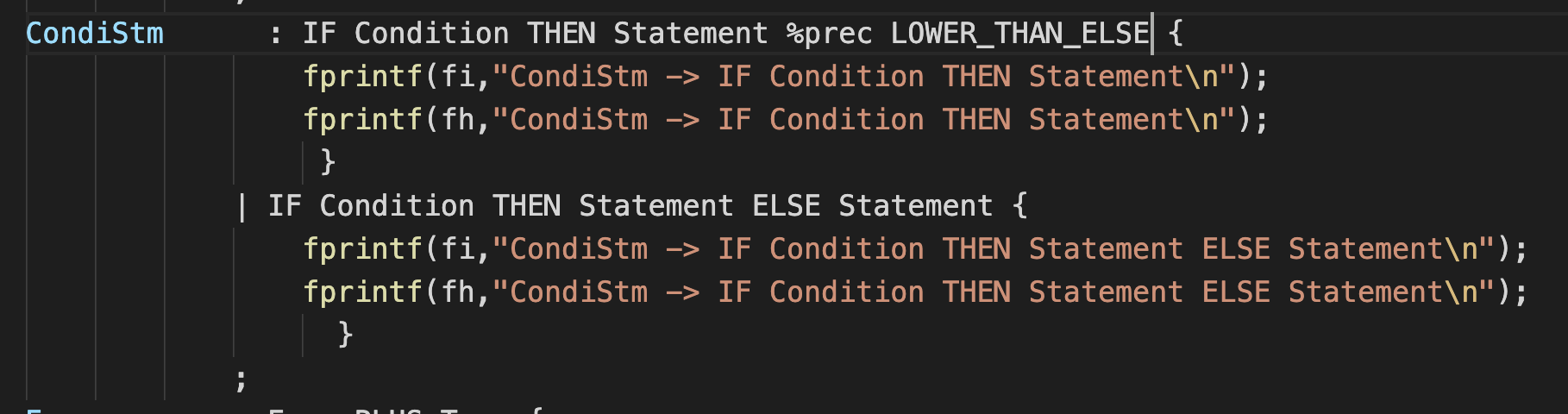
解决方式

* %nonassoc声明LOWER\_THAN\_ELSE和ELSE,如图f所示



图表

* 使用%prec标记说明IF THAN ELSE的优先级和LOWER\_THAN\_ELSE相同，该符号的声明在ELSE的上方，所以其优先级低于ELSE。由此一来通过定义优先级解决了二义文法的冲突。



图表