中国矿业大学计算机学院

系统软件开发实践报告

**课程名称** 系统软件开发实践

**报告时间** 2020年4月22日

**学生姓名** 陆玺文

**学 号** 03170908

**专 业** 计算机科学与技术

**任课教师** 张博

成绩考核

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 编号 | 课程教学目标 | | | 占比 | | 得分 |
| 1 | **目标1：**针对编译器中词法分析器软件要求，能够分析系统需求，并采用FLEX脚本语言描述单词结构。 | | | 15% | |  |
| 2 | **目标2：**针对编译器中语法分析器软件要求，能够分析系统需求，并采用Bison脚本语言描述语法结构。 | | | 15% | |  |
| 3 | **目标3：**针对计算器需求描述，采用Flex/Bison设计实现高级解释器，进行系统设计，形成结构化设计方案。 | | | 30% | |  |
| 4 | **目标4：**针对编译器软件前端与后端的需求描述，采用软件工程进行系统分析、设计和实现，形成工程方案。 | | | 30% | |  |
| 5 | **目标5：**培养独立解决问题的能力,理解并遵守计算机职业道德和规范，具有良好的法律意识、社会公德和社会责任感。 | | | 10% | |  |
| 总成绩 | | | | | |  |
| 指导教师 | |  | 评阅日期 | |  | |

目 录

[1、 实验三 Bison实验一 1](#_Toc38961638)

[1.1 实验目的 1](#_Toc38961639)

[1.2 实验内容 1](#_Toc38961640)

[1.3 实验要求 1](#_Toc38961641)

[1.4 移进规约冲突 1](#_Toc38961642)

[1.4.1 移进规约冲突原因及解决 1](#_Toc38961643)

[1.4.2 本次实验中出现的移进规约冲突 2](#_Toc38961644)

[1.5 实验步骤 2](#_Toc38961645)

[1.5.1 CentOS环境下 2](#_Toc38961646)

[1.5.2 Windows环境下 4](#_Toc38961647)

[1.6 符号表和语法分析树相关代码简述 5](#_Toc38961648)

[1.6.1 符号表分析 5](#_Toc38961649)

[1.6.2 语法分析树代码简述 6](#_Toc38961650)

[1.7 实验总结 9](#_Toc38961651)

[1.7.1 遇到的难题 9](#_Toc38961652)

[1.7.2 实验收获 9](#_Toc38961653)

# 实验四 Bison实验2

## 实验目的

阅读C语言文法的相关参考资料，利用Bison实现一个C语言语法分析器。

## 实验内容

利用语法分析器生成工具 Bison 编写一个C语言的语法分析程序，与词法分析器结合，能够根据语言的上下文无关文法，识别输入的单词序列是否文法的句子。

## 实验要求

1. 阅读Flex源文件input.lex、Bison源文件cgrammar-new.y，并参考《实验四 借助FlexBison进行语法分析.pdf》上机调试。
2. 以给定的测试文件test.c作为输入，输出运行结果到输出文件out.txt中。

## 移进规约冲突

### 移进规约冲突原因及解决

分析表是LR分析其的核心，它跟具体有关，包括动作表和状态转换表两部分。动作表中的元素action[Si , ai]表示栈顶当前状态为Si，和当前输入符号为ai时完成的分析动作。其中，“移进”分析动作表示句柄尚未在分析栈顶行程，正期继续移进符号以形成句柄，“规约”表明当前分析栈的栈顶已形成当前句型的句柄β，要立即进行规约。

下面通过具体例子介绍移进规约冲突。

文法G[S]:



识别文法G[S]的部分可归前缀如图 1‑1所示。在项目集S2中可以看到，既有移进项目(expr->expr·\* expr)，又有规约项目(expr->-expr·)，这说明当分析到状态集2时，如果识别到了expr，下一个符号如果是\*，或者将“\*”移进符号栈，或者按文法的产生式(expr->-expr)进行归约，于是出现了“移进-规约”冲突。



图 1‑1 识别文法G[S]部分可归前缀

### 本次实验中出现的移进规约冲突

本次实验中含有如下文法G[S1]：



识别文法G[S1]特定项目集S如图 1‑2所示。可以看到在项目集S中，当下一个输入符号是ELSE时，或者将“ELSE”移进符号栈，或者按文法的产生式(Stmt -> IF ( Exp ) Stmt ELSE Stmt)进行归约。于是出现了移进归约冲突。



图 1‑2 识别文法G[S1]特定项目集S

## 实验步骤

### CentOS环境下

#### 编译input.lex与cgrammar-new.y

在CenOS终端中，执行命令：

# flex input.lex

# bison -d cgrammar-new.y

cgrammar-new.y : conflicts: 1 shift/reduce

#### 分析移进归约冲突

使用bison的-v选项生成状态机描述文件cgrammar-new.output查看具体问题。

# bison -v cgrammar-new.y

使用vim编辑器查看.output文件时，可以定位到具体的冲突发生位置。

# vim cgrammar-new.output

State 341 conflicts: 1 shift/reduce

#### 修改冲突

Yacc中解决二义性文法的方法通常是指定优先级，%nonassoc意味着没有依赖关系，经常在连接词中和%prec一起使用，用于指定一个规则的优先级。

在yacc的头文件中加入

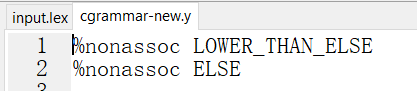
%nonassoc LOWER\_THAN\_ELSE

%nonassoc ELSE

在355行加入

%prec LOWER\_THAN\_ELSE

修改效果如图 1‑3所示



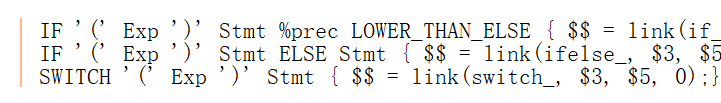


图 1‑3 移进规约冲突修改

上面的修改方案，使得“LOWER\_THAN\_ELSE”的优先级小于“ElSE”，同时语法第一句的优先级被指定为了“LOWER\_THAN\_ELSE”，这样当冲突发生时，编译器将先移进，后规约。

一个关于%prec的解释如下：“It declares that that construct has the same precedence as the ‘.’ operator, which will have been specified earlier.”

#### 编译

使用gcc进行编译。

# gcc -o c-grammar lex.yy.c cgrammar-new.tab.c main.c parser.c

会提示io.h文件无法找到，可以使用“find”命令，找到io.h的位置，移动到实验目录下即可。同时将input.lex中的原语句#include<io.h>修改为#include”io.h”。

# find /usr/include/ -name io.h

/usr/include/sys/io.h

# cp /usr/include/sys/io.h [实验文件目录]

再次运行gcc编译语句，提示ULONG\_MAX未定义，此时可以修改parser.c文件，在其中的头部显示的定义ULONG\_MAX，如图 1‑4所示。

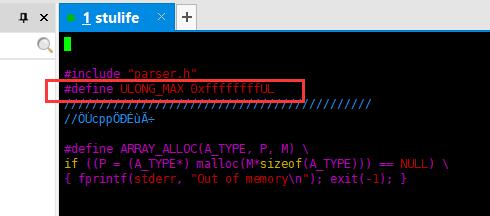


图 1‑4 ULONG\_MAX定义解决

修改后重新编译，提示有关“yyinput”与“int input()”的错误问题。此时可以修改lex.yy.c中两处条件编译语句。

将

#ifdef \_\_cplusplus

Static int yyinput()

#else

Static int input()

修改为：static int yyinput()。

将

#ifdef \_\_cplusplus

Return yyinput();

#else

Return input();

#endif

修改为：return yyinput()。

再次执行gcc编译语句，即可成功生成可执行文件c-grammar。

# gcc -o c-grammar lex.yy.c cgrammar-new.tab.c main.c parser.c

### Windows环境下

在Windows下的整体步骤与LINUX下相近，需要注意的是执行过程中会提示yylineno未定义的问题，可以通过在lex.yy.c中显示定义“int yylineno=0;”加以解决。最终运行输出结果如图 1‑5所示，

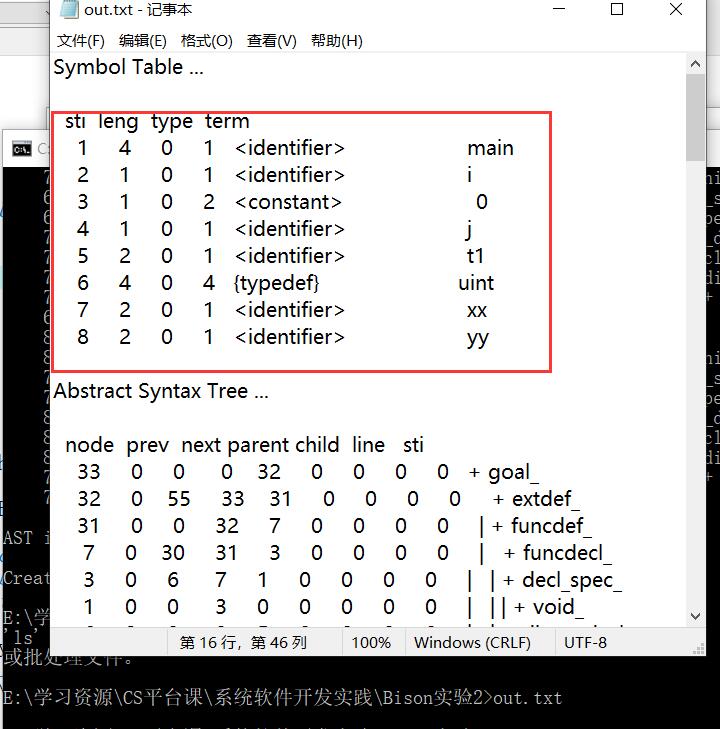


图 1‑5 Windows下运行示意图

## 符号表和语法分析树相关代码简述

### 符号表分析

符号表生成建立在测试代码基础上，详细代码如代码 1‑1所示。

代码 1‑1 test.c

|  |  |
| --- | --- |
|  | */\*\*/* |
|  | **void** main**()** |
|  | **{** |
|  | **int** i **=** 0**;** |
|  | **int** j **=** 0**;** |
|  | **}** |
|  |  |
|  | **void** t1**()** |
|  | **{** |
|  | **int** i **=** 0**;** |
|  | **}** |
|  | */\*\*/* |
|  |  |
|  | **typedef** **unsigned** **int** uint**;** |
|  |  |
|  | uint xx**;** |
|  | uint yy**;** |

表 1‑1 符号表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **sti** | **leng** | **type** | **term** |  |  |
| **1** | **4** | **0** | **1** | **<identifier>** | **main** |
| **2** | **1** | **0** | **1** | **<identifier>** | **i** |
| **3** | **1** | **0** | **2** | **<constant>** | **0** |
| **4** | **1** | **0** | **1** | **<identifier>** | **j** |
| **5** | **2** | **0** | **1** | **<identifier>** | **t1** |
| **6** | **4** | **0** | **4** | **{typedef}** | **uint** |
| **7** | **2** | **0** | **1** | **<identifier>** | **xx** |
| **8** | **2** | **0** | **1** | **<identifier>** | **yy** |

从符号表中可以看出，成功分析出了main、i、j、xx、yy、t1等标识符，分析出了uint类型符，常量0。同时很好的识别出了每一个标识符的长度

### 语法分析树代码简述

Out.txt中输出的部分语法分析树如表 1‑2所示。这是一种子女兄弟链的二叉树输出方式，将其复原为树型，可以得到如图 1‑6所示的图形。

表 1‑2 部分语法分析树输出

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| node | prev | next | parent | child | line | sti |  |  |  |  |  |  |  |
| 3 | 0 | 6 | 7 | 1 | 4 | 0 | 0 | 0 | | | | | + | decl\_spec\_ |  |
| 1 | 0 | 0 | 3 | 0 | 4 | 0 | 0 | 0 | | | | | | | + | void\_ |
| 6 | 3 | 0 | 0 | 5 | 5 | 0 | 0 | 0 | | | | | + | direct\_decl\_ |  |
| 5 | 0 | 0 | 6 | 4 | 4 | 0 | 0 | 0 | | | | | + | funcdecl\_ |  |

下面开始具体分析它的输出代码。

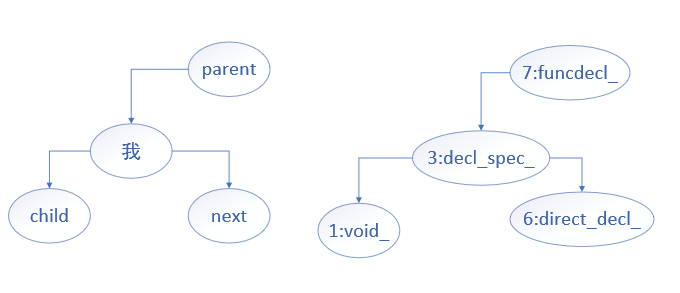


图 1‑6 二叉树型

主函数main()位于main.c中，打开后可以看到如代码 1‑2所示，其中“printast()”函数执行输出。

代码 1‑2 main.c

|  |  |
| --- | --- |
|  | **void** main **(int** na**,** **char** **\***arg**[])** |
|  | **{** |
|  | **char** **\***filename **=** **"test.c";** |
|  | **char** **\*** outfilename **=** **"out.txt";** |
|  | *//指向文件* |
|  | **if(!(**yyin **=** fopen**(**filename**,"r")))** **{** |
|  | printf**("the file not exist\n");** |
|  | exit**(**0**);** |
|  | **}** |
|  | *//初始化* |
|  | init\_parser**(**100**,** 1000**);** |
|  | **if(**yyparse**())**exit**(**1**);** |
|  | *//打开输出文件* |
|  | init\_out\_file**(**outfilename**);** |
|  | print\_symtab **(**term\_symb**);** *// Print the symbol table* |
|  | *//遍历ast树* |
|  | printast**();** *// Print the AST* |
|  | *//关闭输出文件* |
|  | term\_out\_file**(**outfilename**);** |
|  | **return** **;** |
|  | **}** |

代码 1‑3 Node(parser.h)

|  |  |
| --- | --- |
|  | **typedef** **struct** Node |
|  | **{** |
|  | **unsigned** **short** id |
|  | **unsigned** **short** prod**;** |
|  | **int** node\_index**;** *//node 在node数组中的索引* |
|  | **int** sti**;** |
|  | **int** prev**;** |
|  | **int** next**;** |
|  | **int** line**;** |
|  | **int** child**;** |
|  | **int** parent**;** |
|  | **unsigned** **short** layer**;** *//节点所在的层(与block有关，设计到变量的作用域，使用的范围,用uchar型也可以，一般不会达到255层)* |
|  | **unsigned** **char** bsource**;** *//表示是否是被分析的程序的节点(源程序包含头文件，预处理之后，被包含的头文件与源程序会放在同一个中间文件中进行处理，用uchar(unsigned char)也可以)* |
|  | **}**Node**;** |

在代码 1‑3中给出了Node节点的定义，同时，程序在parser.c中给出了遍历方式输出节点的函数段(traverse())。

具体的工作流程为，在lex中通过识别每一个标识符并return相应的符号，bison识别到之后进行相应的语法规则处理，在cgrammar-new.y中，每一个动作都通过调用Node \* link()函数（见代码 1‑4），来完成树节点的相互链接。最终在主函数中层层调用，输出。

代码 1‑4 link代码

|  |  |
| --- | --- |
|  | Node **\*** link**(int** tid**,** Node **\*** rExp**,** **...** **)** |
|  | **{** |
|  | Node **\*** node1**;** |
|  | Node **\*** node2**;** |
|  | va\_list exps**;** |
|  | Node **\*** parent\_node **=** new\_node**();** |
|  |  |
|  | parent\_node**->**id **=** tid**;** |
|  | parent\_node**->**line **=** yylineno**;** |
|  |  |
|  | **if(** rExp **==** NULL **)** **return** parent\_node**;** |
|  |  |
|  | va\_start**(**exps**,**rExp**);** |
|  | node1 **=** rExp**;** *//第一个子节点* |
|  | parent\_node**->**child **=** node1**->**node\_index**;** |
|  | node1**->**parent **=** parent\_node**->**node\_index**;** |
|  |  |
|  | node2**=**va\_arg**(**exps**,**Node **\*);** |
|  |  |
|  | **while(**node2**!=**NULL**){** |
|  | node1**->**next **=** node2**->**node\_index**;** |
|  | node2**->**prev **=** node1**->**node\_index**;** |
|  |  |
|  | node1 **=** node2**;** |
|  | node2**=**va\_arg**(**exps**,**Node **\*);** |
|  | **}** |
|  | va\_end**(** exps **);** |
|  | **return** parent\_node**;** |
|  | **}** |

具体的每个语法动作规则设计，则由整体语法制导翻译规则确定。

## 实验总结

### 遇到的难题

在实验指导PDF的帮助下，这次实验总体上十分顺利，遇到的小错误主要在于函数的声明或是调用方面，根据编程经验，查看相应的引用并修改代码文件均能顺利解决。

具体解决步骤已放于1.5中。

### 实验收获

这一次实验进一步熟悉了使用Flex和yacc联合进行语法分析的步骤，对于一款编译器的诞生有了更加进一步的感受。通过分析源代码文件，更好的理解了语法制导翻译以及抽象语法树的生成。在树节点定义、树遍历等基础编码方面也有了更加直观的学习。