

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，采用软件工程师进行系统分析、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
1.1 实验目的	1
1.2 实验内容	1
1.3 实验要求	1
1.4 移进规约冲突	1
1.4.1 移进规约冲突原因及解决	1
1.4.2 本次实验中出现的移进规约冲突	2
1.5 实验步骤	2
1.5.1 CentOS 环境下	2
1.5.2 Windows 环境下	4
1.6 符号表和语法分析树相关代码简述	5
1.6.1 符号表分析	5
1.6.2 语法分析树代码简述	6
1.7 实验总结	9
1.7.1 遇到的难题	9
1.7.2 实验收获	9

1、实验四 Bison 实验 2

1.1 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。

1.2 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。

1.3 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。

1.4 移进规约冲突

1.4.1 移进规约冲突原因及解决

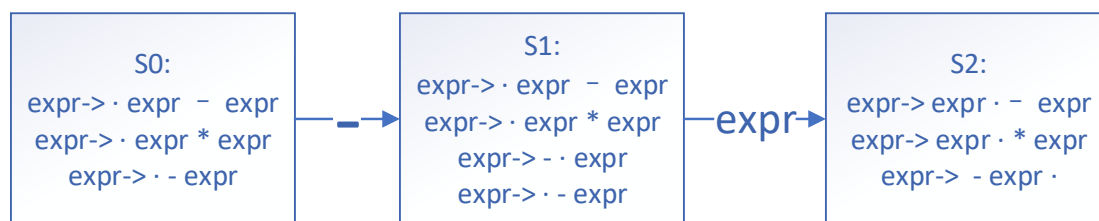
分析表是 LR 分析其的核心，它跟具体有关，包括动作表和状态转换表两部分。动作表中的元素 $action[S_i, a_i]$ 表示栈顶当前状态为 S_i ，和当前输入符号为 a_i 时完成的分析动作。其中，“移进”分析动作表示句柄尚未在分析栈顶行程，正期继续移进符号以形成句柄，“规约”表明当前分析栈的栈顶已形成当前句型的句柄 β ，要立即进行规约。

下面通过具体例子介绍移进规约冲突。

文法 $G[S]$:

$$expr \rightarrow expr - expr$$
$$expr \rightarrow expr * expr$$
$$expr \rightarrow -expr$$

识别文法 $G[S]$ 的部分可归前缀如图 1-1 所示。在项目集 S_2 中可以看到，既有移进项目($expr \rightarrow expr \cdot * expr$)，又有规约项目($expr \rightarrow -expr \cdot$)，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($expr \rightarrow -expr$)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

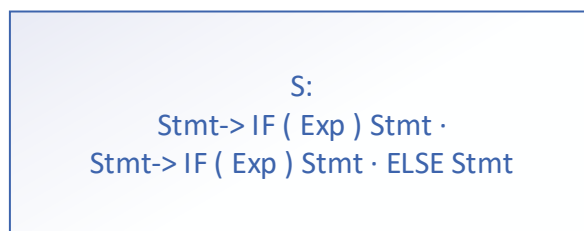
1.4.2 本次实验中出现的移进规约冲突

本次实验中含有如下文法 $G[S1]$:

$$Stmt \rightarrow IF(Exp)Stmt$$

$$Stmt \rightarrow IF(Exp)StmtELSEStmt$$

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($Stmt \rightarrow IF(Exp)StmtELSEStmt$)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

1.5 实验步骤

1.5.1 CentOS 环境下

1.5.1.1 编译 input.lex 与 cgrammar-new.y

在 CentOS 终端中，执行命令：

```
# flex input.lex
# bison -d cgrammar-new.y
cgrammar-new.y : conflicts: 1 shift/reduce
```

1.5.1.2 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
```

1.5.1.3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

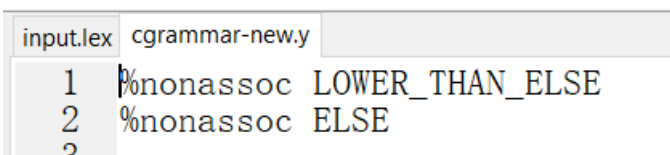
```
%nonassoc LOWER_THAN_ELSE
```

```
%nonassoc ELSE
```

在 355 行加入

```
%prec LOWER_THAN_ELSE
```

修改效果如图 1-3 所示



```
IF ' ( ' Exp ') ' Stmt %prec LOWER_THAN_ELSE { $$ = link(if_
IF ' ( ' Exp ') ' Stmt ELSE Stmt { $$ = link(iffelse_, $3, $5
SWITCH ' ( ' Exp ') ' Stmt { $$ = link(switch_, $3, $5, 0);}
```

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.”

1.5.1.4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 parse

r.c 文件，在其中的头部显示的定义 ULONG_MAX，如图 1-4 所示。

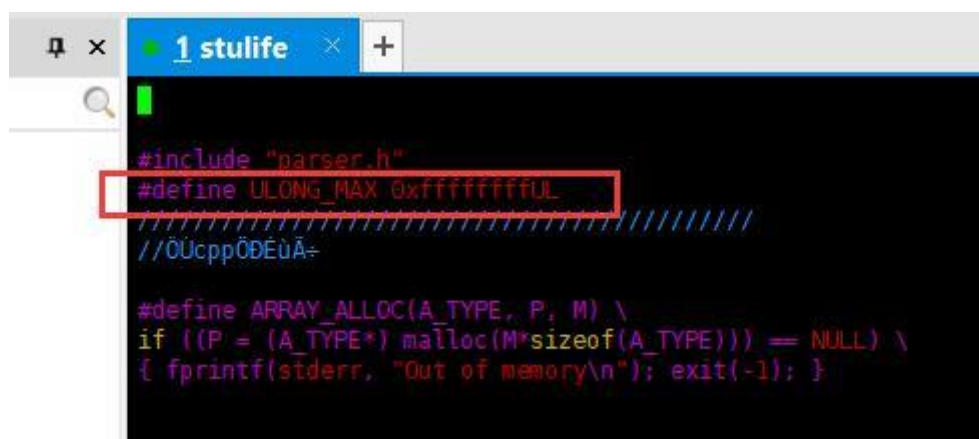


图 1-4 ULONG_MAX 定义解决

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
```

1.5.2 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，

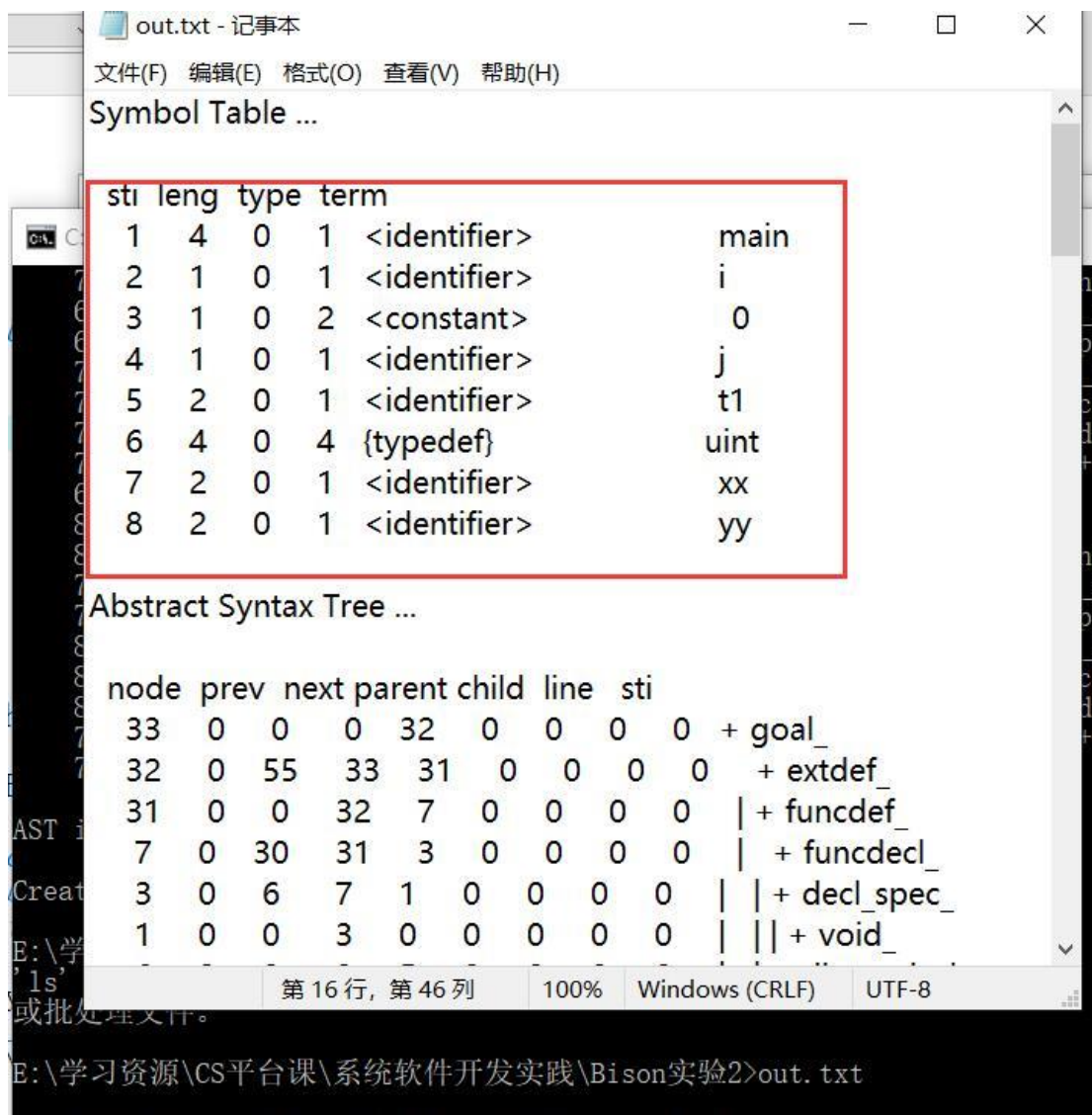


图 1-5 Windows 下运行示意图

1.6 符号表和语法分析树相关代码简述

1.6.1 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

```

1. /**/
2. void main()
3. {
4.     int i = 0;
5.     int j = 0;
6. }
7.
8. void t1()
9. {

```



```

10.    int i = 0;
11. }
12. /**/
13.
14. typedef unsigned int uint;
15.
16. uint xx;
17. 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

1.6.2 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

n	p	n	pa	ch	l	s													
o	r	e	re	il	i	t													
d	e	x	nt	d	n	i													
e	v	t			e														
3	0	6	7	1	4	0	0	0	0			+	decl_spec_						
1	0	0	3	0	4	0	0	0	0				+				void_		
6	3	0	0	5	5	0	0	0	0			+	direct_decl_						
5	0	0	6	4	4	0	0	0	0			+	funcdecl_						

下面开始具体分析它的输出代码。

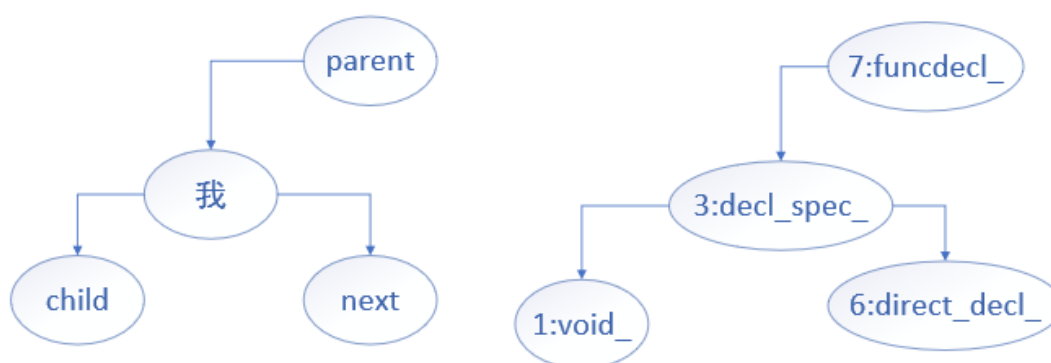


图 1-6 二叉树型

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

```

1. void main (int na, char *arg[])
2. {
3.     char *filename = "test.c";
4.     char * outfilename = "out.txt";
5.     //指向文件
6.     if(!(yyin = fopen(filename,"r"))) {
7.         printf("the file not exist\n");
8.         exit(0);
9.     }
10.    //初始化
11.    init_parser(100, 1000);
12.    if(yyvsparse())exit(1);
13.    //打开输出文件
14.    init_out_file(outfilename);
15.    print_symtab (term_symb); // Print the symbol table
16.    //遍历ast树
17.    printast(); // Print the AST
18.    //关闭输出文件
19.    term_out_file(outfilename);
20.    return ;
21. }
  
```

代码 1-3 Node(parser.h)

```

1. typedef struct Node
  
```

```

2. {
3.     unsigned short id
4.     unsigned short prod;
5.     int node_index;    //node 在node数组中的索引
6.     int     sti;
7.     int     prev;
8.     int     next;
9.     int     line;
10.    int     child;
11.    int     parent;
12.    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)
13.    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)
14. }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 linke 代码

```

1. Node * link(int tid, Node * rExp, ... )
2. {
3.     Node * node1;
4.     Node * node2;
5.     va_list exps;
6.     Node * parent_node = new_node();
7.
8.     parent_node->id = tid;
9.     parent_node->line = yylineno;
10.
11.     if( rExp == NULL ) return parent_node;
12.
13.     va_start(exps, rExp);
14.     node1 = rExp; //第一个子节点
15.     parent_node->child = node1->node_index;
16.     node1->parent = parent_node->node_index;
17.

```

```
18.     node2=va_arg(exps,Node *);
19.
20.     while(node2!=NULL){
21.         node1->next = node2->node_index;
22.         node2->prev = node1->node_index;
23.
24.         node1 = node2;
25.         node2=va_arg(exps,Node *);
26.     }
27.     va_end( exps );
28.     return parent_node;
29. }
```

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。

1.7 实验总结

1.7.1 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。

1.7.2 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。