

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 2022.3.5

学生姓名 王杰永

学 号 03190886

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	

Bison 实验二

1 实验目的.....	1
2 实验内容.....	1
3 实验要求.....	1
4 移进规约冲突.....	1
4.1 Bison 的语法分析方法	1
4.2 移进归约冲突的原因及解决	1
4.3 本次实验中出现的移进归约冲突	2
4.4 解决方案	2
5 实验步骤.....	3
5.1 Windows.....	3
5.2 Ubuntu18.04.6	5
6 符号表和语法分析树相关代码分析.....	5
6.1 符号表分析.....	5
6.2 抽象语法树分析	6
7 实验总结.....	9
7.1 你在编程过程中遇到了哪些难题?	9
7.2 你的收获有哪些?	10

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。

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。

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。

4 移进规约冲突

4.1 Bison 的语法分析方法

通过查阅资料可以发现，bison 的语法分析器可以使用两种语法分析方法——*LALR(1)*与*GLR*。大多数语法分析器使用*LALR(1)*分析就可以满足要求，尽管*LALR(1)*不如*GLR*强大，但更为迅速更容易使用。

*LALR(1)*是在当前已移进和归约出的全部文法符号的基础上，再向前查看 1 个输入符号，就可以确定分析器的动作是移进还是归约。对于少数语法，尽管没有二义性，但必须向前查看两个符号才能处理，此时可以通过添加少量代码来让 bison 使用 *GLR* 语法分析器。

本次实验使用*LALR(1)*分析。

4.2 移进归约冲突的原因及解决

LR 分析法主要由分析栈与分析表组成。其中，分析栈又分为文法符号栈与状态栈（可以合并到一起），分析表则由状态表与动作表组成。

LR 分析开始时，状态栈压入初始状态，文法符号栈压入自定义的输入串开始符。之后，根据当前栈顶状态和输入符号就可以决定下一步的动作是归约还是移进，或是接受或是报错。其中，“移进”指的当前栈顶未出现句型的句柄，将输入符号串左侧符号压入栈；“归约”指的是当前栈顶已经形成了当前句型的句柄，需要进行归约。

下面通过一个具体的例子介绍移进/归约冲突。

对于如下文法 $G[S]$:

$$\begin{aligned} \text{expr} &\rightarrow \text{expr} - \text{expr} \\ \text{expr} &\rightarrow \text{expr} * \text{expr} \\ \text{expr} &\rightarrow -\text{expr} \end{aligned}$$

该文法的部分项目集规范族及识别可归前缀的 DFA 如图 xx 所示。在项目集 S_2 中，既有移进项目 $\text{expr} \rightarrow \cdot - \text{expr}$ ，又有归约项目 $\text{expr} \rightarrow - \text{expr} \cdot$ ，此时，若输入串的下一个字符是减号-、或是乘号*，则既可以将符号移进，又可以按照产生式 $\text{expr} \rightarrow - \text{expr}$ 归约，从而产生了移进/归约冲突。



图 1 识别文法 G 可归前缀的部分 DFA

4.3 本次实验中出现的移进归约冲突

本次实验含有如下文法 $G[S1]$:

$$\begin{aligned} \text{Stmt} &\rightarrow \text{IF}(\text{Exp})\text{Stmt} \\ \text{Stmt} &\rightarrow \text{IF}(\text{Exp})\text{Stmt}\text{ELSE}\text{Stmt} \end{aligned}$$

可以预想到，当符号栈中已经存在了 $\text{IF}(\text{Exp})\text{Stmt}$ 后，若输入串下一个字符是 ELSE ，此时既可以将 ELSE 移进符号栈，也可以按照产生式 $\text{Stmt} \rightarrow \text{IF}(\text{Exp})\text{Stmt}$ 归约。于是出现了移进/归约冲突。

4.4 解决方案

在 Yacc 中，`%token` 定义的是文法中的终结符。而 `%nonassoc` 与 `token` 语法类似，用来定义终结符的结合性为不可结合，即它定义的终结符不可连续出现，或者理解为没有依赖关系。`%nonassoc` 通常与 `%prec` 一起使用，用于指定一个规则的优先级。

在 `cgrammar - new.y` 文件中加入如下声明：

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE
```

并将产生移进归约冲突的产生式改成如下形式，便可以解决冲突。

```
| IF '(' Exp ')' Stmt %prec LOWER_THAN_ELSE { $$ = link(if_, $3, $5, 0); }
| IF '(' Exp ')' Stmt ELSE Stmt { $$ = link(elseif_, $3, $5, $7, 0); }
```

图 2 解决冲突

这样的修改方案,使得 LOWER_THAN_ELSE 的优先级小于 ELSE 的优先级,同时,产生移进归约冲突的第一个产生式优先级被指定为 LOWER_THAN_ELSE,即较小的优先级。因此当冲突发生时,编译器将会选择第二个产生式,即先移进,后归约。

5 实验步骤

5.1 Windows

在对 cgrammar-new.y 文件中加入了 4.4 中的移进/归约冲突解决方案后,对 input.lex 与 cgrammar-new.y 文件编译。终端执行命令:

```
flex input.lex
```

```
bison -d cgrammar-new.y
```

当前工作目录下生成了 lex.yy.c 与 cgrammar-new.tab.c、cgrammar-new.tab.h 文件。

通过 gcc 编译器,对生成的文件编译。

```
gcc lex.yy.c cgrammar-new.tab.c main.c parser.c -o 2_2
```

出现如图 3 所示的错误。

```
D:\Here\recently\系统软件开发实践\Bison2\code>gcc lex.yy.c cgrammar-new.tab.c main.c parser.c -o 2_2
input.lex: In function 'comment':
input.lex:143:14: warning: implicit declaration of function 'yyinput'; did you mean 'yyunput'? [-Wimplicit-function-decl
aration]
    while ((c = yyinput()) != '*' && c != 0)
                  ^
input.lex:143:14: note: 'yyunput' was declared here
input.lex: In function 'check_type':
input.lex:182:23: error: 'yylineno' undeclared (first use in this function); did you mean 'yyleng'?
    yylval.node->line = yylineno;
                        ^
input.lex:182:23: note: each undeclared identifier is reported only once for each function it appears in
input.lex: In function 'check_constant':
input.lex:206:23: error: 'yylineno' undeclared (first use in this function); did you mean 'yyleng'?
    yylval.node->line = yylineno;
                        ^
input.lex: In function 'check_string':
input.lex:217:23: error: 'yylineno' undeclared (first use in this function); did you mean 'yyleng'?
    yylval.node->line = yylineno;
                        ^
main.c: In function 'main':
main.c:42:2: warning: implicit declaration of function 'print_syntab'; did you mean 'printast'? [-Wimplicit-function-decl
aration]
    print_syntab (term_symb); // Print the symbol table contents.
    ^
main.c:42:2: note: 'printast' was declared here
D:\Here\recently\系统软件开发实践\Bison2\code>
```

图 3 第一次编译时的错误

根据实验指导 pdf, 可以找到解决方案。

打开 lex.yy.c 文件，修改下图所示的两处内容。

```
#ifdef __cplusplus
static int yyinput()
#else
static int input()
#endif
{
    int c;

    *yy_c_buf_p = yy_hold_char;
```

图 4 第一处修改

```
if ( ! yy_did_buffer_switch_on_eof
    YY_NEW_FILE;

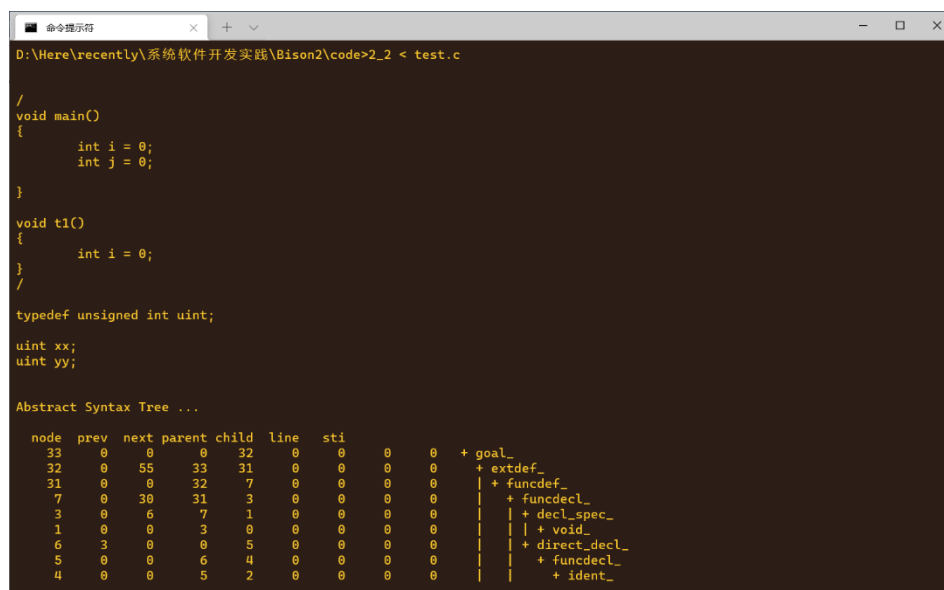
#ifdef __cplusplus
    return yyinput();
#else
    return input();
#endif
}

case EOB_ACT_CONTINUE_SCAN:
    yy_c_buf_p = yytext_ptr + offset;
    break;
}
```

图 5 第二处修改

将第一处全部注释，添加 static int yyinput(); 将第二处全部注释，添加 return yyinput() 语句。

对于 yylineno 未定义的问题，只需要在 lex.yy.c 中全局定义 int yylineno = 0;即可解决。重新编译，编译成功。



```
命令提示符
D:\Here\recently\系统软件开发实践\Bison2\code>2_2 < test.c

/
void main()
{
    int i = 0;
    int j = 0;
}

void t1()
{
    int i = 0;
}
/

typedef unsigned int uint;

uint xx;
uint yy;

Abstract Syntax Tree ...

node  prev  next  parent  child  line  sti  + goal_
33    0     0     0     32    0     0     0     + goal_
32    0     55    33    31    0     0     0     + extdef_
31    0     0     32    7     0     0     0     | + funcdecl_
7     0     30    31    3     0     0     0     | + funcdecl_
3     0     6     7     1     0     0     0     | | + decl_spec_
1     0     0     3     0     0     0     0     | | | + void_
6     3     0     0     5     0     0     0     | | | + direct_decl_
5     0     0     6     4     0     0     0     | | | + funcdecl_
4     0     0     5     2     0     0     0     | | | + ident_
```

图 6 windows 系统运行结果

5.2 Ubuntu18.04.6

Ubuntu 与 Windows 的步骤类似，唯一的区别在于，当编译全部的.c 文件时，会提示找不到 io.h 这个头文件的错误。

查阅资料后发现，Linux 上的 io.h 头文件并没有包含在标准库/usr/include 中。此时的想法是找到 io.h 文件，将其复制到/usr/include 目录下，从而解决问题。

首先使用命令 `find /usr/include -name io.h`，找到 io.h 的真实路径。

```
chen@chen:~/桌面/chen/e4$ find /usr/include -name io.h
/usr/include/x86_64-linux-gnu/sys/io.h
```

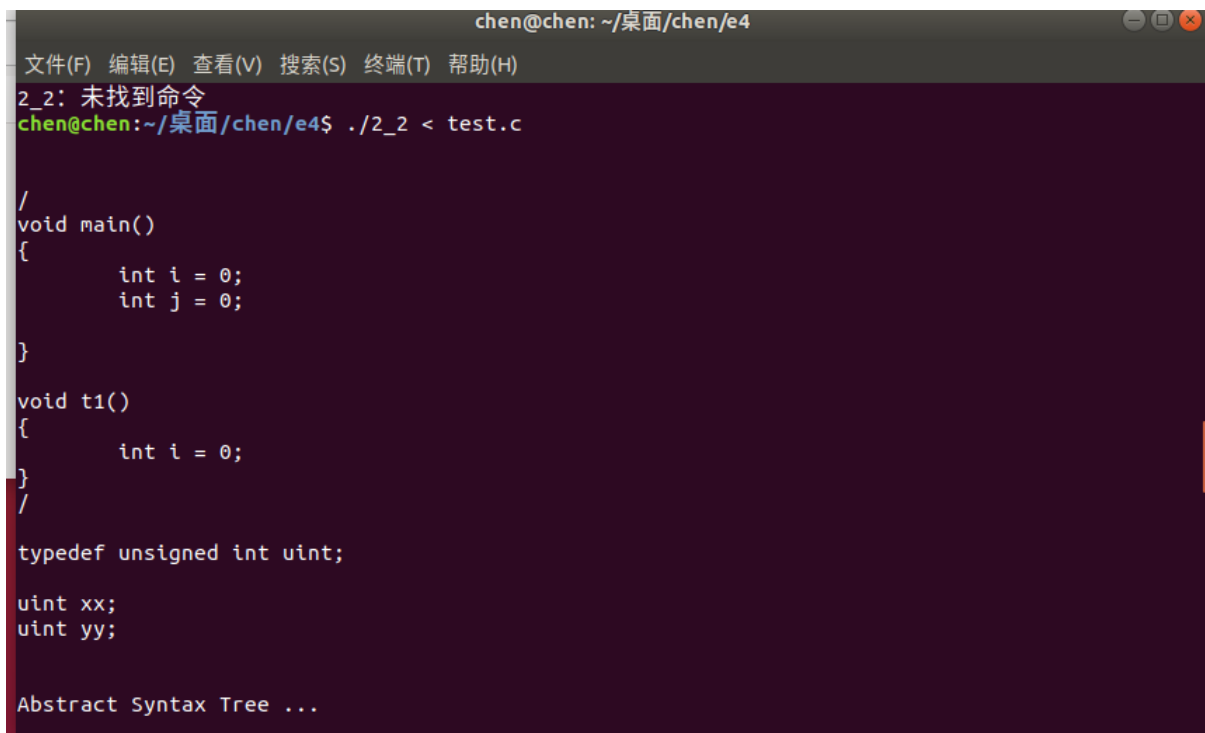
图 7 io.h 的绝对路径

使用命令 `sudo cp /usr/include/x86_64-linux-gnu/sys/io.h /usr/include`，完成复制。

```
chen@chen:~/桌面/chen/e4$ sudo cp /usr/include/x86_64-linux-gnu/sys/io.h /usr/include
chen@chen:~/桌面/chen/e4$ find /usr/include -name io.h
/usr/include/io.h
/usr/include/x86_64-linux-gnu/sys/io.h
chen@chen:~/桌面/chen/e4$
```

图 8 复制操作

之后可以正常编译，并输出结果了。



```
chen@chen: ~/桌面/chen/e4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
2_2: 未找到命令
chen@chen:~/桌面/chen/e4$ ./2_2 < test.c

/
void main()
{
    int i = 0;
    int j = 0;
}

void t1()
{
    int i = 0;
}

/

typedef unsigned int uint;

uint xx;
uint yy;

Abstract Syntax Tree ...
```

图 9 Linux 实验结果

6 符号表和语法分析树相关代码分析

6.1 符号表分析

符号表中存储了所有用户自定义的变量名、值等，对于 `int` 等保留字则不会在符号表中体现。

输出的符号表见下表。

表 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

从表中可以看出，语法分析程序成功分析出了 `main`、`i`、`j`、`t1`、`xx`、`yy` 等标识符，并且识别出了常量 `0`，且辨别了 `uint` 是 `typedef` 重定义的类型。在此基础上，成功识别了每一个标识符的长度。

符号表是抽象语法树的生成过程中不断完善的，因此其相关代码的分析放在了 6.2 抽象语法树分析中。

6.2 抽象语法树分析

语法分析过程中生成的抽象语法树，与二叉树不同，每个节点的子节点数量不再是固定的 2，而是任意数量。因此，对于这样一棵多叉树，我们需要使用特殊的存储结构——子女兄弟链，来存储。

使用子女兄弟链表示二叉树，其每个节点除了值 `val` 外，还有两个 `Node` 类型的指针，左指针指向该节点的一个子节点，右指针指向该节点的一个兄弟节点。因此，子女兄弟链结构是一种存储多叉树的二叉树。

在 `parser.h` 中，有节点 `Node` 的定义：

```
1. typedef struct Node
2. {
3.     unsigned short id;           // Node id number
4.     unsigned short prod;        // Production number
5.     int    node_index;          // node 在 node 数组中的索引
6.     int    sti;                 // Symbol-table index (perm or temp var).
7.     int    prev;                // Previous node.
8.     int    next;                // Next node.
```

```

9.  int    line;    // Line number.
10. int    child;   // Child node.
11. int    parent;  // Parent node.
12. unsigned short layer;
13. unsigned char bsource;
14. }Node;

```

对照生成的抽象语法树，以及 node 结构体的注释，可以看出，node_index 是结点索引，parent 是父节点索引，child 是子节点索引，next 是兄弟节点索引。根据这些值，足够我们恢复整棵抽象语法树了。

挑选部分输出的语法分析树判断如下，根据子女兄弟链的表示规则，整棵树根节点为 decl_init_，其有两个子节点 decl_spec_ 与 init_declarators_；对于子节点 decl_spec_，仅有一个子节点也是叶子节点——int_。最后，复原的语法分析树如图 10 所示。

```

+ decl_init_
| + decl_spec_
| | + int_
| + init_declarators_
|   + declaratorinit_
|     + direct_decl_
|       + ident_
|         + IDENT_ (i)
|       + assign_
|         + CONST_ (0)

```

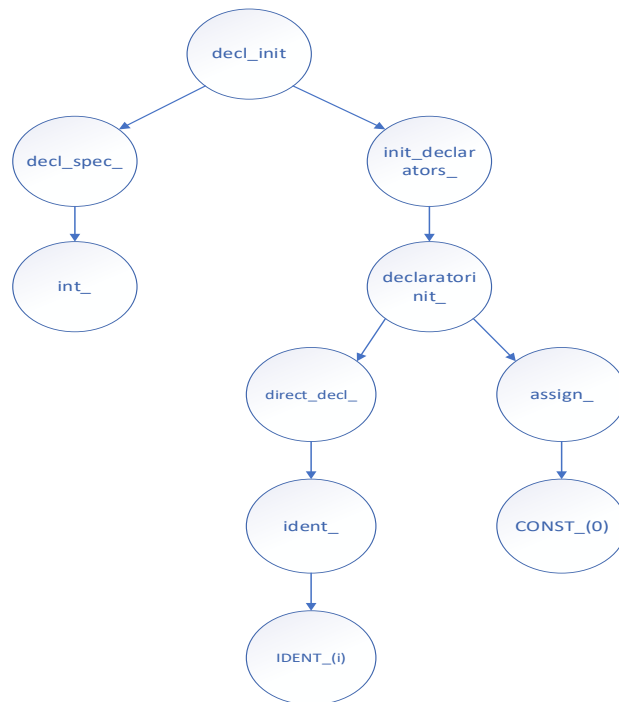


图 10 语法分析树

语法分析树中的所有叶子节点自左向右排列，得到的是当前分析出的句子。所选出的部分语法分析树所表达的句子是 `int_ IDENT_(i) CONST_(0)`。没有等于号的原因根据常数 0 的父节点 `token` 类型为 `assign_`，推测在词法分析或是语法分析中对等号进行了特殊处理，因此常数 0 的父节点才会标记为 `assign`。

语法分析程序的主函数代码如下。

```
1. void main (int na, char *arg[])
2. {
3.     char *filename = "test.c";
4.     char * outfilename = "out.txt";
5.
6.     //指向文件
7.     if(!(yyin = fopen(filename, "r"))) {
8.         printf("the file not exist\n");
9.         exit(0);
10.    }
11.
12.    //初始化
13.    init_parser(100, 1000);
14.
15.    if(yparse())exit(1);
16.
17.    //打开输出文件
18.    init_out_file(outfilename);
19.
20.    print_symtab (term_symb); // Print the symbol table contents.
21.
22.    //遍历 ast 树
23.    printast();           // Print the AST, if ast option indicates.
24.
25.
26.    //关闭输出文件
27.    term_out_file(outfilename);
28.
29.    return ;
30.
31. }
```

真正进行语法分析的部分是函数 `print_symtab()`；以及函数 `printast()`；

`print_symtab()`函数负责生成符号表，其主要部分如下。在 `for` 循环中，对遍历到的每一个符号分别打印该节点的节点索引，长度，类型等信息。

```

1. for (i = 1; i < n_symbols; i++)
2.     {
3.         fprintf (filedesc, "%5d %5d %5d %5d  %-30s  %s\n",
4.             i,
5.             symbol[i].length,
6.             symbol[i].type,
7.             symbol[i].term,
8.             term_symb[symbol[i].term],
9.             symbol_name(i, filedesc));
10.    }

```

printast()函数较为复杂，由于树的定义可以是递归的，因此使用了大量递归逻辑去打印整棵树。对某一子树打印其信息时，调用 print_node()函数，其主要的部分函数体如下。

```

1. printf (" %5d %5d %5d %6d %5d %5d %5d %5d %5d %s%s",
2.    n,
3.    node[n].prev,
4.    node[n].next,
5.    node[n].parent,
6.    node[n].child,
7.    node[n].line,
8.    sti,
9.    node[n].layer,
10.   node[n].bsource,
11.   indent,
12.   node_name[node[n].id]
13. );

```

最后，当某一符号串序列可以被归约时，其语法动作则由语法制导翻译完成。

7 实验总结

7.1 你在编程过程中遇到了哪些难题？

在编译阶段，遇到的主要问题如 io.h、yyinput 等都已根据实验指导 PDF 解决。其余的小问题及解决步骤均在 5 实验步骤 中体现。

在对符号表及抽象语法树的相关代码理解时，由于对 C 的编码风格及相关库函数不够熟练，导致代码理解起来较为困难。在花费了足够长的时间后，还是可以看懂一些的。

7.2 你的收获有哪些？

本次实验的代码复杂，flex 和 bison 的联合编程更加紧密，让我对编译器的前端工作有了更深的理解。特别是以子女兄弟链的模式去存储抽象语法树，这使得在数据结构课程上学习的相关理论得以实践。最后，对词法、语法分析，以及语法制导翻译、抽象语法树的生成得以更好的理解。