



中国矿业大学  
CHINA UNIVERSITY OF MINING AND TECHNOLOGY

# 中国矿业大学计算机学院 2017级本科生课程设计报告

## 实验四 Bison实验2

课程名称 系统软件开发实践

报告时间 2020.04.22

学 号

# 目录

1	实验任务	1
2	移进规约冲突	1
2.1	冲突产生原因 . . . . .	1
2.2	冲突解决办法 . . . . .	1
3	在Windows环境下实验	2
3.1	实验过程 . . . . .	2
3.2	实验排错 . . . . .	2
3.3	实验结果 . . . . .	4
4	在Ubuntu环境下实验	6
4.1	实验过程 . . . . .	6
4.2	实验排错 . . . . .	7
4.3	实验结果 . . . . .	7
5	简述符号表和语法分析树的相关代码	9
5.1	符号表相关代码 . . . . .	9
5.2	分析树相关代码 . . . . .	11
6	实验感想	11

## 实验四 Bison实验2

### 1 实验任务

1. 阅读提供的C语言文法相关资料；
2. 阅读 Flex源文件 input.lex、Bison 源文件 cgrammar new.y，并上机调试（具体过程参考《实验四借助FlexBison进行语法分析.pdf》）；
3. 重点掌握移进/规约冲突产生的原因及解决方法；
4. 掌握符号表和抽象语法树的生成过程。

### 2 移进规约冲突

#### 2.1 冲突产生原因

```
1  expr:
2      expr - expr
3      | expr * expr
4      | - expr
```

对于上述代码，输入  $-1 * 2$  匹配完1后，可以根据 `expr: expr * expr` 继续移进 `*`，但也可以根据 `expr: -expr` 规约为1。

所以解析方式有两种：

- $-1 * 2 = (-1) * 2 = -2$
- $-1 * 2 = (-1 * 2) = -2$

虽然两种解析的结果一样，但是程序不知道用哪种方式解析。之所以冲突，是因为移进和归约的优先级没有确定，即符号 `*` 和规则 `expr: -expr` 的优先级没有确定，出现了 `*` 就不知道该移进 `*` 还是利用规则来归约了。

#### 2.2 冲突解决办法

可以通过定义优先级来解决冲突，定义方法如下：

1. 使用 `%prec` 定义规则对应的符号（即定义此规则和哪个符号的优先级相同）；
2. 使用 `%left %right %noassoc %precedence` 来定义符号的优先级和结合性（分别是左结合、右结合、没有结合性、未定义的结合性）。

本次实验也出现了移进规约冲突，如下：

```

Stmt: IF '(' Exp ')' Stmt .
      | IF '(' Exp ')' Stmt . ELSE Stmt

```

那么 `IF '(' Exp ')' IF '(' Exp ')' Stmt . ELSE Stmt` 有如下两种理解方法:

- `IF '(' Exp ')' { IF '(' Exp ')' Stmt . ELSE Stmt }`
- `IF '(' Exp ')' { IF '(' Exp ')' Stmt }. ELSE Stmt`

这样就出现了冲突, 解决方式是提高移进的优先级, 使它高于规约, 具体方法在实验过程中介绍。

### 3 在Windows环境下实验

#### 3.1 实验过程

将实验代码放在实验目录 `ex4` 下, 打开 Developer Command Prompt for VS 2019, 输入 `flex -l input.lex` 生成 `lex.yy.c` 文件, 如图1所示。然后编译 bison 文件, 命令行输入 `bison -d cgrammar-new.y`, 生成 `cgrammar-new.tab.c` 与 `cgrammar-new.tab.h` 文件, 如图2所示。

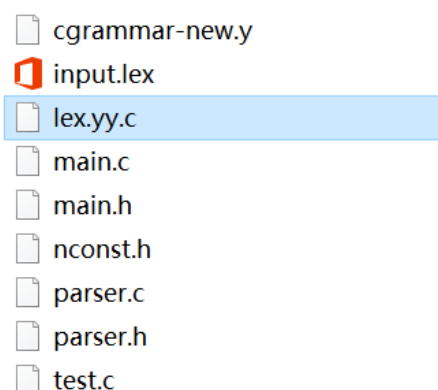


图 1: flex运行结果

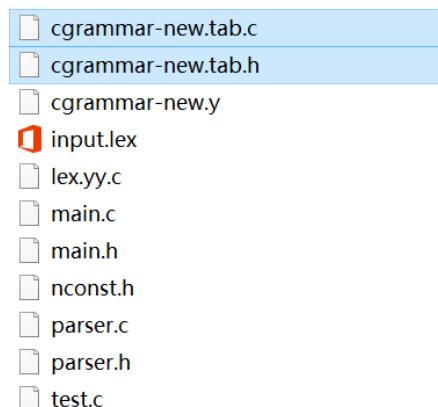


图 2: bison运行结果

```

C:\Users\yuan\Documents\课程\系统软件开发实践\报告\04实验四\ex4>flex -l input.lex
C:\Users\yuan\Documents\课程\系统软件开发实践\报告\04实验四\ex4>bison -d cgrammar-new.y
cgrammar-new.y: conflicts: 1 shift/reduce
C:\Users\yuan\Documents\课程\系统软件开发实践\报告\04实验四\ex4>_

```

图 3: 移进规约错误

但是出现了移进规约错误, 如图3所示, 下面查找错误。

#### 3.2 实验排错

命令行输入 `bison -v cgrammar-new.y`, 会生成一个日志文件 `cgrammar-`

new.output, 打开, 从文件的第一行 **State 341 conflicts: 1 shift/reduce** 可以看出是 state 341 出现了移进规约错误, 查看此位置, 如图4所示。

```
state 341

168 Stmt: IF '(' Exp ')' Stmt .
169   | IF '(' Exp ')' Stmt . ELSE Stmt

    ELSE shift, and go to state 347

    ELSE      [reduce using rule 168 (Stmt)]
$default reduce using rule 168 (Stmt)
```

图 4: state 341 错误

```
cgrammar-new.y x
cgrammar-new.y
1  %nonassoc LOWER_THAN_ELSE
2  %nonassoc ELSE
3
4  %{
5
6  #include "parser.h"
```

图 5: cgrammar-new.y 文件改正1

修改 cgrammar-new.y 文件, 在文件头部加入 **%nonassoc LOWER\_THAN\_ELSE** 与 **%nonassoc ELSE** 语句, 如图所示。

并寻找错误代码位置, 加入 **%prec LOWER\_THAN\_ELSE** 如图6所示, 再次输入 **bison -d cgrammar-new.y** 编译, 结果无误。

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

图 6: cgrammar-new.y 文件改正2

```
lex.yy.c
cgrammar-new.tab.c
main.c
parser.c
正在生成代码...
Microsoft (R) Incremental Linker Version 14.24.28316.0
Copyright (C) Microsoft Corporation. All rights reserved.

/out:lex.yy.exe
lex.yy.obj
cgrammar-new.tab.obj
main.obj
parser.obj
lex.yy.obj : error LNK2019: 无法解析的外部符号 _yyinput, 该符号在函数 _comment 中被引用
lex.yy.exe : fatal error LNK1120: 1 个无法解析的外部命令
```

图 7: 解析错误

然后输入 **cl lex.yy.c cgrammar-new.tab.c main.c parser.c** 编译c代码, 结果如图7所示, 出现解析错误, 原因是缺少 **yyinput()** 函数的定义, 根据资料了解到此函数功能与 **input()** 函数功能一致, 所以在 **input.lex** 文件的C代码部分, **comment()** 函数前加入如下代码, 定义一个 **yyinput()** 函数。

```
1 char yyinput(){
2     return input();
3 }
```

在浏览代码时, 发现注释中存在乱码, 删除后, 重新输入 **cl lex.yy.c cgrammar-new.tab.c main.c parser.c** 编译c代码, 编译成功, 结果如图8所示。

```
lex.yy.c
cgrammar-new.tab.c
main.c
parser.c
正在生成代码...
Microsoft (R) Incremental Linker Version 14.24.28316.0
Copyright (C) Microsoft Corporation. All rights reserved.

/out:lex.yy.exe
lex.yy.obj
cgrammar-new.tab.obj
main.obj
parser.obj
```

图 8: 编译正确结果

3.3 实验结果

编译成功后，下面开始分析代码，输入 `lex.yy.exe < test.c` 得到输出，如图9所示。

```
C:\Users\yuan\Documents\课程\系统软件开发实践\报告\04实验四\ex4>lex.yy.exe < 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
33    0     0      0     32    9    0
32    0    55     33     31    9    0
31    0     0     32     7     9    0
7     0    30     31     3     5    0
3     0     6     7     1     4    0
+ goal_
+ extdef_
+ funcdef_
+ funcdecl_
+ decl_spec
```

图 9: 输出结果

还有一个 `out.txt` 文件，打开后，里面有符号表和抽象语法树，内容如下：

```
Symbol Table ...

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

Abstract Syntax Tree ...
```

node	prev	next	parent	child	line	sti					
33	0	0	0	32	9	0	0	0	+	goal_	
32	0	55	33	31	9	0	0	0	+	extdef_	
31	0	0	32	7	9	0	0	0		+ funcdef_	
7	0	30	31	3	5	0	0	0		+ funcdecl_	
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_	
4	0	0	5	2	4	0	0	0		+ ident_	
2	0	0	4	0	4	1	0	0		+ IDENT_ (main)	
30	7	0	0	29	9	0	0	0		+ funcbody_	
29	0	0	30	18	9	0	0	0		+ compound_stmt_	
18	0	0	29	17	6	0	0	0		+ declarations_	
17	0	28	18	10	6	0	0	0		+ decl_init_	
10	0	16	17	8	6	0	0	0		+ decl_spec_	
8	0	0	10	0	6	0	0	0		+ int_	
16	10	0	0	15	6	0	0	0		+ init_declarators_	
15	0	0	16	12	6	0	0	0		+ declaratorinit_	
12	0	14	15	11	6	0	0	0		+ direct_decl_	
11	0	0	12	9	6	0	0	0		+ ident_	
9	0	0	11	0	6	2	0	0		+ IDENT_ (i)	
14	12	0	0	13	6	0	0	0		+ assign_	
13	0	0	14	0	6	3	0	0		+ CONST_ (0)	
28	17	0	0	21	7	0	0	0		+ decl_init_	
21	0	27	28	19	7	0	0	0		+ decl_spec_	
19	0	0	21	0	7	0	0	0		+ int_	
27	21	0	0	26	7	0	0	0		+ init_declarators_	
26	0	0	27	23	7	0	0	0		+ declaratorinit_	
23	0	25	26	22	7	0	0	0		+ direct_decl_	
22	0	0	23	20	7	0	0	0		+ ident_	
20	0	0	22	0	7	4	0	0		+ IDENT_ (j)	
25	23	0	0	24	7	0	0	0		+ assign_	
24	0	0	25	0	7	3	0	0		+ CONST_ (0)	
55	32	66	0	54	14	0	0	0	+	extdef_	
54	0	0	55	40	14	0	0	0		+ funcdef_	
40	0	53	54	36	12	0	0	0		+ funcdecl_	
36	0	39	40	34	11	0	0	0		+ decl_spec_	
34	0	0	36	0	11	0	0	0		+ void_	
39	36	0	0	38	12	0	0	0		+ direct_decl_	
38	0	0	39	37	11	0	0	0		+ funcdecl_	
37	0	0	38	35	11	0	0	0		+ ident_	
35	0	0	37	0	11	5	0	0		+ IDENT_ (t1)	
53	40	0	0	52	14	0	0	0		+ funcbody_	
52	0	0	53	51	14	0	0	0		+ compound_stmt_	
51	0	0	52	50	13	0	0	0		+ declarations_	
50	0	0	51	43	13	0	0	0		+ decl_init_	
43	0	49	50	41	13	0	0	0		+ decl_spec_	

```

41      0      0      43      0      13      0      0      0      |      | + int_
49    43      0      0      48      13      0      0      0      |      + init_declarators_
48      0      0      49      45      13      0      0      0      |      + declaratorinit_
45      0      47      48      44      13      0      0      0      |      + direct_decl_
44      0      0      45      42      13      0      0      0      |      | + ident_
42      0      0      44      0      13      2      0      0      |      | + IDENT_ (i)
47    45      0      0      46      13      0      0      0      |      + assign_
46      0      0      47      0      13      3      0      0      |      + CONST_ (0)
66    55      75      0      65      17      0      0      0      + extdef_
65      0      0      66      56      17      0      0      0      | + decl_init_
56      0      64      65      57      17      0      0      0      | + typedef_
57      0      0      56      60      17      0      0      0      | | + unsigned_
60      0      0      57      58      17      0      0      0      | | + decl_spec_
58      0      0      60      0      17      0      0      0      | | + int_
64    56      0      0      63      17      0      0      0      | + init_declarators_
63      0      0      64      62      17      0      0      0      | + declarator_
62      0      0      63      61      17      0      0      0      | + direct_decl_
61      0      0      62      59      17      0      0      0      | + ident_
59      0      0      61      0      17      6      0      0      | + IDENT_ (uint)
75    66      84      0      74      19      0      0      0      + extdef_
74      0      0      75      69      19      0      0      0      | + decl_init_
69      0      73      74      67      19      0      0      0      | + decl_spec_
67      0      0      69      0      19      6      0      0      | | + type_name_ (uint)
73    69      0      0      72      19      0      0      0      | + init_declarators_
72      0      0      73      71      19      0      0      0      | + declarator_
71      0      0      72      70      19      0      0      0      | + direct_decl_
70      0      0      71      68      19      0      0      0      | + ident_
68      0      0      70      0      19      7      0      0      | + IDENT_ (xx)
84    75      0      0      83      20      0      0      0      + extdef_
83      0      0      84      78      20      0      0      0      + decl_init_
78      0      82      83      76      20      0      0      0      + decl_spec_
76      0      0      78      0      20      6      0      0      | + type_name_ (uint)
82    78      0      0      81      20      0      0      0      + init_declarators_
81      0      0      82      80      20      0      0      0      + declarator_
80      0      0      81      79      20      0      0      0      + direct_decl_
79      0      0      80      77      20      0      0      0      + ident_
77      0      0      79      0      20      8      0      0      + IDENT_ (yy)

```

End of Output.

## 4 在Ubuntu环境下实验

### 4.1 实验过程

为了便于实验，将windows下实验代码复制到 ubuntu 虚拟机，如图10所示。

终端输入 `flex -l input.lex` 生成 `lex.yy.c` 文件，如图11所示。然后编译



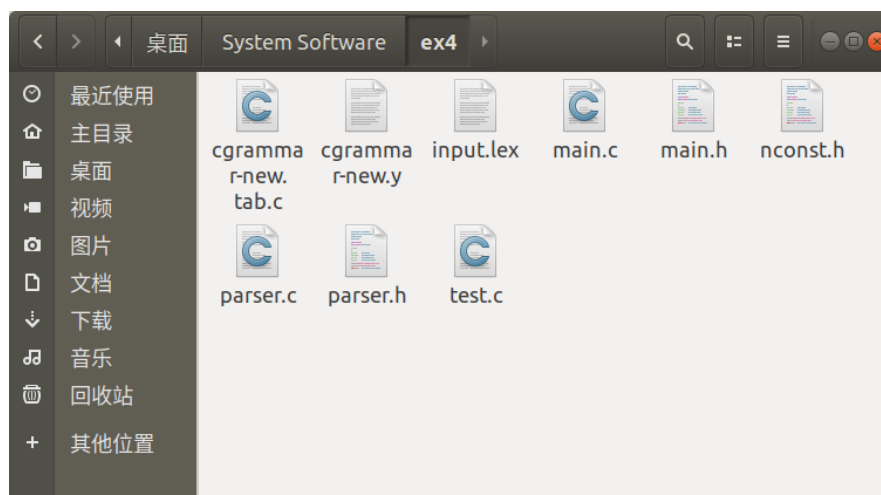


图 10: 文件列表

bison 文件，命令行输入 `bison -d cgrammar-new.y`，生成 `cgrammar-new.tab.c` 与 `cgrammar-new.tab.h` 文件，如图12所示。

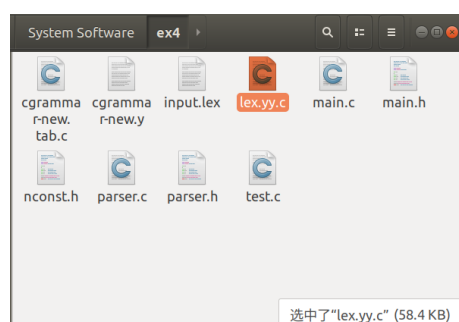


图 11: flex运行结果

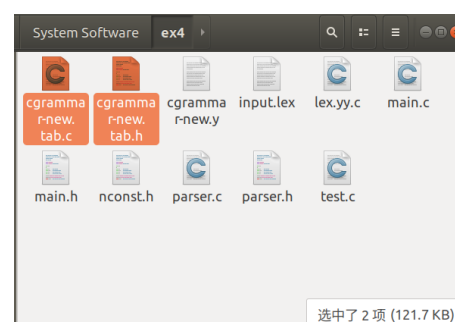


图 12: bison运行结果

然后输入 `cl lex.yy.c cgrammar-new.tab.c main.c parser.c` 编译c代码，结果如图13所示，出现两个错误，一个警告，下面解决问题。

## 4.2 实验排错

第一个错误是 `cgrammar-new.y` 代码存在代码问题，缺少一个 ; 号，在响应位置修改即可，如图14所示。

第二个是警告，原因是没有显式定义函数 `print_symtab()`，通过查看调用此函数位置的代码，得到此函数的参数和返回值类型，所以在 `main.c` 文件头部定义此函数 `extern void print_symtab(char * term_symb[]);`，如图15所示。

第三个也是错误，缺少 `ULONG_MAX` 的定义，通过查阅资料，发现它是 `limits.h` 头文件的一个常量，在 `parse.c` 文件头部引用即可，如图16所示。

## 4.3 实验结果

编译成功后，下面开始分析代码，输入 `./a.out < test.c` 得到输出，如

```

yuan@ubuntu: ~/桌面/System Software/ex4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
yuan@ubuntu:~/桌面/System Software/ex4$ bison -d cgrammar-new.y
yuan@ubuntu:~/桌面/System Software/ex4$ cc lex.yy.c cgrammar-new.tab.c main.c pa
rser.c
cgrammar-new.y: In function 'yyparse':
cgrammar-new.y:159:108: error: expected ';' before '}' token
    | Declaration_Spec init_decl_s ';' { $$ = link(decl_init_, $1, $2, 0); adjust
_term($$, 4) }
                                ^
main.c: In function 'main':
main.c:36:2: warning: implicit declaration of function 'print_syntab'; did you m
ean 'printast'? [-Wimplicit-function-declaration]
    print_syntab (term_syntab); // Print the symbol table contents.
    ^~~~~~
parser.c: In function 'init_syntab':
parser.c:339:18: error: 'ULONG_MAX' undeclared (first use in this function); did
you mean 'RAND_MAX'?
    hashdiv = ULONG_MAX / max_cells + 1;
              ^~~~~~
              RAND_MAX
parser.c:339:18: note: each undeclared identifier is reported only once for each
function it appears in
yuan@ubuntu:~/桌面/System Software/ex4$

```

图 13: 编译错误

```

cgrammar-new.y
~/桌面/System Software/ex4
保存(S)

;

Declaration
: Declaration_Spec ';' { $$ = link(decl_, $1, 0); }
| Declaration_Spec init_decl_s ';' { $$ = link(decl_init_, $1, $2, 0);
adjust_term($$, 4) }
;

Declaration_Spec
: StorageSpec { $$ = link(decl_spec_, $1, 0); }
| StorageSpec Declaration_Spec { $$ = $1; addchild($1, $2); }
| TypeSpec { $$ = link(decl_spec_, $1, 0); }
| TypeSpec Declaration_Spec { $$ = $1; addchild($1, $2); }
| TypeQua { $$ = link(decl_spec_, $1, 0); }
| TypeQua Declaration_Spec { $$ = $1; addchild($1, $2); }

```

图 14: 语法错误改正

```

main.c
~/桌面/System Software/ex4
保存(S)

#include "main.h"

extern int yyparse();
extern void print_syntab(char * term_syntab[]);
extern FILE *yyin;

extern char * term_syntab[];

extern void init_parser(int, int);
extern void printast
()
```

图 15: main.c 代码改正

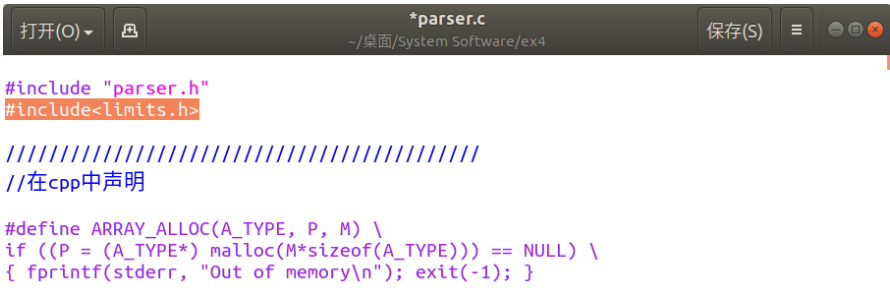


图 16: parse.c 代码改正

图17所示，同样的，也会得到一个 out.txt 文件，内容同 windows 下的结果，此处不在赘述。

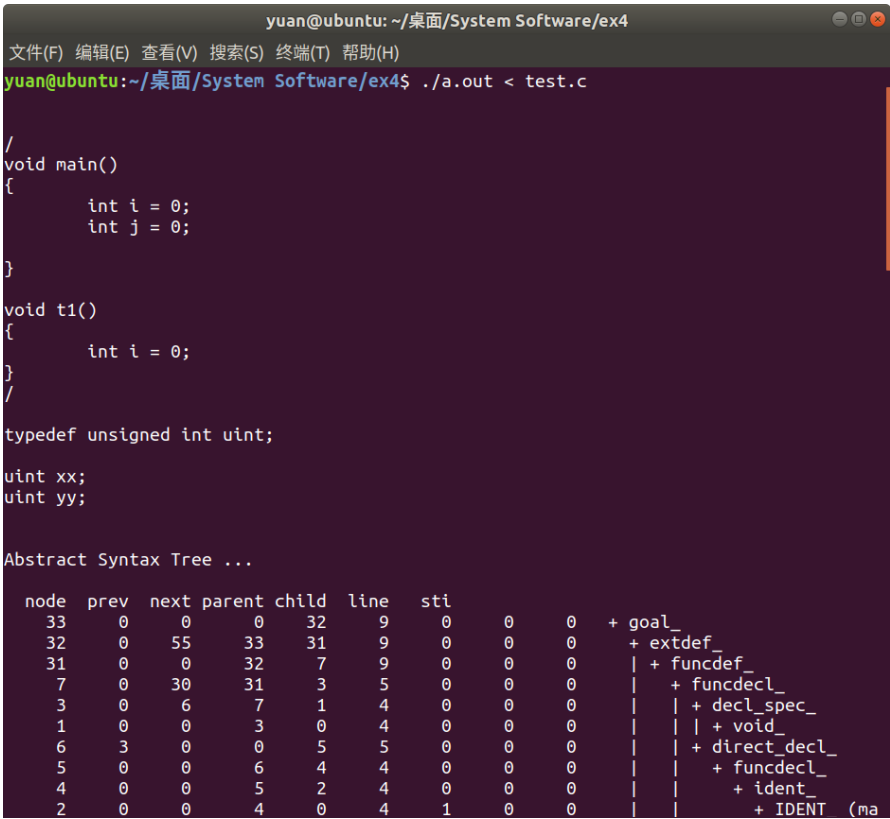


图 17: 输出结果

## 5 简述符号表和语法分析树的相关代码

### 5.1 符号表相关代码

由输出文件 out.txt 可以看出下面是获取符号表名称的主要代码，symbol\_name 的作用是根据索引 i 从程序中获取符号名称，如果符号的长度≤ 2000，则输出过长信息，同时停止程序，但在此程序执行之前需要执行 init\_syntab 函数来初始化符号表。

```

1  char* symbol_name (short i, FILE * filedesc){
2      static char name[2000];
3      char* p;
4      short L;
5
6      if (i == 0){
7          name[0] = 0;
8          return name;
9      }
10
11     p = symbol[i].name;
12     L = symbol[i].length;
13
14     if (L >= 2000){
15         for (i = 0; i < 100; i++) name[i] = p[i];
16         name[i] = 0;
17         printf (          "Symbol length of %d is too big (>= 2000).\n", L);
18         fprintf (filedesc, "Symbol length of %d is too big (>= 2000).\n", L);
19         printf (          "for '%s'\n.", name);
20         fprintf (filedesc, "for '%s'\n.", name);
21         quit();
22     }
23     for (i = 0; i < L; i++) name[i] = p[i];
24     name[i] = 0;
25     return name;
26 }

```

下面这些代码是输出符号表的主要代码，如果符号数量> 1，则遍历符号数组，同时调用 symbol\_name 获取符号名称进行输出。

```

1  void print_syntab (char* term_symb[]){
2      short i;
3      FILE* filedesc = out_file_fp;
4      if (n_symbols > 1){
5          fprintf (filedesc, "Symbol Table ...\n\n");
6          fprintf (filedesc, "  sti  leng  type  term  \n");
7
8          for (i = 1; i < n_symbols; i++){
9              fprintf (filedesc, "%5d %5d %5d %5d  %-30s  %s\n",
10                  i,
11                  symbol[i].length,
12                  symbol[i].type,
13                  symbol[i].term,

```

```

14             term_symb[symbol[i].term],
15             symbol_name(i, filedesc));
16         }
17         fprintf (filedesc, "\n");
18     }
19     else{
20         fprintf (filedesc, "Symbol Table is empty!\n\n");
21     }
22 }

```

## 5.2 分析树相关代码

下面主要是输入分析树的相关代码，在执行 `init_ast` 函数初始化抽象语法树后，开始执行下述代码。如果当前结点的索引  $<$  总结点数，并且  $> 0$ ，则进行相应的输出，`traverse` 函数是开始遍历 AST，抽象语法树。若是索引 `n` 不满足上述条件，则输出错误信息。

```

1 void print_ast (int n){ // Print subtree.
2     if (n < n_nodes && n > 0){
3         char indent [512];
4         strcpy (indent, draw_space);
5
6         fprintf(out_file_fp, "Abstract Syntax Tree ...\n\n");
7         fprintf(out_file_fp, "  node  prev  next parent child  line   sti \n");
8
9         printf("Abstract Syntax Tree ...\n\n");
10        printf("  node  prev  next parent child  line   sti \n");
11
12        traverse(indent, n); // Start AST traversal.
13
14        fprintf(out_file_fp, "\n");
15        printf("\n");
16    }else{
17        fprintf(out_file_fp, "Internal error, node %d is not in AST.\n\n", n);
18        printf("Internal error, node %d is not in AST.\n\n", n);
19    }
20 }

```

