EXPERIMENT 3

实验 3　使用 Lex 自动生成扫描程序

实验难度：★★☆☆☆
建议学时：2 学时

一、实验目的

- 掌握 Lex 输入文件的格式。
- 掌握使用 Lex 自动生成扫描程序的方法。

二、预备知识

- 要求已经学习了正则表达式的编写方法，能够正确使用“*”“?”“+”等基本的元字符，并且学习了 Lex 程序中定义的特有的元字符，例如“[ ]”“\n”等。
- 了解了标识符和关键字的识别方法。
- 本实验使用 Lex 的一个实现版本——GNU Flex 作为扫描程序。

三、实验内容

3.1　准备实验

按照下面的步骤准备实验：

(1) 启动 CP Lab。

(2) 在“文件”菜单中选择“新建”|“项目”，打开“新建项目”对话框。

(3) 使用模板“003 使用 Lex 自动生成扫描程序”新建一个项目。

3.2　阅读实验源代码

1. sample.txt 文件(参见源代码清单 3-1)

在 sample.txt 文件中是一个使用 TINY 语言编写的小程序，这个小程序在执行时从标准输入(键盘)读取一个整数，计算其阶乘后显示到标准输出(显示器)。

在本实验中并不需要执行这个小程序，只需使用 Lex 生成的扫描程序对这个 TINY 源代码文件进行扫描，最后统计出各种符号的数量。TINY 语言中的符号说明可以参考表 3-1。

表 3-1

序 号	符号或语句	说　明
1	{…}	在两个大括号之间的是注释
2	read	从标准输入(键盘)读取数据。是一个关键字
3	write	将数据写入标准输出(屏幕)。是一个关键字
4	if…then…else…end	if 语句。if 的后面是一个布尔表达式,then 和 else 的后面是一个语句块,其中 else 是可选的。end 表示结束。包括了 4 个关键字
5	repeat…until…	repeat 语句。repeat 后面是一个语句块,until 后面是一个布尔表达式。包括了两个关键字
6	<、=	比较运算符。<是小于符号;=是等于符号
7	+、-、*、/	算术运算符
8	:=	赋值运算符
9	正整数	由数字 0～9 组成
10	标识符	由大写字母和小写字母组成
11	;	在语句的结束位置有一个分号

2. define.h 文件(参见源代码清单 3-2)

在此文件中定义了一个枚举类型,对应于 TINY 语言中的各种符号。注意,此文件中还包括了“文件结束”和“错误”,其中 yylex 函数在遇到文件结束时,默认会返回 0,所以将“文件结束”定义在开始位置,这样它的值也为 0。

3. scan.txt 文件(参见源代码清单 3-3)

此文件是 Lex 的输入文件。根据 Lex 输入文件的格式,此文件分为 3 个部分(由%%分隔),各个部分的说明可以参见表 3-2。

表 3-2

名　称	说　明
第一部分	• 在%{和%}之间直接插入 C 源代码文件的内容。包括了要包含的头文件,以及 TINY 语言中各种符号的计数器,用于保存扫描的结果 • 定义了换行(newline)和空白(whitespace)的正则表达式。注意,空白包括了空格和制表符。
第二部分	包含了一组规则,当规则中的正则表达式匹配时,yylex 函数会执行规则提供的源代码。主要包含下面的规则: • 比较运算符、算术运算符等的规则。这些规则直接使用字符串进行匹配。若匹配成功,yylex 函数返回对应的枚举值 • 匹配换行的规则。匹配成功,就统计行数。 • 匹配空白的规则。匹配成功,就忽略。 • 由于编写注释的正则表达式比较复杂,所以在匹配注释的开始符号后,直接编写 C 代码来匹配注释的结束符号。这里用到的 input 函数是一个 Lex 的内部函数,它会返回一个输入字符。 • 使用 Lex 正则表达式中的“.”元字符匹配其他的任何字符。当之前的规则均匹配失败时,就会在此规则匹配成功,从而返回错误类型。
第三部分	这部分中的内容会直接插入 C 源代码文件。此部分内容的说明可以参见表 3-3。

表 3-3

内 容	说 明
main 函数	主函数。这里用到了 C 语言定义的 main 函数的两个参数，这两个参数的用法可以参考函数的注释 在 main 函数中首先使用 fopen 函数打开了待处理的文件，然后将此文件作为 Lex 扫描程序的输入(yyin)，之后调用 yylex 函数开始扫描，并调用 stat 函数统计各种符号的数量，最后调用 output 函数输出统计的结果。
id2keyword 函数	此函数将标识符转换为对应的关键字类型，如果通过参数传入的字符串不能与任何关键字匹配，就仍然返回标识符类型。此函数的函数体还不完整，留给读者完成
stat 函数	此函数根据 tt 参数传入的符号类型，增加符号类型对应的计数器。如果 tt 是标识符类型(ID)，会首先调用 id2keyword 函数，尝试将标识符类型转换为对应的关键字类型
output 函数	输出统计的结果

4. main.c 文件

此文件默认是一个空文件。Lex 根据输入文件生成的 C 源代码会输出到此文件中。此文件会包含自动生成的 yylex 函数的定义，此函数实现了与输入文件相对应的 DFA 表驱动。

源代码清单 3-1：sample.txt 文件

```
{ Sample program
  int TINY language-
  computes factorial
}
read x; { input an integer }
if 0<x then { don't compute if x<=0}
  fact :=1;
  repeat
    fact :=fact * x;
    x :=x-1
  until
    x=0;
  write fact { output factorial of x }
end
```

源代码清单 3-2：define.h 文件

```
#ifndef _DEFINE_H_
#define _DEFINE_H_

typedef enum
{
    //文件结束
    ENDFILE,
```

```
    //错误
    ERROR,

    //关键字
    IF,                 //if
    THEN,               //then
    ELSE,               //else
    END,                //end
    REPEAT,             //repeat
    UNTIL,              //until
    READ,               //read
    WRITE,              //write

    //标识符
    ID,

    //无符号整数
    NUM,

    //特殊符号
    ASSIGN,             //:=
    EQ,                 //=
    LT,                 //<
    PLUS,               //+
    MINUS,              //-
    TIMES,              //*
    OVER,               ///
    LPAREN,             //(
    RPAREN,             //)
    SEMI,               //;

    //注释
    COMMENT             //{...}

}TokenType;

#endif                  //_DEFINE_H_
```

源代码清单 3-3：scan.txt 文件

```
%{

#include<stdio.h>
#include "define.h"
```

```
int lineno=0;            //行数

//符号计数器
int error_no=0;

int if_no=0;
int then_no=0;
int else_no=0;
int end_no=0;
int repeat_no=0;
int until_no=0;
int read_no=0;
int write_no=0;

int id_no=0;
int num_no=0;

int assign_no=0;
int eq_no=0;
int lt_no=0;
int plus_no=0;
int minus_no=0;
int times_no=0;
int over_no=0;
int lparen_no=0;
int rparen_no=0;
int semi_no=0;

int comment_no=0;

%}

newline             \n
whitespace    [\t]+

%%

":="                { return ASSIGN; }
"="                 { return EQ; }
"<"                 { return LT; }
"+"                 { return PLUS; }
"-"                 { return MINUS; }
"*"                 { return TIMES; }
"/"                 { return OVER; }
```

```
"("                     { return LPAREN; }
")"                     { return RPAREN; }
";"                     { return SEMI; }

{newline}               { lineno++; }
{whitespace}            { /* 忽略空白 */ }

"{"                     {
                            //匹配注释{...}
                            char c;
                            int comment=1;
                            do
                            {
                                c=input();
                                if(c==EOF)
                                {
                                    comment=0;
                                    break;
                                }
                                else if(c=='\n')
                                    lineno++;
                                else if(c=='}')
                                    break;
                            }while(1);

                            return comment ? COMMENT : ERROR;
                        }

.                       { return ERROR; }

%%

TokenType id2keyword(const char * token);
void stat(TokenType tt, const char * token);
void output();

/*
功能:
    主函数。

参数:
    argc-argv 数组的长度,大小至少为 1,argc-1 为命令行参数的数量。
```

```
    argv-字符串指针数组,数组长度为命令行参数个数+1。其中,argv[0]固定指向当前所
        执行的可执行文件的路径字符串,argv[1]及后面的指针指向各个命令行参数。
        例如,通过命令行输入"C:\hello.exe-a-b"后,main 函数的 argc 的值为 3,
        argv[0]指向字符串"C:\hello.exe",argv[1]指向字符串"-a",argv[2]指向字
        符串"-b"。

返回值:
    成功返回 0,失败返回 1。
*/
int main(int argc, char *argv[])
{
    TokenType tt;

    //使用第一个参数输入待处理文件的名称,若没有输入此参数,就报告错误
    if(argc<2)
    {
        printf("Usage: scan.exe filename.\n");
        return 1;
    }

    //打开待处理的文件
    FILE *file=fopen(argv[1], "rt");
    if(NULL==file)
    {
        printf("Can not open file \"%s\".\n", argv[1]);
        return 1;
    }

    //将打开的文件作为 lex 扫描程序的输入
    yyin=file;

    //开始扫描,直到文件结束
    while((tt=yylex())!=ENDFILE)
    {
        //根据符号类型统计其数量
        stat(tt, yytext);
    }

    //输出统计结果
    output();

    //关闭文件
    fclose(file);
```

```
    return 0;
}

//定义关键字与其类型的映射关系
typedef struct _KeyWord_Entry
{
    const char * word;
    TokenType type;
}KeyWord_Entry;

static const KeyWord_Entry key_table[]=
{
    { "if",     IF         },
    { "then",   THEN    },
    { "else",   ELSE    },
    { "end",    END     },
    { "repeat", REPEAT    },
    { "until",  UNTIL      },
    { "read",   READ    },
    { "write",  WRITE      }
};

/*
功能:
    将标识符转换为对应的关键字类型。

参数:
    id-标识符字符串指针。可能是一个关键字,也可能是用户定义的标识符。

返回值:
    成功返回 0,失败返回 1。
*/
TokenType id2keyword(const char * id)
{
    //
    //TODO: 在此添加源代码
    //

    return ID;
}

/*
功能:
```

```
    根据符号类型进行数量统计。

参数:
    tt-符号类型。
    token-符号字符串指针。当符号被识别为标识符时,需要判断其是否为一个关键字。

返回值:
    空。
*/
void stat(TokenType tt, const char * token)
{
    if(ID==tt)
    {
        tt=id2keyword(token);
    }

    switch(tt)
    {
    case IF:        //if
        if_no++;
        break;
    case THEN:      //then
        then_no++;
        break;
    case ELSE:      //else
        else_no++;
        break;
    case END:       //end
        end_no++;
        break;
    case REPEAT:    //repeat
        repeat_no++;
        break;
    case UNTIL:     //until
        until_no++;
        break;
    case READ:      //read
        read_no++;
        break;
    case WRITE:     //write
        write_no++;
        break;
    case ID:        //标识符
```

```
        id_no++;
        break;
    case NUM:        //无符号整数
        num_no++;
        break;
    case ASSIGN:     //:=
        assign_no++;
        break;
    case EQ:         //=
        eq_no++;
        break;
    case LT:         //<
        lt_no++;
        break;
    case PLUS:       //+
        plus_no++;
        break;
    case MINUS:      //-
        minus_no++;
        break;
    case TIMES:      //*
        times_no++;
        break;
    case OVER:       // /
        over_no++;
        break;
    case LPAREN:     //(
        lparen_no++;
        break;
    case RPAREN:     //)
        rparen_no++;
        break;
    case SEMI:       //;
        semi_no++;
        break;
    case COMMENT:    //{...}
        comment_no++;
        break;
    case ERROR:      //错误
        error_no++;
        break;
    }
}
```

```
//输出统计结果
void output()
{
    printf("if: %d\n", if_no);
    printf("then: %d\n", then_no);
    printf("else: %d\n", else_no);
    printf("end: %d\n", end_no);
    printf("repeat: %d\n", repeat_no);
    printf("until: %d\n", until_no);
    printf("read: %d\n", read_no);
    printf("write: %d\n", write_no);

    printf("id: %d\n", id_no);
    printf("num: %d\n", num_no);

    printf("assign: %d\n", assign_no);
    printf("eq: %d\n", eq_no);
    printf("lt: %d\n", lt_no);
    printf("plus: %d\n", plus_no);
    printf("minus: %d\n", minus_no);
    printf("times: %d\n", times_no);
    printf("over: %d\n", over_no);
    printf("lparen: %d\n", lparen_no);
    printf("rparen: %d\n", rparen_no);
    printf("semi: %d\n", semi_no);

    printf("comment: %d\n", comment_no);
    printf("error: %d\n", error_no);
    printf("line: %d\n", lineno);
}
```

3.3 生成项目

按照下面的步骤生成项目：

(1) 在“生成”菜单中选择“重新生成项目”(快捷键是 Ctrl+Alt+F7)。

(2) 在生成的过程中，CP Lab 会首先使用 Flex 程序根据输入文件 scan.txt 来生成 main.c 文件，然后，将 main.c 文件重新编译、链接为可以运行的可执行文件。

(3) 在生成的 main.c 文件中，尝试找到 scan.txt 文件中第一部分和第三部分 C 源代码插入的位置，并尝试查找 input 函数和 yylex 函数的定义，以及 yyin 和 yytext 等变量的定义。

注意：在下面的实验步骤中，如果需要生成项目，应尽量使用“重新生成项目”功能。如果习惯使用“生成项目”(快捷键是 F7)功能，可能需要连续使用两次此功能才能生成最

新的项目。

3.4 运行项目

在没有对项目的源代码进行任何修改的情况下，按照下面的步骤运行项目：

(1) 选择“调试”菜单中的“开始执行(不调试)”(快捷键是 Ctrl+F5)。

(2) 在启动运行时，CP Lab 会自动将 sample.txt 文件的名称作为参数传给可执行文件(即 main 函数中 argv[1]指向的字符串)，所以，程序运行完毕后，会在 Windows 控制台窗口中显示对 sample.txt 文件的扫描结果，如图 3-1 所示。

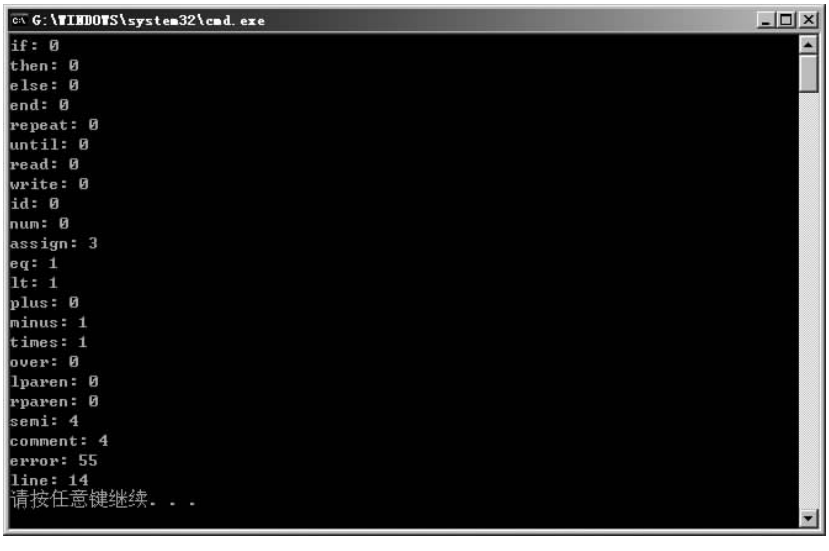

图 3-1 对 sample.txt 文件的扫描结果

由于此时还没有在 scan.txt 中添加能够与标识符和正整数匹配的正则表达式，所以 id 和 num 的值均为 0，而标识符和正整数会被匹配为错误，所以，error 的数量大于 0。

3.5 添加标识符和正整数的统计功能

按照下面的步骤完成此练习：

(1) 在 scan.txt 文件的第一部分添加标识符和正整数的正则表达式。

(2) 在 scan.txt 文件的第二部分添加标识符和正整数的正则表达式匹配时的 C 源代码。注意，标识符的正则表达式也用来匹配所有的关键字，不要直接使用字符串来逐个匹配关键字。

(3) 重新生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。

(4) 按 Ctrl+F5 键启动执行项目。

执行的结果如图 3-2 所示，已经可以正确统计出标识符和正整数的数量。注意，由于此时 id2keyword 函数还无法将标识符转换为关键字，所以，所有的关键字都匹配成了标识符，关键字的数量都为 0。

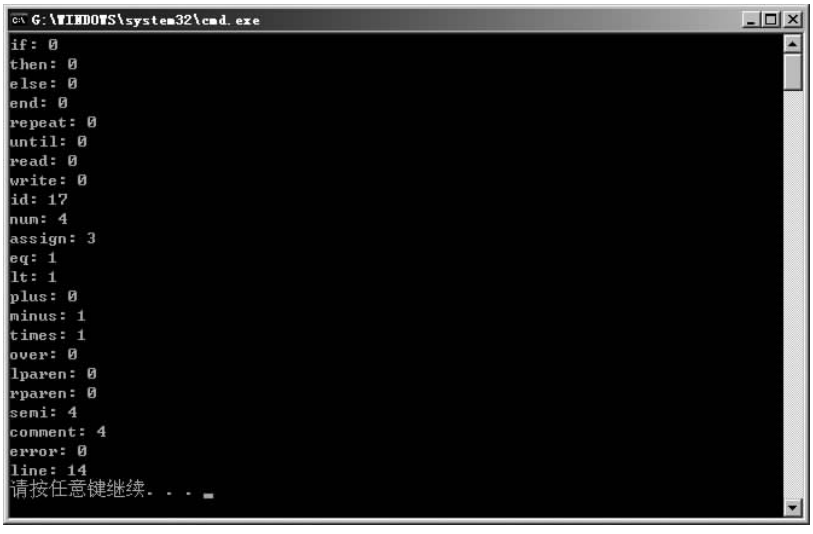

图 3-2　正确统计出标识符和正整数的数量

3.6　添加关键字的统计功能

按照下面的步骤完成此练习：

(1) 为 id2keyword 函数编写源代码。要求使用此函数前面定义的 key_table 表格中的数据，通过线性搜索的方式，根据标识符的字符串确定其对应的关键字类型。

(2) 重新生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。

(3) 按 Ctrl＋F5 键启动执行项目。

执行的结果如图 3-3 所示，已经可以正确统计出各个关键字的数量。

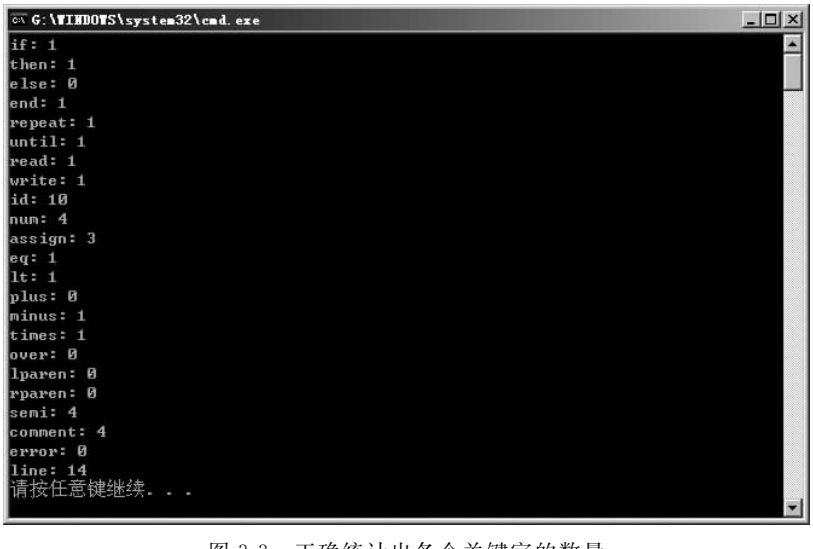

图 3-3　正确统计出各个关键字的数量

注意：

(1) 本实验的模板不提供演示功能，所以，如果执行的结果不正确，可以通过添加断点和单步调试的方法来查找错误的原因。

(2) 断点应该添加在 scan.txt 文件中需要中断的 C 源代码行，不要添加在 main.c 文件中，否则无法命中断点。

3.7 添加 C 语言风格的注释

按照下面的步骤完成此练习：

(1) 将 sample.txt 文件中的多行注释包括在“/*”和“*/”之间，将语句末尾的注释放在“//”的后面。

(2) 修改 scan.txt 文件，使之能够正确匹配和统计这两种 C 语言风格的注释。

(3) 重新生成项目。如果生成失败，根据“输出”窗口中的提示信息修改源代码中的语法错误。

(4) 按 Ctrl+F5 键启动执行项目，确保统计的注释数量是正确的。

四、思考与练习

1. 修改 key_table 表格中数据的顺序，使关键字按照字母顺序排列，然后修改 id2keyword 函数中的代码，使用二分法从 key_table 表格中查找关键字。

2. 使用 gperf 工具为 TINY 语言的关键字生成杂凑表(哈希表)，然后修改 id2keyword 函数中的代码从杂凑表中查找关键字。(**提示**：CP Lab 提供了 gperf 工具，可以选择 CP Lab“工具”菜单中的“CP Lab 命令提示”，在弹出的 Windows 控制台窗口中输入命令“gperf C:\a.txt --output-file=C:\a.c”，即可为 a.txt 文件中列出的关键字生成杂凑表到 a.c 源代码文件中。选择 CP Lab“帮助”菜单中“其他帮助文档”中的“gperf 手册”可以获得更多帮助。)

3. 选择 CP Lab“帮助”菜单中“其他帮助文档”中的“Flex 手册”，学习 Flex 工具的更多用法。如果需要修改 Flex 程序在处理输入文件时的选项，可以在“项目管理器”窗口中右击文件 scan.txt，在弹出的快捷菜单中选择“属性”，然后选择“属性页”左侧“自定义生成步骤”的“常规”，就可以编辑右侧的 “命令行”选项了。

EXPERIMENT 4

实验 4　消除左递归(无替换)

实验难度：★★★☆☆
建议学时：2 学时

一、实验目的

- 了解在上下文无关文法中的左递归的概念。
- 掌握直接左递归的消除算法。

二、预备知识

- 在这个实验中用到了单链表插入和删除操作。如果读者对这一部分知识有遗忘，可以复习一下数据结构中的相关内容。
- 理解指针的指针的概念和用法。在这个实验中，指针的指针被用来确定单链表插入的位置，以及插入的具体操作。
- 理解左递归和右递归的含义以及左递归的各种形式，如简单直接左递归、普遍的直接左递归、一般的左递归。

三、实验内容

3.1　准备实验

按照下面的步骤准备实验：

(1) 启动 CP Lab。

(2) 在“文件”菜单中选择“新建”|“项目”，打开“新建项目”对话框。

(3) 使用模板“004 消除左递归(无替换)”新建一个项目。

3.2　阅读实验源代码

1. RemoveLeftRecursion.h 文件(参见源代码清单 4-1)

此文件主要定义了与文法相关的数据结构，这些数据结构定义了文法的单链表存储形式。其中 Rule 结构体用于定义文法的名称和文法链表；RuleSymbol 结构体用于定义文法产生式中的终结符和非终结符。具体内容可参见表 4-1 和表 4-2。

表 4-1

Rule 的域	说　明
RuleName	文法的名称
pFirstSymbol	指向文法的第一个 Select 的第一个 Symbol
pNextRule	指向下一条文法

表 4-2

RuleSymbol 的域	说　明
pNextSymbol	指向下一个 Symbol
pOther	指向下一个 Select
isToken	是否为终结符。1 表示终结符,0 表示非终结符
TokenName	终结符的名称。isToken 为 1 时这个域有效
pRule	指向 Symbol 对应的 Rule。isToken 为 0 时这个域有效

下面是一个简单文法,并使用图 4-1 和图 4-2 说明了该文法的存储结构:

```
A->Aa|aB
B->bB
```

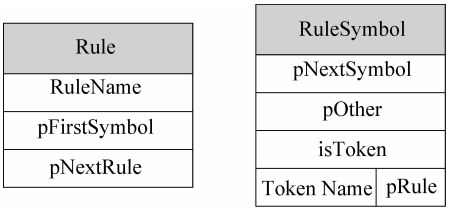

图 4-1　Rule 和 RuleSymbol 结构体图例

2. main.c 文件(参见源代码清单 4-2)

此文件定义了 main 函数。在 main 函数中首先调用 InitRules 函数初始化了文法,然后调用 PrintRule 函数打印消除左递归之前的文法,接着调用 RemoveLeftRecursion 函数对文法消除左递归,最后再次调用 PrintRule 函数打印消除左递归之后的文法。

在 main 函数的后面定义了一系列函数,有关这些函数的具体内容参见表 4-3。关于这些函数的参数和返回值,可以参见其注释。

表 4-3

函　数　名	功 能 说 明
AddSymbolToSelect	将一个 Symbol 添加到 Select 的末尾。此函数的函数体还不完整,留给读者完成
AddSelectToRule	将 Select 加入到文法末尾,如果 Select 为 NULL 则将 ε 终结符加入到文法末尾。在本程序中 ε 可以用 $ 来代替。此函数的函数体还不完整,留给读者完成

续表

函 数 名	功 能 说 明
RemoveLeftRecursion	对文法消除左递归。在本函数中使用指针的指针 pSelectPtr 来确定符号在单链表中插入和删除的位置，在进入下一次循环之前应为 pSelectPtr 设置正确的值。此函数的函数体还不完整，留给读者完成
InitRules	使用给定的数据初始化文法链表
CreateRule	创建一个新的 Rule
CreateSymbol	创建一个新的 Symbol
FindRule	根据 RuleName 在文法链表中查找名字相同的文法
PrintRule	输出文法。此函数的函数体还不完整，留给读者完成

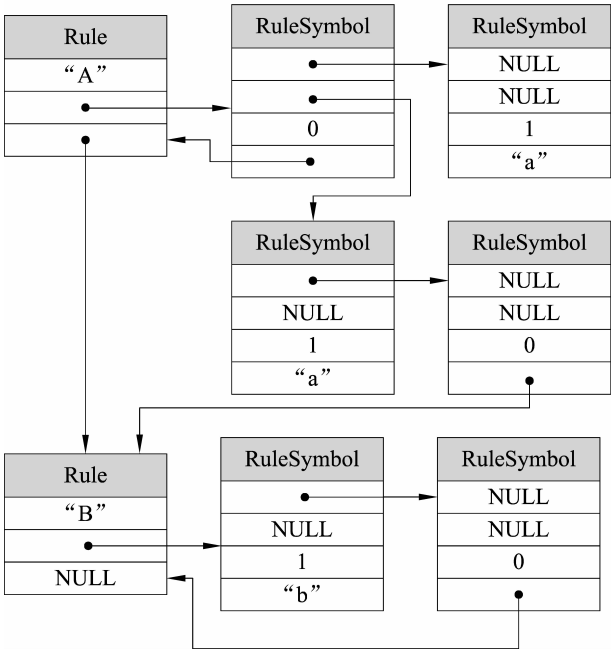

图 4-2 简单文法的存储结构

源代码清单 4-1：RemoveLeftRecursion.h 文件

```
#ifndef _REMOVELEFTRECURSIONNOREPLACE_H_
#define _REMOVELEFTRECURSIONNOREPLACE_H_

//
//在此处包含 C 标准库头文件
//

#include<stdio.h>
```

```
//
//在此处包含其他头文件
//
//
//在此处定义数据结构
//

#define MAX_STR_LENGTH 64

struct _Rule;
typedef struct _RuleSymbol{
    struct _RuleSymbol * pNextSymbol;  //指向下一个 Symbol
    struct _RuleSymbol * pOther;       //指向下一个 Select
    int isToken;                      //是否为终结符。1 表示终结符,0 表示非终结符
    char TokenName[MAX_STR_LENGTH];
                                       //终结符的名称。isToken 为 1 时这个域有效
    struct _Rule * pRule;
                              //指向 Symbol 对应的 Rule。isToken 为 0 时这个域有效
}RuleSymbol;

typedef struct _Rule{
    char RuleName[MAX_STR_LENGTH];     //文法的名称
    struct _RuleSymbol * pFirstSymbol;  //指向文法的第一个 Select 的第一个 Symbol
    struct _Rule * pNextRule;          //指向下一条文法
}Rule;

//
//在此处声明函数
//

Rule * InitRules();
Rule * CreateRule(const char * pRuleName);
RuleSymbol * CreateSymbol();
Rule * FindRule(Rule * pHead, const char * RuleName);

void AddSymbolToSelect(RuleSymbol * pSelect, RuleSymbol * pNewSymbol);
void AddSelectToRule(Rule * pRule, RuleSymbol * pNewSelect);
void RemoveLeftRecursion(Rule * pHead);

void PrintRule(Rule * pHead);

//
//在此处声明全局变量
```

```
//

extern const char * VoidSymbol;
extern const char * Postfix;

#endif /* _REMOVELEFTRECURSIONNOREPLACE_H_ */
```

源代码清单 4-2：main.c 文件

```
#include "RemoveLeftRecursion.h"

const char * VoidSymbol="$ "; //"ε"
const char * Postfix="'";

int main(int argc, char * argv[])
{
    //
    //调用 InitRules 函数初始化文法
    //
    Rule * pHead=InitRules();

    //
    //输出消除左递归之前的文法
    //
    printf("Before Remove Left Recursion:\n");
    PrintRule(pHead);

    //
    //调用 RemoveLeftRecursion 函数对文法消除左递归
    //
    RemoveLeftRecursion(pHead);

    //
    //输出消除左递归之后的文法
    //
    printf("\nAfter Remove Left Recursion:\n");
    PrintRule(pHead);

    return 0;
}

/*
功能：
```

```
    将一个 Symbol 添加到 Select 的末尾。

参数:
    pSelect--Select 指针。
    pNewSymbol--Symbol 指针。
*/
void AddSymbolToSelect(RuleSymbol *pSelect, RuleSymbol *pNewSymbol)
{

    //
    //TODO: 在此处添加代码
    //

}

/*
功能:
    将一个 Select 加入到文法末尾。当 Select 为 NULL 时就将一个 ε 终结符加入到文法末尾。

参数:
    pRule--文法指针。
    pNewSelect--Select 指针。
*/
void AddSelectToRule(Rule *pRule, RuleSymbol *pNewSelect)
{

    //
    //TODO: 在此处添加代码
    //

}

/*
功能:
    消除左递归。

参数:
    pHead--文法链表的头指针。
*/
void RemoveLeftRecursion(Rule *pHead)
{
    RuleSymbol *pSelect;        //Select 游标
```